THIRD EDITION

Game Engine Architecture

Jason Gregory

CRC Press
Taylor & Francis Group

THIRD EDITION

Game Engine Architecture

Jason Gregory

CRC Press
Taylor & Francis Group

游戏引擎架构

Game Engine Architecture

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

游戏引擎架构

第三版

杰森 · 格雷戈里

CRC 出版社 Taylor & Francis
集团 博卡拉顿 伦敦 纽约

CRC Press 是
Taylor & Francis Group，一家信息业务公司
AK 彼得斯的书

Game Engine Architecture

Third Edition

Jason Gregory

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

封面图片：Brian Hauger (www.bionic3d.com) 创建的 SpaceX Merlin 火箭发动机 3D 模型。

CRC出版社
泰勒弗朗西斯集团
6000 Broken Sound Parkway NW，300 室
佛罗里达州博卡拉顿 33487-2742

© 2019 Taylor & Francis Group, LLC CRC Press 是 Taylor & Francis Group 的印记，隶属于 Informa 旗下

不主张对美国政府原创作品的所有权

印于无酸纸上
版本日期：20180529

国际标准书号-13：978-1-1380-3545-4（精装本）

本书信息来源可靠且备受推崇。我们已尽合理努力确保数据和信息的可靠性，但作者和出版商不对所有材料的有效性及其使用后果承担责任。作者和出版商已尽力追踪本书中所有转载材料的版权所有者，如未获得以此类形式发表的许可，我们谨向版权所有者致歉。如有任何版权材料未注明出处，请致函告知，以便我们在今后的转载中予以更正。

除美国版权法允许外，未经出版商书面许可，不得以任何电子、机械或其他方式（现在已知或以后发明的）或任何信息存储或检索系统重印、复制、传播或利用本书的任何部分，包括影印、缩微胶片和录音。

如需获得复印或以电子方式使用本作品材料的许可，请访问 www.copyright.com (http://www.copyright.com/) 或联系版权许可中心 (CCC)，地址：222 Rosewood Drive, Danvers, MA 01923，电话：978-750-8400。CCC 是一家非营利组织，为各类用户提供许可和注册服务。对于已获得 CCC 复印许可的组织，我们将安排单独的付款系统。

商标声明：产品或公司名称可能是商标或注册商标，仅用于识别和解释，不具有侵权意图。

美国国会图书馆出版编目数据

姓名：格雷戈里，杰森，1970 年作家。
标题：游戏引擎架构/Jason Gregory。
描述：第三版。| 博卡拉顿：Taylor & Francis，CRC Press，2018 年。| 包括参考书目和索引。
标识符：LCCN 2018004893 | ISBN 9781138035454（精装：碱性纸）
主题：LCSH：电脑游戏——编程——电脑程序。| 软件架构。| 电脑游戏——设计。
分类：LCC QA76.76.C672 G77 2018 | DDC 794.8/1525--dc23
LC 记录可在 https://lccn.loc.gov/2018004893 获取

访问 Taylor & Francis 网站：http://www.taylorandfrancis.com

以及 CRC Press 网站 http://www.crcpress.com

Cover image: 3D model of SpaceX Merlin rocket engine created by Brian Hauger (www.bionic3d.com).

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2019 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20180529

International Standard Book Number-13: 978-1-1380-3545-4 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Gregory, Jason, 1970- author.
Title: Game engine architecture / Jason Gregory.
Description: Third edition. | Boca Raton : Taylor & Francis, CRC Press, 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2018004893 | ISBN 9781138035454 (hardback : alk. paper)
Subjects: LCSH: Computer games--Programming--Computer programs. | Software architecture. | Computer games--Design.
Classification: LCC QA76.76.C672 G77 2018 | DDC 794.8/1525--dc23
LC record available at https://lccn.loc.gov/2018004893

Visit the Taylor & Francis Web site at http://www.taylorandfrancis.com

and the CRC Press Web site at http://www.crcpress.com

献给 Trina、Evan 和 Quinn Gregory，以纪念我们的英雄 Joyce Osterhus、Kenneth Gregory 和 Erica Gregory。

Dedicated to
Trina, Evan and Quinn Gregory,

in memory of our heroes,
Joyce Osterhus, Kenneth Gregory and Erica Gregory.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

内容

前言 十三

I 基础知识 1 简介 1

1.1 3

典型游戏团队的结构 1.2 什么是游戏? 5

8

1.3 什么是游戏引擎? 11

1.4 不同游戏类型的引擎差异 1.5 游戏引擎概览 1.6 运行时引擎架构 1.7 工具和资源管道 2 常用工具 2.1 版本控制 2.2 编译器、链接器和 IDE 2.3 分析工具 13 31 38 59

69

69

78

99

Contents

Preface xiii

I Foundations 1

1 Introduction 3

1.1 Structure of a Typical Game Team 5

1.2 What Is a Game? 8

1.3 What Is a Game Engine? 11

1.4 Engine Differences across Genres 13

1.5 Game Engine Survey 31

1.6 Runtime Engine Architecture 38

1.7 Tools and the Asset Pipeline 59

2 Tools of the Trade 69

2.1 Version Control 69

2.2 Compilers, Linkers and IDEs 78

2.3 Profiling Tools 99

2.4 内存泄漏和损坏检测 101
2.5 其他工具 102

游戏软件工程的 3 个基本原理 105
3.1 C++ 回顾和最佳实践 105
3.2 捕获和处理错误 119
3.3 数据、代码和内存布局 131
3.4 计算机硬件基础知识 164
3.5 内存架构 181

4 并行和并发编程 203
4.1 定义并发和并行 204
4.2 隐式并行 211
4.3 显式并行 225
4.4 操作系统基础知识 230
4.5 并发编程简介 256
4.6 线程同步原语 267
4.7 基于锁的并发问题 281
4.8 并发的一些经验法则 286
4.9 无锁并发 289
4.10 SIMD/矢量处理 331
4.11 GPGPU编程简介 348

5 游戏中的 3D 数学 359
5.1 在二维空间中求解三维问题 359
5.2 点和向量 360
5.3 矩阵 375
5.4 四元数 394
5.5 旋转表示的比较 403
5.6 其他有用的数学对象 407
5.7 随机数生成 412

2.4 Memory Leak and Corruption Detection 101
2.5 Other Tools 102

3 Fundamentals of Software Engineering for Games 105
3.1 C++ Review and Best Practices 105
3.2 Catching and Handling Errors 119
3.3 Data, Code and Memory Layout 131
3.4 Computer Hardware Fundamentals 164
3.5 Memory Architectures 181

4 Parallelism and Concurrent Programming 203
4.1 Defining Concurrency and Parallelism 204
4.2 Implicit Parallelism 211
4.3 Explicit Parallelism 225
4.4 Operating System Fundamentals 230
4.5 Introduction to Concurrent Programming 256
4.6 Thread Synchronization Primitives 267
4.7 Problems with Lock-Based Concurrency 281
4.8 Some Rules of Thumb for Concurrency 286
4.9 Lock-Free Concurrency 289
4.10 SIMD/Vector Processing 331
4.11 Introduction to GPGPU Programming 348

5 3D Math for Games 359
5.1 Solving 3D Problems in 2D 359
5.2 Points and Vectors 360
5.3 Matrices 375
5.4 Quaternions 394
5.5 Comparison of Rotational Representations 403
5.6 Other Useful Mathematical Objects 407
5.7 Random Number Generation 412

II 低级引擎系统 415

6 发动机支持系统 417
6.1 子系统启动和关闭 417
6.2 内存管理 426
6.3 容器 441
6.4 字符串 456
6.5 引擎配置 470

7 资源和文件系统 481
7.1 文件系统 482
7.2 资源管理器 493

8 游戏循环和实时模拟 525
8.1 渲染循环 525
8.2 游戏循环 526
8.3 游戏循环架构风格 529
8.4 抽象时间线 532
8.5 测量和处理时间 534
8.6 多处理器游戏循环 544

9 人机接口设备 559
9.1 人机接口设备的类型 559
9.2 与 HID 接口 561
9.3 输入类型 563
9.4 输出类型 569
9.5 游戏引擎 HID 系统 570
9.6 人机接口设备实践 587

10 个调试和开发工具 589
10.1 日志记录和跟踪 589
10.2 调试绘图工具 594
10.3 游戏内菜单 601
10.4 游戏内控制台 604
10.5 调试相机和暂停游戏 605
10.6 秘籍 606

II Low-Level Engine Systems 415

6 Engine Support Systems 417
6.1 Subsystem Start-Up and Shut-Down 417
6.2 Memory Management 426
6.3 Containers 441
6.4 Strings 456
6.5 Engine Configuration 470

7 Resources and the File System 481
7.1 File System 482
7.2 The Resource Manager 493

8 The Game Loop and Real-Time Simulation 525
8.1 The Rendering Loop 525
8.2 The Game Loop 526
8.3 Game Loop Architectural Styles 529
8.4 Abstract Timelines 532
8.5 Measuring and Dealing with Time 534
8.6 Multiprocessor Game Loops 544

9 Human Interface Devices 559
9.1 Types of Human Interface Devices 559
9.2 Interfacing with a HID 561
9.3 Types of Inputs 563
9.4 Types of Outputs 569
9.5 Game Engine HID Systems 570
9.6 Human Interface Devices in Practice 587

10 Tools for Debugging and Development 589
10.1 Logging and Tracing 589
10.2 Debug Drawing Facilities 594
10.3 In-Game Menus 601
10.4 In-Game Console 604
10.5 Debug Cameras and Pausing the Game 605
10.6 Cheats 606

10.7 屏幕截图和影片捕捉 606
10.8 游戏内分析 608
10.9 游戏内内存统计和泄漏检测 615

III 图形、动作和声音 619

11 渲染引擎 621
11.1 深度缓冲三角形光栅化基础 622
11.2 渲染管线 667
11.3 高级照明和全局照明 697
11.4 视觉效果和叠加 710
11.5 进一步阅读 719

12个动画系统 721
12.1 角色动画的类型 721
12.2 骨骼 727
12.3 姿势 729
12.4 剪辑 734
12.5 皮肤和矩阵调色板生成 750
12.6 动画混合 755
12.7 后处理 774
12.8 压缩技术 777
12.9 动画管线 784
12.10 动作状态机 786
12.11 约束 806

13 碰撞和刚体动力学 817
13.1 你想在游戏中加入物理效果吗? 818
13.2 碰撞/物理中间件 823
13.3 碰撞检测系统 825
13.4 刚体动力学 854
13.5 将物理引擎集成到游戏中 892
13.6 高级物理特性 909

10.7 Screenshots and Movie Capture 606
10.8 In-Game Profiling 608
10.9 In-Game Memory Stats and Leak Detection 615

III Graphics, Motion and Sound 619

11 The Rendering Engine 621
11.1 Foundations of Depth-Buffered Triangle Rasterization 622
11.2 The Rendering Pipeline 667
11.3 Advanced Lighting and Global Illumination 697
11.4 Visual Effects and Overlays 710
11.5 Further Reading 719

12 Animation Systems 721
12.1 Types of Character Animation 721
12.2 Skeletons 727
12.3 Poses 729
12.4 Clips 734
12.5 Skinning and Matrix Palette Generation 750
12.6 Animation Blending 755
12.7 Post-Processing 774
12.8 Compression Techniques 777
12.9 The Animation Pipeline 784
12.10 Action State Machines 786
12.11 Constraints 806

13 Collision and Rigid Body Dynamics 817
13.1 Do You Want Physics in Your Game? 818
13.2 Collision/Physics Middleware 823
13.3 The Collision Detection System 825
13.4 Rigid Body Dynamics 854
13.5 Integrating a Physics Engine into Your Game 892
13.6 Advanced Physics Features 909

14 音频 911
14.1 声音的物理学 912
14.2 声音的数学 924
14.3 声音技术 941
14.4 以 3D 形式渲染音频 955
14.5 音频引擎架构 974
14.6 游戏专属音频功能 995

IV 游戏玩法 1013

15 游戏系统简介 1015
15.1 游戏世界的剖析 1016
15.2 实现动态元素：游戏对象 1021
15.3 数据驱动的游戏引擎 1024
15.4 游戏世界编辑器 1025

16 个运行时游戏基础系统 1039
16.1 游戏基础系统的组件 1039
16.2 运行时对象模型架构 1043
16.3 世界区块数据格式 1062
16.4 加载和流式传输游戏世界 1069
16.5 对象引用和世界查询 1079
16.6 实时更新游戏对象 1086
16.7 将并发性应用于游戏对象更新 1101
16.8 事件和消息传递 1114
16.9 脚本 1134
16.10 高级游戏流程 1157

V 结论 1159

17 你的意思是还有更多？ 1161
17.1 一些我们未涉及的发动机系统 1161
17.2 游戏系统 1162

参考书目 1167

指数 1171

14 Audio 911
14.1 The Physics of Sound 912
14.2 The Mathematics of Sound 924
14.3 The Technology of Sound 941
14.4 Rendering Audio in 3D 955
14.5 Audio Engine Architecture 974
14.6 Game-Specific Audio Features 995

IV Gameplay 1013

15 Introduction to Gameplay Systems 1015
15.1 Anatomy of a Game World 1016
15.2 Implementing Dynamic Elements: Game Objects 1021
15.3 Data-Driven Game Engines 1024
15.4 The Game World Editor 1025

16 Runtime Gameplay Foundation Systems 1039
16.1 Components of the Gameplay Foundation System 1039
16.2 Runtime Object Model Architectures 1043
16.3 World Chunk Data Formats 1062
16.4 Loading and Streaming Game Worlds 1069
16.5 Object References and World Queries 1079
16.6 Updating Game Objects in Real Time 1086
16.7 Applying Concurrency to Game Object Updates 1101
16.8 Events and Message-Passing 1114
16.9 Scripting 1134
16.10 High-Level Game Flow 1157

V Conclusion 1159

17 You Mean There's More? 1161
17.1 Some Engine Systems We Didn't Cover 1161
17.2 Gameplay Systems 1162

Bibliography 1167

Index 1171

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

前言

欢迎来到《游戏引擎架构》。本书旨在全面探讨构成典型商业游戏引擎的主要组件。游戏编程是一个庞大的话题，因此我们有很多内容需要探讨。尽管如此，我相信您一定会发现，我们深入的讨论足以让您对我们将要涵盖的每个工程学科的理论和常用实践有深入的理解。话虽如此，这本书实际上只是一段引人入胜且可能影响终身的旅程的开端。本书涵盖了游戏技术各个方面的丰富信息，它既可以作为基础工具，也可以作为进一步学习的起点。

本书将重点关注游戏引擎技术和架构。这意味着我们将涵盖构成商业游戏引擎的各个子系统的底层理论，通常用于实现它们的数据结构、算法和软件接口，以及这些子系统如何在整个游戏引擎中协同工作。游戏引擎和游戏之间的界限相当模糊。我们将主要关注引擎本身，包括一系列底层基础系统、渲染引擎、碰撞系统、物理模拟、角色动画、音频，并深入探讨我所说的游戏基础层。这一层包括游戏的对象模型、世界编辑器、事件系统和脚本系统。我们还会涉及游戏编程的一些方面——

Preface

Welcome to *Game Engine Architecture.* This book aims to present a complete discussion of the major components that make up a typical commercial game engine. Game programming is an immense topic, so we have a lot of ground to cover. Nevertheless, I trust you'll find that the depth of our discussions is sufficient to give you a solid understanding of both the theory and the common practices employed within each of the engineering disciplines we'll cover. That said, this book is really just the beginning of a fascinating and potentially lifelong journey. A wealth of information is available on all aspects of game technology, and this text serves both as a foundation-laying device and as a jumping-off point for further learning.

Our focus in this book will be on game engine technologies and architecture. This means we'll cover the theory underlying the various subsystems that comprise a commercial game engine, the data structures, algorithms and software interfaces that are typically used to implement them, and how these subsystems function together within a game engine as a whole. The line between the game engine and the game is rather blurry. We'll focus primarily on the engine itself, including a host of low-level foundation systems, the rendering engine, the collision system, the physics simulation, character animation, audio, and an in-depth discussion of what I call the gameplay foundation layer. This layer includes the game's object model, world editor, event system and scripting system. We'll also touch on some aspects of gameplay program-

ming，包括玩家机制、摄像头和AI。然而，出于必要，这些讨论的范围将主要限于游戏系统与引擎交互的方式。

本书旨在作为两到三门大学中级游戏编程系列课程的教材。业余软件工程师、游戏爱好者、自学成才的游戏程序员以及游戏行业的资深从业人员均可使用本书。初级工程师可以使用本书巩固对游戏数学、引擎架构和游戏技术的理解。而一些专注于某一特定专业的高级工程师，或许也能从本书所呈现的更广阔的视野中受益。

为了充分利用本书，你应该具备面向对象编程的基本概念，并至少具备一些 C++ 编程经验。游戏行业通常会使用各种各样的编程语言，但工业级的 3D 游戏引擎仍然主要使用 C++ 编写。因此，任何专业的游戏程序员都需要能够使用 C++ 编写代码。我们将在第 3 章回顾面向对象编程的基本原则，阅读本书时你无疑会学到一些新的 C++ 技巧，但扎实的 C++ 语言基础最好从 [46]、[36] 和 [37] 中获得。如果你对 C++ 有点生疏，我建议你在阅读本书时参考这些或类似的书籍来更新你的知识。如果你之前没有 C++ 经验，在深入阅读本书之前，你可能需要至少阅读 [46] 的前几章和/或在线学习一些 C++ 教程。

学习任何类型的计算机编程的最佳方式是实际编写一些代码。在阅读本书的过程中，我强烈建议你选择几个你特别感兴趣的主题领域，并在这些领域为自己设计一些项目。例如，如果你对角色动画感兴趣，可以先安装 OGRE 并探索其蒙皮动画演示。然后，你可以尝试使用 OGRE 实现本书中介绍的一些动画混合技术。接下来，你可以决定实现一个简单的由手柄控制的动画角色，它可以在平面上奔跑。一旦你完成了一些相对简单的工作，就可以对其进行扩展！然后转向游戏技术领域的其他领域。如此反复。项目是什么并不重要，只要你在实践游戏编程的艺术，而不是仅仅阅读相关内容即可。

游戏技术是一个活生生的东西，不可能在一本书里完全展现。因此，我们会不时在

ming, including player mechanics, cameras and AI. However, by necessity, the scope of these discussions will be limited mainly to the ways in which gameplay systems interface with the engine.

This book is intended to be used as a course text for a two- or three-course college-level series in intermediate game programming. It can also be used by amateur software engineers, hobbyists, self-taught game programmers and existing members of the game industry alike. Junior engineers can use this text to solidify their understanding of game mathematics, engine architecture and game technology. And some senior engineers who have devoted their careers to one particular specialty may benefit from the bigger picture presented in these pages as well.

To get the most out of this book, you should have a working knowledge of basic object-oriented programming concepts and at least some experience programming in C++. The game industry routinely makes use of a wide range of programming languages, but industrial-strength 3D game engines are still written primarily in C++. As such, any serious game programmer needs to be able to code in C++. We'll review the basic tenets of object-oriented programming in Chapter 3, and you will no doubt pick up a few new C++ tricks as you read this book, but a solid foundation in the C++ language is best obtained from [46], [36] and [37]. If your C++ is a bit rusty, I recommend you refer to these or similar books to refresh your knowledge as you read this text. If you have no prior C++ experience, you may want to consider reading at least the first few chapters of [46] and/or working through a few C++ tutorials online, before diving into this book.

The best way to learn computer programming of any kind is to actually write some code. As you read through this book, I strongly encourage you to select a few topic areas that are of particular interest to you and come up with some projects for yourself in those areas. For example, if you find character animation interesting, you could start by installing OGRE and exploring its skinned animation demo. Then you could try to implement some of the animation blending techniques described in this book, using OGRE. Next you might decide to implement a simple joypad-controlled animated character that can run around on a flat plane. Once you have something relatively simple working, expand upon it! Then move on to another area of game technology. Rinse and repeat. It doesn't particularly matter what the projects are, as long as you're practicing the art of game programming, not just reading about it.

Game technology is a living, breathing thing that can never be entirely captured within the pages of a book. As such, additional resources, errata, updates, sample code and project ideas will be posted from time to time on

本书的网站是http://www.gameenginebook.com。你也可以在Twitter上关注我：@jqgregory。

第三版新增内容

当今游戏机、移动设备和个人电脑的核心计算硬件大量使用了并行性。在这些设备的 CPU 和 GPU 内部，多个功能单元同时运行，采用“分而治之”的方法实现高速计算。虽然并行计算硬件可以加快传统单线程程序的运行速度，但程序员需要编写并发软件才能真正利用现代计算平台中无处不在的硬件并行性。

在《游戏引擎架构》之前的版本中，并行和并发这两个主题在游戏引擎设计的背景下有所涉及。然而，它们并没有得到应有的深入探讨。在本书的第三版中，这个问题得到了解决，增加了一个关于并发和并行的全新章节。第8章和第16章也进行了扩充，详细讨论了并发编程技术通常如何应用于游戏引擎子系统和游戏对象模型更新，以及如何使用通用作业系统来释放游戏引擎中并发的强大功能。

我已经提到过，每个优秀的游戏程序员都必须具备扎实的 C++ 工作知识（此外，还要掌握游戏行业中常用的各种其他实用语言）。在我看来，程序员对高级语言的掌握应该建立在对其底层软件和硬件系统的扎实理解之上。因此，在本版中，我扩展了第三章，涵盖了计算机硬件基础知识、汇编语言和操作系统内核。

《游戏引擎架构》第三版也改进了前几版中涉及的各种主题的处理。本书增加了对局部和全局编译器优化的讨论，并更全面地涵盖了各种 C++ 语言标准。本书扩展了关于内存缓存和缓存一致性的部分，并精简了动画章节。此外，与第二版一样，本书也修复了由我忠实的读者们指出的各种勘误。谢谢！希望各位发现的错误都已得到修复。（当然，之前发现的错误已经被大量新的错误所取代，欢迎各位随时告知我，以便我在本书第四版中更正它们！）

this book's website at http://www.gameenginebook.com. You can also follow me on Twitter `@jqgregory`.

New to the Third Edition

The computing hardware that lies at the heart of today's game consoles, mobile devices and personal computers makes heavy use of *parallelism.* Deep within the CPUs and GPUs in these devices, multiple functional units operate simultaneously, employing a "divide and conquer" approach to high-speed computation. While parallel computing hardware can make traditional single-threaded programs run faster, programmers need to write *concurrent software* to truly take advantage of the hardware parallelism that has become ubiquitous in modern computing platforms.

In prior editions of *Game Engine Architecture,* the topics of parallelism and concurrency were touched on in the context of game engine design. However, they weren't given the in-depth treatment they deserved. In this, the third edition of the book, this problem has been remedied via the addition of a brand new chapter on concurrency and parallelism. Chapters 8 and 16 have also been augmented to include detailed discussions of how concurrent programming techniques are typically applied to game engine subsystem and game object model updates, and how a general-purpose job system can be used to unlock the power of concurrency within a game engine.

I've already mentioned that every good game programmer must have a strong working knowledge of C++ (in addition to the wide variety of other useful languages used regularly in the game industry). In my view, a programmer's knowledge of high-level languages should rest upon a solid understanding of the software and hardware systems that underlie them. As such, in this edition I've expanded Chapter 3 to include a treatment of the fundamentals of computer hardware, assembly language, and the operating system kernel.

This third edition of *Game Engine Architecture* also improves upon the treatment of various topics covered in prior editions. A discussion of local and global compiler optimizations has been added. Fuller coverage of the various C++ language standards is included. The section on memory caching and cache coherency has been expanded. The animation chapter has been streamlined. And, as with the second edition, various errata have been repaired that were brought to my attention by you, my devoted readers. Thank you! I hope you'll find that the mistakes you found have all been fixed. (Although no doubt they have been replaced by a slew of *new* mistakes, about which you can feel free to inform me, so that I may correct them in the fourth edition of the book!)

当然，正如我之前所说，游戏引擎编程领域广博而深奥，几乎难以想象。一本书不可能涵盖所有主题。因此，本书的主要目的仍然是作为一种认知构建工具和进一步学习的起点。我希望本书能帮助您探索引人入胜且多面的游戏引擎架构领域。

致谢

没有一本书是凭空创作的，这本书当然也不例外。如果没有我的家人、朋友和游戏行业同事的帮助，这本书——以及你们现在手中的第三版——是不可能问世的。我要向所有帮助我完成这个项目的人致以最诚挚的谢意。

当然，受此类项目影响最大的，莫过于作者的家人。因此，首先，我要再次特别感谢我的妻子特丽娜。在创作原著期间，她一直是我的精神支柱；在我编写第二版和第三版的过程中，她也一如既往地给予我支持和无价的帮助。在我忙于敲击键盘时，特丽娜总是日复一日、夜复一夜地照顾着我们的两个儿子，埃文（现年15岁）和奎因（12岁）。她常常放弃自己的计划，帮我做家务（次数多得我都不愿承认），并且总是在我最需要的时候给予我鼓励的话语。

我还要特别感谢第一版的编辑Matt Whiting和Jeff Lander。他们富有洞察力、针对性强且及时的反馈总是恰到好处。他们在游戏行业的丰富经验使我相信本书中提供的信息尽可能准确且及时。与Matt和Jeff合作非常愉快，我很荣幸有机会与如此精湛的专业人士合作完成这个项目。我还要特别感谢Jeff，是他让我联系上了Alice Peters，并帮助我启动了这个项目。Matt，也感谢你再次挺身而出，为我提供了关于第三版新增并发章节的宝贵反馈。

顽皮狗的几位同事也为本书做出了贡献，他们或提供反馈，或帮助我完善其中某一章节的结构和主题内容。我要感谢 Marshall Robin 和 Carlos Gonzalez-Ochoa 在我撰写渲染章节时提供的指导和指导，以及 Pål-Kristian Engstad 提供的出色且富有洞察力的反馈。

Of course, as I've said before, the field of game engine programming is almost unimaginably broad and deep. There's no way to cover every topic in one book. As such, the primary purpose of this book remains to serve as an awareness-building tool and a jumping-off point for further learning. I hope you find this edition helpful on your journey through the fascinating and multifaceted landscape of game engine architecture.

Acknowledgments

No book is created in a vacuum, and this one is certainly no exception. This book—and its third edition, which you now hold in your hands—would not have been possible without the help of my family, friends and colleagues in the game industry, and I'd like to extend warm thanks to everyone who helped me to bring this project to fruition.

Of course, the ones most impacted by a project like this are invariably the author's family. So I'd like to start by offering, for a *third time*, a special thank-you to my wife Trina. She was a pillar of strength during the writing of the original book, and she was as supportive and invaluably helpful as ever during my work on the second and third editions. While I was busy tapping away on my keyboard, Trina was always there to take care of our two boys, Evan (now age 15) and Quinn (age 12), day after day and night after night, often forgoing her own plans, doing my chores as well as her own (more often than I'd like to admit), and always giving me kind words of encouragement when I needed them the most.

I would also like to extend special thanks to my editors for the first edition, Matt Whiting and Jeff Lander. Their insightful, targeted and timely feedback was always right on the money, and their vast experience in the game industry helped to give me confidence that the information presented in these pages is as accurate and up-to-date as humanly possible. Matt and Jeff were both a pleasure to work with, and I am honored to have had the opportunity to collaborate with such consummate professionals on this project. I'd like to extend a special thank-you to Jeff for putting me in touch with Alice Peters and helping me to get this project off the ground in the first place. Matt, thank you also for stepping up to the plate once again and providing me with valuable feedback on the new concurrency chapter in the third edition.

A number of my colleagues at Naughty Dog also contributed to this book, either by providing feedback or by helping me with the structure and topic content of one of the chapters. I'd like to thank Marshall Robin and Carlos Gonzalez-Ochoa for their guidance and tutelage as I wrote the rendering chapter, and Pål-Kristian Engstad for his excellent and insightful feedback

关于该章的内容。我要感谢 Christian Gyrling 对本书各个部分的反馈，包括关于动画的章节以及关于并行和并发的新章节。我要特别感谢顽皮狗的常驻高级音频程序员 Jonathan Lanier，他为我提供了音频章节中大量的原始信息，在我有问题时随时与我聊天，并在阅读初稿后提供了重点突出且非常宝贵的反馈。我还要感谢顽皮狗编程团队的最新成员之一 Kareem Omar，他对新的并发章节提供了宝贵的见解和反馈。我还要感谢整个顽皮狗工程团队，是他们创造了我在本书中重点介绍的所有令人难以置信的游戏引擎系统。

另外，还要感谢艺电公司的 Keith Schaeffer，他为我提供了大量关于物理效果对游戏影响的原始内容，这些内容见第 13.1 节。我还要衷心感谢 Christophe Balestra（我在顽皮狗工作的前十年，他担任该公司联席总裁）、Paul Keet（我在艺电工作期间，他担任《荣誉勋章》系列的首席工程师）以及 Steve Ranck（圣地亚哥 Midway 工作室 Hydro Thunder 项目的首席工程师），感谢他们多年来对我的指导。虽然他们没有直接参与本书的编写，但他们确实帮助我成为了如今的工程师，他们的影响几乎以某种方式体现在本书的每一页中。

本书源于我为 ITP-485 课程撰写的笔记：
我在南加州大学信息技术项目下教授了《游戏引擎编程》这门课程大约四年。我要感谢当时ITP系主任Anthony Borquez博士，他聘请我开发ITP-485课程。

我的亲朋好友也值得感谢，一方面是他们坚定不移的鼓励，另一方面是他们在我工作期间多次招待我的妻子和两个儿子。我要感谢我的姐夫特蕾西·李和姐夫道格·普罗文斯，我的表弟马特·格伦，以及我们所有了不起的朋友，包括金·克拉克和德鲁·克拉克夫妇、谢里琳和吉姆·克里策夫妇、安妮和迈克尔·谢勒夫妇、金·华纳和迈克·华纳夫妇，以及肯德拉和安迪·沃尔瑟夫妇。我十几岁的时候，父亲肯尼斯·格雷戈里写了《非凡的股票利润》——一本关于投资股票市场的书——这本书启发了我写这本书。为此以及更多，我永远感激他。我还要感谢我的母亲埃里卡·格雷戈里，一方面是她坚持让我开始这个项目，另一方面是她在我小时候花了无数的时间陪我打麻将。

on the content of that chapter. My thanks go to Christian Gyrling for his feedback on various sections of the book, including the chapter on animation, and the new chapter on parallelism and concurrency. I want to extend a special thank-you to Jonathan Lanier, Naughty Dog's resident senior audio programmer extraordinaire, for providing me with a great deal of the raw information you'll find in the audio chapter, for always being available to chat when I had questions, and for providing laser-focused and invaluable feedback after reading the initial draft. I'd also like to thank one of the newest members of our Naughty Dog programming team, Kareem Omar, for his valuable insights and feedback on the new concurrency chapter. My thanks also go to the entire Naughty Dog engineering team for creating all of the incredible game engine systems that I highlight in this book.

Additional thanks go to Keith Schaeffer of Electronic Arts for providing me with much of the raw content regarding the impact of physics on a game, found in Section 13.1. I'd also like to extend a warm thank-you to Christophe Balestra (who was co-president of Naughty Dog during my first ten years there), Paul Keet (who was a lead engineer on the *Medal of Honor* franchise during my time at Electronic Arts), and Steve Ranck (the lead engineer on the *Hydro Thunder* project at Midway San Diego), for their mentorship and guidance over the years. While they did not contribute to the book directly, they did help to make me the engineer that I am today, and their influences are echoed on virtually every page in one way or another.

This book arose out of the notes I developed for a course entitled ITP-485: Programming Game Engines, which I taught under the auspices of the Information Technology Program at the University of Southern California for approximately four years. I would like to thank Dr. Anthony Borquez, the director of the ITP department at the time, for hiring me to develop the ITP-485 course curriculum in the first place.

My extended family and friends also deserve thanks, in part for their unwavering encouragement, and in part for entertaining my wife and our two boys on so many occasions while I was working. I'd like to thank my sister- and brother-in-law, Tracy Lee and Doug Provins, my cousin-in-law Matt Glenn, and all of our incredible friends, including Kim and Drew Clark, Sherilyn and Jim Kritzer, Anne and Michael Scherer, Kim and Mike Warner, and Kendra and Andy Walther. When I was a teenager, my father Kenneth Gregory wrote *Extraordinary Stock Profits*—a book on investing in the stock market—and in doing so, he inspired me to write this book. For this and so much more, I am eternally grateful to him. I'd also like to thank my mother Erica Gregory, in part for her insistence that I embark on this project, and in part for spending countless hours with me when I was a child, beating the art

写作深深地烙印在我的脑海里。我的写作技巧、我的职业道德以及我那略带扭曲的幽默感都归功于她！

我要感谢 Alice Peters 和 Kevin Jackson-Mead 以及 AK Peters 的全体员工，感谢他们为出版本书第一版所付出的巨大努力。从那时起，AK Peters 就被 Taylor & Francis 集团的主要科技图书部门 CRC Press 收购。我祝愿 Alice 和 Klaus Peters 在未来的工作中一切顺利。我还要感谢 Taylor & Francis 的 Rick Adams、Jennifer Ahringer、Jessica Vega 和 Cynthia Klivecka 在创作《游戏引擎架构》第二版和第三版的过程中给予的耐心支持和帮助，感谢 Jonathan Pennell 为第二版设计的封面，感谢 Scott Shamblin 为第三版设计的封面，感谢 Brian Haeger (http://www.bionic3d.com) 慷慨地允许我在第三版的封面上使用他设计的漂亮的 Space X Merlin 火箭引擎 3D 模型。

我很高兴地宣布，《游戏引擎架构》第一版和第二版都已翻译成或正在翻译成日语、中文和韩语！我要衷心感谢万代南梦宫游戏的湊和久和他的团队承担了日语翻译这项极其艰巨的任务，并且出色地完成了两个版本的翻译工作。我还要感谢软银创意公司（Softbank Creative, Inc.）的各位同事出版了本书的日文版。我还要衷心感谢叶文龙（Milo Yip）为中文翻译项目付出的辛勤工作和奉献精神。我还要衷心感谢电子工业出版社出版了本书的中文版，以及橡果出版社和弘陵科学出版社分别出版了本书第一版和第二版的韩语版。

许多读者抽出时间给我反馈，指出第一版和第二版中的错误，为此，我要向所有为此做出贡献的人致以诚挚的谢意。我还要特别感谢 Milo Yip、Joe Conley 和 Zachary Turner，他们在这方面付出了超出职责范围的努力。你们三位都为我提供了数页的文档，其中充满了勘误表以及极其宝贵且富有洞察力的建议。我已尽力将所有反馈意见融入到第三版中——请继续提供！

杰森 · 格雷戈里
2018年4月

of writing into my cranium. I owe my writing skills, my work ethic, and my rather twisted sense of humor entirely to her!

I'd like to thank Alice Peters and Kevin Jackson-Mead, as well as the entire A K Peters staff, for their Herculean efforts in publishing the first edition of this book. Since that time, A K Peters has been acquired by the CRC Press, the principal science and technology book division of the Taylor & Francis Group. I'd like to wish Alice and Klaus Peters all the best in their future endeavors. I'd also like to thank Rick Adams, Jennifer Ahringer, Jessica Vega and Cynthia Klivecka of Taylor & Francis for their patient support and help throughout the process of creating the second and third editions of *Game Engine Architecture*, Jonathan Pennell for his work on the cover for the second edition, Scott Shamblin for his work on the third edition's cover art, and Brian Haeger (http://www.bionic3d.com) for graciously permitting me to use his beautiful 3D model of the Space X Merlin rocket engine on the cover of the third edition.

I am thrilled to be able to say that both the first and second editions of *Game Engine Architecture* have been or are being translated into Japanese, Chinese and Korean! I would like to extend my sincere thanks to Kazuhisa Minato and his team at Namco Bandai Games for taking on the incredibly daunting task of the Japanese translation, and for doing such a great job with both editions. I'd also like to thank the folks at Softbank Creative, Inc. for publishing the Japanese version of the book. I would also like to extend my warmest thanks to Milo Yip for his hard work and dedication to the Chinese translation project. My sincere appreciation goes to the Publishing House of the Electronics Industry for publishing the Chinese translation of the book, and to both the Acorn Publishing Company and Hongreung Science Publishing Co. for their publication of the Korean translations of the first and second editions, respectively.

Many of my readers took the time to send me feedback and alert me to errors in the first and second editions, and for that I'd like to extend my sincere thanks to all of you who contributed. I'd like to give a special thank-you to Milo Yip, Joe Conley and Zachary Turner for going above and beyond the call of duty in this regard. All three of you provided me with many-page documents, chock full of errata and incredibly valuable and insightful suggestions. I've tried my best to incorporate all of this feedback into the third edition—please keep it coming!

Jason Gregory
April 2018

第一部分 基础

Part I Foundations

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1 介绍

1979 年，当我拥有第一台游戏机（美泰公司出品的超酷 Intellivision 系统）时，“游戏引擎”一词还不存在。那时，大多数成年人认为电子游戏和街机游戏不过是玩具，驱动它们的软件高度专业化，既针对游戏本身，也针对运行游戏的硬件。如今，游戏已成为价值数十亿美元的主流产业，其规模和受欢迎程度可与好莱坞相媲美。驱动这些如今无处不在的三维世界的软件——游戏引擎，如 Epic Games 的虚幻引擎 4、Valve 的 Source 引擎、Crytek 的 CRYENGINE® 3、Electronic Arts DICE 的 FrostbiteTM 引擎和 Unity 游戏引擎——已成为功能齐全、可重复使用的软件开发工具包，可以获得许可并用于构建几乎任何可以想象到的游戏。

虽然游戏引擎在架构和实现细节上差异巨大，但无论是公开授权的游戏引擎还是其专有的内部引擎，都呈现出了可识别的粗粒度模式。几乎所有游戏引擎都包含一组熟悉的核心组件，包括渲染引擎、碰撞和物理引擎、动画系统、音频系统、游戏世界对象模型、人工智能系统等等。在每个组件中，也开始出现相对较少的半标准设计方案。

1 Introduction

When I got my first game console in 1979—a way-cool Intellivision system by Mattel—the term “game engine” did not exist. Back then, video and arcade games were considered by most adults to be nothing more than toys, and the software that made them tick was highly specialized to both the game in question and the hardware on which it ran. Today, games are a multi-billion-dollar mainstream industry rivaling Hollywood in size and popularity. And the software that drives these now-ubiquitous three-dimensional worlds—*game engines* like Epic Games’ Unreal Engine 4, Valve’s Source engine and, Crytek’s CRYENGINE® 3, Electronic Arts DICE’s Frostbite™ engine, and the Unity game engine—have become fully featured reusable software development kits that can be licensed and used to build almost any game imaginable.

While game engines vary widely in the details of their architecture and implementation, recognizable coarse-grained patterns have emerged across both publicly licensed game engines and their proprietary in-house counterparts. Virtually all game engines contain a familiar set of core components, including the rendering engine, the collision and physics engine, the animation system, the audio system, the game world object model, the artificial intelligence system and so on. Within each of these components, a relatively small number of semi-standard design alternatives are also beginning to emerge.

市面上有很多书籍详尽地介绍了各个游戏引擎子系统，例如三维图形。还有一些书籍则拼凑了各种游戏技术领域的宝贵技巧和窍门。然而，我至今仍未找到一本书，能够为读者提供构成现代游戏引擎的所有组件的合理完整图景。因此，本书的目标是带领读者亲身体验游戏引擎架构的广阔而复杂的格局。

在本书中你将了解到：

- 如何构建真正的工业强度生产游戏引擎；
- 游戏开发团队在现实世界中如何组织和工作；
- 哪些主要子系统和设计模式在几乎每个游戏引擎中反复出现；
- 每个主要子系统的典型要求；
- 哪些子系统与游戏类型无关，哪些子系统通常是专门为特定类型或游戏设计的；
- 引擎通常结束和游戏开始的地方。

我们还将亲身体验一些热门游戏引擎（例如 Quake、Unreal 和 Unity）的内部工作原理，以及一些知名的中间件包，例如 Havok Physics 库、OGRE 渲染引擎以及 Rad Game Tools 的 Granny 3D 动画和几何管理工具包。此外，我们还将探索一些我有幸使用过的专有游戏引擎，包括顽皮狗为其《神秘海域》和《最后生还者》系列游戏开发的引擎。

在开始之前，我们将回顾一些游戏引擎环境中大规模软件工程的技术和工具，包括：

- 逻辑和物理软件架构之间的区别；
- 配置管理、修订控制和构建系统；以及
- 处理 C 和 C++ 的常见开发环境之一 Microsoft Visual Studio 的一些技巧和窍门。

在本书中，我假设你对 C++（大多数现代游戏开发者的首选语言）有扎实的理解，并且了解基本的软件工程原理。我还假设你具备一些

There are a great many books that cover individual game engine subsystems, such as three-dimensional graphics, in exhaustive detail. Other books cobble together valuable tips and tricks across a wide variety of game technology areas. However, I have been unable to find a book that provides its reader with a reasonably complete picture of the entire gamut of components that make up a modern game engine. The goal of this book, then, is to take the reader on a guided hands-on tour of the vast and complex landscape of game engine architecture.

In this book you will learn:

- how real industrial-strength production game engines are architected;
- how game development teams are organized and work in the real world;
- which major subsystems and design patterns appear again and again in virtually every game engine;
- the typical requirements for each major subsystem;
- which subsystems are genre- or game-agnostic, and which ones are typically designed explicitly for a specific genre or game; and
- where the engine normally ends and the game begins.

We'll also get a first-hand glimpse into the inner workings of some popular game engines, such as Quake, Unreal and Unity, and some well-known middleware packages, such as the Havok Physics library, the OGRE rendering engine and Rad Game Tools' Granny 3D animation and geometry management toolkit. And we'll explore a number of proprietary game engines that I've had the pleasure to work with, including the engine Naughty Dog developed for its *Uncharted* and *The Last of Us* game series.

Before we get started, we'll review some techniques and tools for large-scale software engineering in a game engine context, including:

- the difference between logical and physical software architecture;
- configuration management, revision control and build systems; and
- some tips and tricks for dealing with one of the common development environments for C and C++, Microsoft Visual Studio.

In this book I assume that you have a solid understanding of C++ (the language of choice among most modern game developers) and that you understand basic software engineering principles. I also assume you have some

接触线性代数、三维向量和矩阵数学以及三角学（尽管我们将在第5章回顾核心概念）。理想情况下，你应该已经了解实时和事件驱动编程的基本概念。不过不用担心——我会简要回顾这些主题，如果你觉得在开始之前需要进一步磨练技能，我还会为你指明正确的方向。

1.1 典型游戏团队的结构

在深入探讨典型游戏引擎的架构之前，我们先来简单了解一下典型游戏开发团队的架构。游戏工作室通常由五大类基本人员组成：工程师、美术师、游戏设计师、制作人以及其他管理和支持人员（市场营销、法务、信息技术/技术支持、行政等）。每个部门又可细分为多个子部门。下文我们将分别进行简要介绍。

1.1.1 工程师

工程师负责设计和实现游戏和工具运行所需的软件。工程师通常分为两大类：运行时程序员（负责引擎和游戏本身的开发）和工具程序员（负责开发离线工具，以便开发团队其他成员高效工作）。在运行时/工具的两端，工程师们各有专长。一些工程师专注于单一引擎系统，例如渲染、人工智能、音频或碰撞和物理。一些工程师专注于游戏编程和脚本编写，而另一些工程师则更倾向于在系统层面工作，不太参与游戏的实际运行方式。一些工程师是多面手——他们能够胜任各种工作，并解决开发过程中可能出现的任何问题。

高级工程师有时会被要求承担技术领导角色。
首席工程师通常仍设计和编写代码，但他们也帮助管理团队的日程安排，就项目的整体技术方向做出决策，有时还从人力资源的角度直接管理人员。

一些公司还会设立一名或多名技术总监 (TD)，其职责是从高层监督一个或多个项目，确保团队了解潜在的技术挑战、即将到来的行业发展、新技术等等。游戏工作室的最高工程相关职位是首席技术官 (CTO)（如果工作室设有）。

exposure to linear algebra, three-dimensional vector and matrix math and trigonometry (although we'll review the core concepts in Chapter 5). Ideally, you should have some prior exposure to the basic concepts of real time and event-driven programming. But never fear—I will review these topics briefly, and I'll also point you in the right direction if you feel you need to hone your skills further before we embark.

1.1 Structure of a Typical Game Team

Before we delve into the structure of a typical game engine, let's first take a brief look at the structure of a typical game development team. Game studios are usually composed of five basic disciplines: engineers, artists, game designers, producers and other management and support staff (marketing, legal, information technology/technical support, administrative, etc.). Each discipline can be divided into various subdisciplines. We'll take a brief look at each below.

1.1.1 Engineers

The engineers design and implement the software that makes the game, and the tools, work. Engineers are often categorized into two basic groups: *runtime* programmers (who work on the engine and the game itself) and *tools* programmers (who work on the offline tools that allow the rest of the development team to work effectively). On both sides of the runtime/tools line, engineers have various specialties. Some engineers focus their careers on a single engine system, such as rendering, artificial intelligence, audio or collision and physics. Some focus on gameplay programming and scripting, while others prefer to work at the systems level and not get too involved in how the game actually plays. Some engineers are generalists—jacks of all trades who can jump around and tackle whatever problems might arise during development.

Senior engineers are sometimes asked to take on a technical leadership role. Lead engineers usually still design and write code, but they also help to manage the team's schedule, make decisions regarding the overall technical direction of the project, and sometimes also directly manage people from a human resources perspective.

Some companies also have one or more technical directors (TD), whose job it is to oversee one or more projects from a high level, ensuring that the teams are aware of potential technical challenges, upcoming industry developments, new technologies and so on. The highest engineering-related position at a game studio is the chief technical officer (CTO), if the studio has one. The

CTO 的职责是担任整个工作室的技术总监，同时在公司中担任关键的执行角色。

1.1.2 艺术家

正如游戏行业常说的，“内容为王”。美术师负责制作游戏中所有的视觉和音频内容，他们作品的质量实际上可以成就一款游戏，也可以毁掉一款游戏。美术师的职业背景多种多样：

- 概念艺术家创作草图和绘画，为团队提供游戏最终外观的愿景。他们在开发概念阶段的早期就开始工作，但通常会在整个项目生命周期中持续提供视觉指导。从已发行游戏中截取的截图与概念图惊人地相似是很常见的。

- 3D 建模师负责为虚拟游戏世界中的一切创建三维几何图形。该领域通常分为两个分支：前景建模师和背景建模师。前者负责创建游戏世界中的物体、角色、车辆、武器和其他对象，而后者则负责构建游戏世界中的静态背景几何图形（地形、建筑物、桥梁等）。

- 纹理艺术家创建称为纹理的二维图像，将其应用于 3D 模型的表面以提供细节和真实感。

- 灯光艺术家布置游戏世界中所有的光源，包括静态和动态，并利用颜色、强度和光线方向来最大限度地提高每个场景的艺术性和情感影响力。

- 动画师为游戏中的角色和物体注入动作。
  动画师在游戏制作中扮演着演员的角色，就像在CG电影制作中一样。然而，游戏动画师必须具备一套独特的技能，才能制作出与游戏引擎技术基础无缝衔接的动画。

- 动作捕捉演员通常用于提供一组粗略的动作数据，然后由动画师进行清理和调整，然后集成到游戏中。

- 声音设计师与工程师密切合作，制作和混合游戏中的音效和音乐。

CTO's job is to serve as a sort of technical director for the entire studio, as well as serving a key executive role in the company.

1.1.2 Artists

As we say in the game industry, "Content is king." The artists produce all of the visual and audio content in the game, and the quality of their work can literally make or break a game. Artists come in all sorts of flavors:

- *Concept artists* produce sketches and paintings that provide the team with a vision of what the final game will look like. They start their work early in the concept phase of development, but usually continue to provide visual direction throughout a project's life cycle. It is common for screenshots taken from a shipping game to bear an uncanny resemblance to the concept art.

- *3D modelers* produce the three-dimensional geometry for everything in the virtual game world. This discipline is typically divided into two subdisciplines: foreground modelers and background modelers. The former create objects, characters, vehicles, weapons and the other objects that populate the game world, while the latter build the world's static background geometry (terrain, buildings, bridges, etc.).

- *Texture artists* create the two-dimensional images known as textures, which are applied to the surfaces of 3D models in order to provide detail and realism.

- *Lighting artists* lay out all of the light sources in the game world, both static and dynamic, and work with color, intensity and light direction to maximize the artfulness and emotional impact of each scene.

- *Animators* imbue the characters and objects in the game with motion. The animators serve quite literally as actors in a game production, just as they do in a CG film production. However, a game animator must have a unique set of skills in order to produce animations that mesh seamlessly with the technological underpinnings of the game engine.

- *Motion capture actors* are often used to provide a rough set of motion data, which are then cleaned up and tweaked by the animators before being integrated into the game.

- *Sound designers* work closely with the engineers in order to produce and mix the sound effects and music in the game.

- 许多游戏中的角色均由配音演员配音。
- 许多游戏都有一位或多位作曲家，他们为游戏创作原创乐谱。

与工程师一样，高级艺术家也经常被要求担任团队领导。一些游戏团队有一个或多个艺术总监——非常资深的艺术家，他们管理整个游戏的外观并确保所有团队成员的工作一致性。

1.1.3 游戏设计师

游戏设计师的工作是设计玩家体验的互动部分，通常称为游戏玩法。不同类型的设计师在不同的细节层面上工作。一些（通常是高级）游戏设计师在宏观层面工作，确定故事情节、章节或关卡的整体顺序以及玩家的高级目标。其他设计师则致力于虚拟游戏世界中的各个关卡或地理区域，布置静态背景几何形状，确定敌人出现的地点和时间，放置武器和健康包等补给品，设计谜题元素等。还有一些设计师在高技术层面上工作，与游戏玩法工程师紧密合作和/或编写代码（通常使用高级脚本语言）。一些游戏设计师是前工程师，他们决定在确定游戏玩法方面发挥更积极的作用。

有些游戏团队会雇佣一名或多名编剧。游戏编剧的工作范围很广，从与高级游戏设计师合作构建整个游戏的故事情节，到编写单独的对话。

与其他学科一样，一些高级设计师担任管理角色。许多游戏团队都有游戏总监，其职责是监督游戏设计的各个方面，协助管理进度，并确保每位设计师的工作在整个产品中保持一致。高级设计师有时也会成为制作人。

1.1.4 生产者

不同工作室对制作人角色的定义有所不同。在一些游戏公司，制作人的工作是管理日程安排并担任人力资源经理。在其他公司，制作人则担任高级游戏设计师。还有一些工作室要求制作人担任开发团队与公司业务部门（财务、法务、市场营销等）之间的联络人。一些规模较小的工作室甚至没有制作人。例如

- *Voice actors* provide the voices of the characters in many games.
- Many games have one or more *composers*, who compose an original score for the game.

As with engineers, senior artists are often called upon to be team leaders. Some game teams have one or more *art directors*—very senior artists who manage the look of the entire game and ensure consistency across the work of all team members.

1.1.3 Game Designers

The game designers' job is to design the interactive portion of the player's experience, typically known as *gameplay*. Different kinds of designers work at different levels of detail. Some (usually senior) game designers work at the macro level, determining the story arc, the overall sequence of chapters or levels, and the high-level goals and objectives of the player. Other designers work on individual levels or geographical areas within the virtual game world, laying out the static background geometry, determining where and when enemies will emerge, placing supplies like weapons and health packs, designing puzzle elements and so on. Still other designers operate at a highly technical level, working closely with gameplay engineers and/or writing code (often in a high-level scripting language). Some game designers are ex-engineers, who decided they wanted to play a more active role in determining how the game will play.

Some game teams employ one or more *writers*. A game writer's job can range from collaborating with the senior game designers to construct the story arc of the entire game, to writing individual lines of dialogue.

As with other disciplines, some senior designers play management roles. Many game teams have a game director, whose job it is to oversee all aspects of a game's design, help manage schedules, and ensure that the work of individual designers is consistent across the entire product. Senior designers also sometimes evolve into producers.

1.1.4 Producers

The role of producer is defined differently by different studios. In some game companies, the producer's job is to manage the schedule and serve as a human resources manager. In other companies, producers serve in a senior game design capacity. Still other studios ask their producers to serve as liaisons between the development team and the business unit of the company (finance, legal, marketing, etc.). Some smaller studios don't have producers at all. For

例如，在顽皮狗，公司里的每个人，包括两位联席总裁，都直接参与了游戏的构建；团队管理和业务职责由工作室的高级成员分担。

1.1.5 其他人员

直接构建游戏的团队通常会得到一个关键的支持团队的支持。这包括工作室的执行管理团队、市场部门（或与外部市场部门联络的团队）、行政人员以及IT部门，IT部门的职责是为团队购买、安装和配置硬件和软件，并提供技术支持。

1.1.6 出版商和工作室

游戏的营销、制作和发行通常由发行商负责，而不是由游戏工作室本身。发行商通常是一家大公司，例如 Electronic Arts、THQ、Vivendi、Sony、Nintendo 等。许多游戏工作室并不隶属于特定的发行商。他们将自己制作的每款游戏卖给与其达成最佳交易的任何发行商。其他工作室则只与单一发行商合作，通过长期出版合同或作为出版公司的全资子公司。例如，THQ 的游戏工作室是独立管理的，但它们归 THQ 所有并最终由 THQ 控制。Electronic Arts 通过直接管理其工作室将这种关系更进一步。第一方开发商是游戏机制造商（索尼、任天堂和微软）直接拥有的游戏工作室。例如，顽皮狗是索尼的第一方开发商。这些工作室专门为其母公司生产的游戏硬件制作游戏。

1.2 什么是游戏?

我们可能都对游戏的概念有着相当直观的理解。“游戏”这个通用术语涵盖了棋盘游戏（例如国际象棋和大富翁）、纸牌游戏（例如扑克和二十一点）、赌场游戏（例如轮盘赌和老虎机）、军事战争游戏、电脑游戏以及各种儿童游戏等等。在学术界，我们有时会提到博弈论，其中多个智能体会选择策略和战术，以便在一套明确定义的游戏规则框架内最大化收益。当用于游戏机或电脑娱乐时，“游戏”一词

example, at Naughty Dog, literally everyone in the company, including the two co-presidents, plays a direct role in constructing the game; team management and business duties are shared between the senior members of the studio.

1.1.5 Other Staff

The team of people who directly construct the game is typically supported by a crucial team of support staff. This includes the studio's executive management team, the marketing department (or a team that liaises with an external marketing group), administrative staff and the IT department, whose job is to purchase, install and configure hardware and software for the team and to provide technical support.

1.1.6 Publishers and Studios

The marketing, manufacture and distribution of a game title are usually handled by a *publisher*, not by the game studio itself. A publisher is typically a large corporation, like Electronic Arts, THQ, Vivendi, Sony, Nintendo, etc. Many game studios are not affiliated with a particular publisher. They sell each game that they produce to whichever publisher strikes the best deal with them. Other studios work exclusively with a single publisher, either via a long-term publishing contract or as a fully owned subsidiary of the publishing company. For example, THQ's game studios are independently managed, but they are owned and ultimately controlled by THQ. Electronic Arts takes this relationship one step further, by directly managing its studios. *First-party developers* are game studios owned directly by the console manufacturers (Sony, Nintendo and Microsoft). For example, Naughty Dog is a first-party Sony developer. These studios produce games exclusively for the gaming hardware manufactured by their parent company.

1.2 What Is a Game?

We probably all have a pretty good intuitive notion of what a game is. The general term "game" encompasses board games like chess and *Monopoly*, card games like poker and blackjack, casino games like roulette and slot machines, military war games, computer games, various kinds of play among children, and the list goes on. In academia we sometimes speak of *game theory*, in which multiple agents select strategies and tactics in order to maximize their gains within the framework of a well-defined set of game rules. When used in the context of console or computer-based entertainment, the word "game"

提到“游戏”，人们通常会联想到一个三维虚拟世界，其中的主角是人形生物、动物或交通工具，由玩家操控。（对于我们这些老玩家来说，也许会联想到二维经典游戏，比如《乒乓》、《吃豆人》或《大金刚》。）拉夫·科斯特在其优秀著作《游戏设计的乐趣理论》中将游戏定义为一种互动体验，它为玩家提供一系列挑战性不断递增的模式，玩家可以学习并最终掌握这些模式 [30]。科斯特认为，学习和掌握这些活动是我们所说的“乐趣”的核心，就像一个笑话在我们通过识别模式“理解”它的那一刻就变得好笑一样。

本书将重点关注那些包含少量玩家（大约 1 到 16 人）的二维和三维虚拟世界游戏。本书的大部分内容也适用于互联网上的 HTML5/JavaScript 游戏、像《俄罗斯方块》这样的纯益智游戏，以及大型多人在线游戏 (MMOG)。但我们的主要关注点将放在能够制作第一人称射击游戏、第三人称动作/平台游戏、赛车游戏、格斗游戏等的游戏引擎上。

1.2.1 电子游戏作为软实时模拟

大多数二维和三维视频游戏都是计算机科学家所说的基于代理的软实时交互式计算机模拟的例子。
让我们分解一下这个短语，以便更好地理解它的含义。

在大多数电子游戏中，现实世界的某个子集（或虚拟世界）会被数学建模，以便计算机进行操作。该模型是对现实（即使是虚构的现实）的近似和简化，因为要包含所有细节，直至原子或夸克的层面，显然是不切实际的。因此，数学模型是对真实或虚拟游戏世界的模拟。近似和简化是游戏开发者最强大的两大工具。如果运用得当，即使是一个极其简化的模型，有时也能与现实几乎难以区分，而且乐趣十足。

基于代理的模拟是指多个被称为“代理”的不同实体相互作用的模拟。这非常符合大多数三维电脑游戏的描述，其中的代理包括车辆、角色、火球、能量点等等。鉴于大多数游戏都基于代理的特性，如今大多数游戏都采用面向对象（或至少是松散的基于对象的）编程语言实现也□□就不足为奇了。

usually conjures images of a three-dimensional virtual world featuring a humanoid, animal or vehicle as the main character under player control. (Or for the old geezers among us, perhaps it brings to mind images of two-dimensional classics like *Pong*, *Pac-Man*, or *Donkey Kong*.) In his excellent book, *A Theory of Fun for Game Design*, Raph Koster defines a game to be an interactive experience that provides the player with an increasingly challenging sequence of patterns which he or she learns and eventually masters [30]. Koster's assertion is that the activities of learning and mastering are at the heart of what we call "fun," just as a joke becomes funny at the moment we "get it" by recognizing the pattern.

For the purposes of this book, we'll focus on the subset of games that comprise two- and three-dimensional virtual worlds with a small number of players (between one and 16 or thereabouts). Much of what we'll learn can also be applied to HTML5/JavaScript games on the Internet, pure puzzle games like *Tetris*, or massively multiplayer online games (MMOG). But our primary focus will be on game engines capable of producing first-person shooters, third-person action/platform games, racing games, fighting games and the like.

1.2.1 Video Games as Soft Real-Time Simulations

Most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*. Let's break this phrase down in order to better understand what it means.

In most video games, some subset of the real world—or an imaginary world—is *modeled* mathematically so that it can be manipulated by a computer. The model is an approximation to and a simplification of reality (even if it's an *imaginary* reality), because it is clearly impractical to include every detail down to the level of atoms or quarks. Hence, the mathematical model is a *simulation* of the real or imagined game world. Approximation and simplification are two of the game developer's most powerful tools. When used skillfully, even a greatly simplified model can sometimes be almost indistinguishable from reality—and a lot more fun.

An *agent-based* simulation is one in which a number of distinct entities known as "agents" interact. This fits the description of most three-dimensional computer games very well, where the agents are vehicles, characters, fireballs, power dots and so on. Given the agent-based nature of most games, it should come as no surprise that most games nowadays are implemented in an object-oriented, or at least loosely object-based, programming language.

所有交互式视频游戏都是时间模拟，这意味着虚拟游戏世界模型是动态的——游戏世界的状态随着游戏事件和故事的展开而变化。视频游戏还必须响应来自人类玩家的不可预测的输入——因此是交互式时间模拟。最后，大多数视频游戏实时呈现故事并响应玩家输入，使其成为交互式实时模拟。一个值得注意的例外是回合制游戏，如电脑象棋或回合制策略游戏。但即使是这些类型的游戏通常也会为用户提供某种形式的实时图形用户界面。因此，就本书的目的而言，我们假设所有视频游戏都至少有一些实时约束。

每个实时系统的核心都是截止期限的概念。视频游戏中一个明显的例子是要求屏幕每秒至少更新 24 次以提供运动的幻觉。（大多数游戏以每秒 30 或 60 帧的速度渲染屏幕，因为这些是 NTSC 显示器刷新率的倍数。）当然，视频游戏中还有许多其他类型的截止期限。物理模拟可能需要每秒更新 120 次才能保持稳定。角色的人工智能系统可能需要每秒至少“思考”一次，以防止显得愚蠢。音频库可能需要每 1/60 秒至少调用一次，以保持音频缓冲区填满并防止出现可听见的故障。

在“软”实时系统中，错过最后期限不会造成灾难性的后果。因此，所有视频游戏都是软实时系统——即使帧率下降，人类玩家通常也不会受到影响！这与硬实时系统形成对比，在硬实时系统中，错过最后期限可能意味着人类操作员严重受伤甚至死亡。直升机的航空电子系统或核电站的控制棒系统就是硬实时系统的例子。

数学模型可以是解析的，也可以是数值的。例如，在重力作用下，刚体在恒定加速度作用下下落的解析（闭式）数学模型通常写成如下形式：

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0. \tag{1.1}$$

只需给定初始条件 v 0 和 y 0 以及常数 g ，就可以针对其自变量的任意值（例如上式中的时间 t）求出解析模型。如果能够找到这样的模型，那么它们会非常方便。然而，数学中的许多问题并没有闭式解。而在电子游戏中，由于用户的输入不可预测，我们无法指望用解析模型来模拟整个游戏。

同一刚体在重力作用下的数值模型可以表示为

All interactive video games are *temporal simulations*, meaning that the virtual game world model is *dynamic*—the state of the game world changes over time as the game's events and story unfold. A video game must also respond to unpredictable inputs from its human player(s)—thus *interactive temporal simulations*. Finally, most video games present their stories and respond to player input in real time, making them *interactive real-time simulations*. One notable exception is in the category of turn-based games like computerized chess or turn-based strategy games. But even these types of games usually provide the user with some form of real-time graphical user interface. So for the purposes of this book, we'll assume that all video games have at least *some* real-time constraints.

At the core of every real-time system is the concept of a *deadline*. An obvious example in video games is the requirement that the screen be updated at least 24 times per second in order to provide the illusion of motion. (Most games render the screen at 30 or 60 frames per second because these are multiples of an NTSC monitor's refresh rate.) Of course, there are many other kinds of deadlines in video games as well. A physics simulation may need to be updated 120 times per second in order to remain stable. A character's artificial intelligence system may need to "think" at least once every second to prevent the appearance of stupidity. The audio library may need to be called at least once every 1/60 second in order to keep the audio buffers filled and prevent audible glitches.

A "soft" real-time system is one in which missed deadlines are not catastrophic. Hence, all video games are *soft real-time systems*—if the frame rate dies, the human player generally doesn't! Contrast this with a *hard real-time system*, in which a missed deadline could mean severe injury to or even the death of a human operator. The avionics system in a helicopter or the control-rod system in a nuclear power plant are examples of hard real-time systems.

Mathematical models can be *analytic* or *numerical*. For example, the analytic (closed-form) mathematical model of a rigid body falling under the influence of constant acceleration due to gravity is typically written as follows:

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0. \tag{1.1}$$

An analytic model can be evaluated for any value of its independent variables, such as the time t in the above equation, given only the initial conditions v_0 and y_0 and the constant g. Such models are very convenient when they can be found. However, many problems in mathematics have no closed-form solution. And in video games, where the user's input is unpredictable, we cannot hope to model the entire game analytically.

A numerical model of the same rigid body under gravity can be expressed

如下：

$$y(t+\Delta t) = F(y(t), \dot{y}(t), \ddot{y}(t), \ldots). \tag{1.2}$$

也就是说，刚体在未来某个时间（t + Δ t）的高度可以通过当前时间 t 的高度及其一阶、二阶以及可能更高阶的时间导数的函数来找到。数值模拟通常通过重复运行计算来实现，以确定系统在每个离散时间步长的状态。游戏以相同的方式工作。主“游戏循环”重复运行，在循环的每次迭代中，各种游戏系统（如人工智能、游戏逻辑、物理模拟等）都有机会计算或更新下一个离散时间步长的状态。然后通过显示图形、发出声音以及可能产生其他输出（如游戏手柄上的力反馈）来“渲染”结果。

1.3 什么是游戏引擎?

“游戏引擎”一词出现于 20 世纪 90 年代中期，指的是第一人称射击 (FPS) 游戏，例如 id Software 开发的超人气游戏《毁灭战士》。《毁灭战士》的架构非常明确地将其核心软件组件（例如 3D 图形渲染系统、碰撞检测系统或音频系统）与构成玩家游戏体验的美术资源、游戏世界和游戏规则区分开来。随着开发者开始授权游戏并通过仅对“引擎”软件进行极少改动来创建新的美术、世界布局、武器、角色、车辆和游戏规则，从而将其重新制作成新产品，这种分离的价值变得显而易见。这标志着“mod 社区”的诞生——一群个人玩家和小型独立工作室，他们使用原始开发者提供的免费工具包修改现有游戏来创建新游戏。

20 世纪 90 年代末，一些游戏（例如《雷神之锤 III：竞技场》和虚幻引擎）在设计时就考虑到了重复使用和“模组化”的因素。引擎通过脚本语言（例如 id 的《雷神之锤 C》）实现了高度可定制性，引擎授权也开始成为开发者可行的第二收入来源。如今，游戏开发者可以授权游戏引擎，并重复使用其大部分关键软件组件来构建游戏。虽然这种做法仍然需要在定制软件工程方面投入大量资金，但比内部开发所有核心引擎组件要经济得多。

as follows:

$$y(t+\Delta t) = F(y(t), \dot{y}(t), \ddot{y}(t), \ldots). \tag{1.2}$$

That is, the height of the rigid body at some future time ($t + \Delta t$) can be found as a function of the height and its first, second, and possibly higher-order time derivatives at the current time t. Numerical simulations are typically implemented by running calculations repeatedly, in order to determine the state of the system at each discrete time step. Games work in the same way. A main “game loop” runs repeatedly, and during each iteration of the loop, various game systems such as artificial intelligence, game logic, physics simulations and so on are given a chance to calculate or update their state for the next discrete time step. The results are then “rendered” by displaying graphics, emitting sound and possibly producing other outputs such as force-feedback on the joypad.

1.3 What Is a Game Engine?

The term “game engine” arose in the mid-1990s in reference to first-person shooter (FPS) games like the insanely popular *Doom* by id Software. *Doom* was architected with a reasonably well-defined separation between its core software components (such as the three-dimensional graphics rendering system, the collision detection system or the audio system) and the art assets, game worlds and rules of play that comprised the player’s gaming experience. The value of this separation became evident as developers began licensing games and retooling them into new products by creating new art, world layouts, weapons, characters, vehicles and game rules with only minimal changes to the “engine” software. This marked the birth of the “mod community”—a group of individual gamers and small independent studios that built new games by modifying existing games, using free toolkits provided by the original developers.

Towards the end of the 1990s, some games like *Quake III Arena* and *Unreal* were designed with reuse and “modding” in mind. Engines were made highly customizable via scripting languages like id’s Quake C, and engine licensing began to be a viable secondary revenue stream for the developers who created them. Today, game developers can license a game engine and reuse significant portions of its key software components in order to build games. While this practice still involves considerable investment in custom software engineering, it can be much more economical than developing all of the core engine components in-house.

图 1.1. 游戏引擎可重用性范围。

游戏与其引擎之间的界限通常很模糊。有些引擎的区分相当清晰，而有些引擎则几乎没有尝试区分两者。在一款游戏中，渲染代码可能“知道”如何绘制兽人。而在另一个游戏中，渲染引擎可能提供通用材质和着色功能，“兽人特性”可能完全由数据定义。没有哪个工作室能够完美地区分游戏和引擎，考虑到这两个组件的定义经常随着游戏设计的成熟而发生变化，这也可以理解。

可以说，数据驱动架构正是游戏引擎与一款游戏而非引擎的软件之间的区别。当一款游戏包含硬编码的逻辑或游戏规则，或者使用特殊代码来渲染特定类型的游戏对象时，很难甚至不可能重用该软件来制作其他游戏。我们或许应该将“游戏引擎”一词保留给那些可扩展且无需进行重大修改即可作为许多不同游戏基础的软件。

显然，这并非黑白分明的区别。我们可以设想每个引擎都具备一定的可复用性。图 1.1 列出了一些知名游戏/引擎在这一范围内的定位。

人们可能会认为游戏引擎类似于 Apple QuickTime 或 Microsoft Windows Media Player——一款能够播放几乎所有可以想象到的游戏内容的通用软件。然而，这种理想状态尚未实现（甚至可能永远无法实现）。大多数游戏引擎都经过精心设计和微调，以在特定硬件平台上运行特定游戏。即使是最通用的多平台引擎，实际上也只适合构建某一特定类型的游戏，例如第一人称射击游戏或赛车游戏。可以肯定地说，游戏引擎或中间件组件越通用，它在特定平台上运行特定游戏的性能就越差。

出现这种现象是因为设计任何高效的软件都必然需要做出权衡，而这些权衡是基于对软件的使用方式和/或目标的假设

Figure 1.1. Game engine reusability gamut.

The line between a game and its engine is often blurry. Some engines make a reasonably clear distinction, while others make almost no attempt to separate the two. In one game, the rendering code might “know” specifically how to draw an orc. In another game, the rendering engine might provide general-purpose material and shading facilities, and “orc-ness” might be defined entirely in data. No studio makes a perfectly clear separation between the game and the engine, which is understandable considering that the definitions of these two components often shift as the game’s design solidifies.

Arguably a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine. When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a different game. We should probably reserve the term “game engine” for software that is extensible and can be used as the foundation for many different games without major modification.

Clearly this is not a black-and-white distinction. We can think of a gamut of reusability onto which every engine falls. Figure 1.1 takes a stab at the locations of some well-known games/engines along this gamut.

One would think that a game engine could be something akin to Apple QuickTime or Microsoft Windows Media Player—a general-purpose piece of software capable of playing virtually *any* game content imaginable. However, this ideal has not yet been achieved (and may never be). Most game engines are carefully crafted and fine-tuned to run a particular game on a particular hardware platform. And even the most general-purpose multiplatform engines are really only suitable for building games in one particular genre, such as first-person shooters or racing games. It’s safe to say that the more general-purpose a game engine or middleware component is, the less optimal it is for running a particular game on a particular platform.

This phenomenon occurs because designing any efficient piece of software invariably entails making trade-offs, and those trade-offs are based on assumptions about how the software will be used and/or about the target

它将在其上运行的硬件上进行渲染。例如，一个设计用于处理私密室内环境的渲染引擎可能不太擅长渲染广阔的室外环境。室内引擎可能会使用二叉空间分区 (BSP) 树或门户系统，以确保绘制的几何图形不会被墙壁或靠近相机的物体遮挡。另一方面，室外引擎可能会使用不太精确的遮挡机制，或者根本不使用遮挡机制，但它可能会积极使用细节层次 (LOD) 技术，以确保使用最少数量的三角形渲染远处的物体，同时使用高分辨率三角形网格渲染靠近相机的几何图形。

随着速度越来越快的计算机硬件和专用显卡的出现，以及更高效的渲染算法和数据结构的出现，不同类型的图形引擎之间的差异正在逐渐缩小。例如，现在可以使用第一人称射击游戏引擎来构建策略游戏。然而，通用性和最优性之间的权衡仍然存在。通过根据特定游戏和/或硬件平台的具体要求和限制对引擎进行微调，总能使游戏更加令人印象深刻。

1.4 不同游戏类型的引擎差异

游戏引擎通常在某种程度上特定于游戏类型。为拳击场上的双人格斗游戏设计的引擎与大型多人在线游戏 (MMOG) 引擎、第一人称射击游戏 (FPS) 引擎或实时战略游戏 (RTS) 引擎截然不同。然而，它们之间也存在很大的重叠——所有 3D 游戏，无论类型如何，都需要某种形式的低级用户输入（例如游戏手柄、键盘和/或鼠标）、某种形式的 3D 网格渲染、某种形式的平视显示器 (HUD)（包括以各种字体渲染文本）、强大的音频系统等等。例如，虽然虚幻引擎是为第一人称射击游戏设计的，但它也已成功用于构建许多其他类型的游戏，包括 Epic Games 开发的广受欢迎的第三人称射击游戏《战争机器》、Rocksteady Studios 开发的热门动作冒险游戏《蝙蝠侠：阿卡姆》系列、万代南梦宫工作室开发的著名格斗游戏《铁拳 7》以及 BioWare 开发的《质量效应》系列的前三款角色扮演第三人称射击游戏。

让我们来看看一些最常见的游戏类型，并探索每种游戏特定的技术要求的一些示例。

hardware on which it will run. For example, a rendering engine that was designed to handle intimate indoor environments probably won't be very good at rendering vast outdoor environments. The indoor engine might use a binary space partitioning (BSP) tree or portal system to ensure that no geometry is drawn that is being occluded by walls or objects that are closer to the camera. The outdoor engine, on the other hand, might use a less-exact occlusion mechanism, or none at all, but it probably makes aggressive use of level-of-detail (LOD) techniques to ensure that distant objects are rendered with a minimum number of triangles, while using high-resolution triangle meshes for geometry that is close to the camera.

The advent of ever-faster computer hardware and specialized graphics cards, along with ever-more-efficient rendering algorithms and data structures, is beginning to soften the differences between the graphics engines of different genres. It is now possible to use a first-person shooter engine to build a strategy game, for example. However, the trade-off between generality and optimality still exists. A game can always be made more impressive by fine-tuning the engine to the specific requirements and constraints of a particular game and/or hardware platform.

1.4 Engine Differences across Genres

Game engines are typically somewhat genre specific. An engine designed for a two-person fighting game in a boxing ring will be very different from a massively multiplayer online game (MMOG) engine or a first-person shooter (FPS) engine or a real-time strategy (RTS) engine. However, there is also a great deal of overlap—all 3D games, regardless of genre, require some form of low-level user input from the joypad, keyboard and/or mouse, some form of 3D mesh rendering, some form of heads-up display (HUD) including text rendering in a variety of fonts, a powerful audio system, and the list goes on. So while the Unreal Engine, for example, was designed for first-person shooter games, it has been used successfully to construct games in a number of other genres as well, including the wildly popular third-person shooter franchise *Gears of War* by Epic Games, the hit action-adventure games in the *Batman: Arkham* series by Rocksteady Studios, the well-known fighting game *Tekken 7* by Bandai Namco Studios, and the first three role-playing third-person shooter games in the *Mass Effect* series by BioWare.

Let's take a look at some of the most common game genres and explore some examples of the technology requirements particular to each.

图 1.2. 暴雪娱乐《守望先锋》（Xbox One、PlayStation 4、Windows）。（见颜色图 I.）

1.4.1 第一人称射击游戏（FPS）

第一人称射击 (FPS) 类型的典型代表是《雷神之锤》、《虚幻竞技场》、《半条命》、《战地》、《命运》、《泰坦陨落》和《守望先锋》（见图 1.2）。这些游戏历来都涉及在庞大但主要以走廊为主的世界中以相对缓慢的速度徒步漫游。然而，现代第一人称射击游戏可以在各种各样的虚拟环境中进行，包括广阔的开放室外区域和狭窄的室内区域。现代 FPS 的移动机制可以包括步行移动、在轨道上或自由漫游的地面车辆、气垫船、船只和飞机。有关此类型的概述，请参阅 http://en.wikipedia.org/wiki/First-person_shooter。

第一人称游戏通常是技术难度最高的游戏之一，其复杂程度可能只有第三人称射击游戏、动作平台游戏和大型多人在线游戏才能与之匹敌。这是因为第一人称射击游戏旨在让玩家沉浸在一个细节丰富、超现实的世界中。游戏行业的许多重大技术创新都源于此类游戏，这并不奇怪。

第一人称射击游戏通常注重以下技术：

- 高效渲染大型 3D 虚拟世界；

Figure 1.2. *Overwatch* by Blizzard Entertainment (Xbox One, PlayStation 4, Windows). (See Color Plate I.)

1.4.1 First-Person Shooters (FPS)

The first-person shooter (FPS) genre is typified by games like *Quake*, *Unreal Tournament*, *Half-Life*, *Battlefield*, *Destiny*, *Titanfall* and *Overwatch* (see Figure 1.2). These games have historically involved relatively slow on-foot roaming of a potentially large but primarily corridor-based world. However, modern first-person shooters can take place in a wide variety of virtual environments including vast open outdoor areas and confined indoor areas. Modern FPS traversal mechanics can include on-foot locomotion, rail-confined or free-roaming ground vehicles, hovercraft, boats and aircraft. For an overview of this genre, see http://en.wikipedia.org/wiki/First-person_shooter.

First-person games are typically some of the most technologically challenging to build, probably rivaled in complexity only by third-person shooters, action-platformer games, and massively multiplayer games. This is because first-person shooters aim to provide their players with the illusion of being immersed in a detailed, hyperrealistic world. It is not surprising that many of the game industry's big technological innovations arose out of the games in this genre.

First-person shooters typically focus on technologies such as:

- efficient rendering of large 3D virtual worlds;

- 灵敏的摄像头控制/瞄准机制；
- 玩家虚拟手臂和武器的高保真动画；
- 各种强大的手持武器；
- 宽容的玩家角色运动和碰撞模型，这常常给这些游戏一种“漂浮”的感觉；
- 非玩家角色（NPC）——玩家的敌人和盟友——的高保真动画和人工智能；
- 小规模在线多人游戏功能（通常支持 10 到 100 名玩家同时游戏），以及无处不在的“死亡竞赛”游戏模式。

第一人称射击游戏所采用的渲染技术几乎总是经过高度优化，并根据特定类型的渲染环境进行精心调整。例如，室内“地下城探索”游戏通常采用二进制空间分区树或基于传送门的渲染系统。户外 FPS 游戏则使用其他类型的渲染优化，例如遮挡剔除，或对游戏世界进行离线分区，手动或自动指定从每个源扇区可见的目标扇区。

当然，要让玩家沉浸在超现实的游戏世界中，需要的不仅仅是优化的高质量图形技术。角色动画、音频和音乐、刚体物理、游戏内过场动画以及众多其他技术，对于第一人称射击游戏来说都必须是尖端的。因此，这类游戏对技术的要求在业内堪称最严格、最广泛。

1.4.2 平台游戏和其他第三人称游戏

“平台游戏”是指以角色为主导的第三人称动作游戏，其主要玩法机制是从一个平台跳到另一个平台。2D 时代的典型游戏包括《太空恐慌》、《大金刚》、《陷阱！》和《超级马里奥兄弟》。3D 时代的平台游戏包括《超级马里奥 64》、《古惑狼》、《雷曼 2》、《刺猬索尼克》、《杰克与达斯特》系列（图 1.3）、《瑞奇与叮当》系列和《超级马里奥银河》。详情请访问 http://en.wikipedia.org/wiki/Platformer 对这一类型进行了深入讨论。

从技术要求来看，平台游戏通常可以与第三人称射击游戏和第三人称动作/冒险游戏归为一类，例如《正当防卫 2》、《战争机器 4》（图 1.4）、《神秘海域》系列、《生化危机》系列、《最后生还者》系列、《荒野大镖客 2》等等。

- a responsive camera control/aiming mechanic;
- high-fidelity animations of the player’s virtual arms and weapons;
- a wide range of powerful handheld weaponry;
- a forgiving player character motion and collision model, which often gives these games a “floaty” feel;
- high-fidelity animations and artificial intelligence for the non-player characters (NPCs)—the player’s enemies and allies; and
- small-scale online multiplayer capabilities (typically supporting between 10 and 100 simultaneous players), and the ubiquitous “death match” gameplay mode.

The rendering technology employed by first-person shooters is almost always highly optimized and carefully tuned to the particular type of environment being rendered. For example, indoor “dungeon crawl” games often employ binary space partitioning trees or portal-based rendering systems. Outdoor FPS games use other kinds of rendering optimizations such as occlusion culling, or an offline sectorization of the game world with manual or automated specification of which target sectors are visible from each source sector.

Of course, immersing a player in a hyperrealistic game world requires much more than just optimized high-quality graphics technology. The character animations, audio and music, rigid body physics, in-game cinematics and myriad other technologies must all be cutting-edge in a first-person shooter. So this genre has some of the most stringent and broad technology requirements in the industry.

1.4.2 Platformers and Other Third-Person Games

“Platformer” is the term applied to third-person character-based action games where jumping from platform to platform is the primary gameplay mechanic. Typical games from the 2D era include *Space Panic*, *Donkey Kong*, *Pitfall!* and *Super Mario Brothers*. The 3D era includes platformers like *Super Mario 64*, *Crash Bandicoot*, *Rayman 2*, *Sonic the Hedgehog*, the *Jak and Daxter* series (Figure 1.3), the *Ratchet & Clank* series and *Super Mario Galaxy*. See http://en.wikipedia.org/wiki/Platformer for an in-depth discussion of this genre.

In terms of their technological requirements, platformers can usually be lumped together with third-person shooters and third-person action/adventure games like *Just Cause 2*, *Gears of War 4* (Figure 1.4), the *Uncharted* series, the *Resident Evil* series, the *The Last of Us* series, *Red Dead Redemption 2*, and the list goes on.

第三人称角色扮演游戏与第一人称射击游戏有很多共同之处，但第三人称角色扮演游戏更注重主角的能力和移动方式。此外，第三人称角色扮演游戏要求玩家角色拥有高保真度的全身角色动画，这与典型的第一人称射击游戏对“悬浮手臂”动画的要求相对低一些。需要注意的是，几乎所有第一人称射击游戏都包含在线多人游戏组件，因此除了第一人称手臂动画外，还必须渲染玩家角色的全身动画。然而，这些第一人称射击游戏玩家角色的保真度通常无法与同类游戏中非玩家角色的保真度相比，也无法与第三人称游戏中玩家角色的保真度相比。

在平台游戏中，主角通常采用卡通风格，并非特别逼真或高分辨率。然而，第三人称射击游戏通常采用高度逼真的人形玩家角色。在这两种情况下，玩家角色通常都拥有非常丰富的动作和动画。

此类游戏特别关注的一些技术包括：

图 1.3. 顽皮狗出品的《杰克二世》（《杰克、达斯特》、《杰克与达斯特》和《杰克二世》© 2003、2013/TM SIE。由顽皮狗创作和开发，PlayStation 2。）（参见彩色图版 II。）

Third-person character-based games have a lot in common with first-person shooters, but a great deal more emphasis is placed on the main character's abilities and locomotion modes. In addition, high-fidelity full-body character animations are required for the player's avatar, as opposed to the somewhat less-taxing animation requirements of the "floating arms" in a typical FPS game. It's important to note here that almost all first-person shooters have an online multiplayer component, so a full-body player avatar must be rendered in addition to the first-person arms. However, the fidelity of these FPS player avatars is usually not comparable to the fidelity of the non-player characters in these same games; nor can it be compared to the fidelity of the player avatar in a third-person game.

In a platformer, the main character is often cartoon-like and not particularly realistic or high-resolution. However, third-person shooters often feature a highly realistic humanoid player character. In both cases, the player character typically has a very rich set of actions and animations.

Some of the technologies specifically focused on by games in this genre include:

Figure 1.3. *Jak II* by Naughty Dog (Jak, Daxter, Jak and Daxter, and Jak II © 2003, 2013/™ SIE. Created and developed by Naughty Dog, PlayStation 2.) (See Color Plate II.)

图 1.4. The Coalition 出品的《战争机器 4》（Xbox One）。（参见彩色图版 III。）

• 移动平台、梯子、绳索、棚架和其他有趣的运动模式；

• 类似谜题的环境元素；

第三人称“跟随摄像机”，其焦点始终集中在玩家角色上，其旋转通常由人类玩家通过右操纵杆（在控制台上）或鼠标（在 PC 上 - 请注意，虽然 PC 上有许多流行的第三人称射击游戏，但平台游戏类型几乎只存在于控制台上）控制；

• 复杂的摄像机碰撞系统，确保视点永远不会“剪切”背景几何或动态前景物体。

1.4.3 格斗游戏

格斗游戏通常是双人游戏，其中人形角色在某种形式的擂台上互相攻击。这类游戏的典型代表是《灵魂能力》和《铁拳3》（见图1.5）。维基百科页面http://en.wikipedia.org/wiki/Fighting_game概述了这类游戏。

传统上，格斗类游戏的技术重点集中在：

Figure 1.4. *Gears of War 4* by The Coalition (Xbox One). (See Color Plate III.)

- moving platforms, ladders, ropes, trellises and other interesting locomotion modes;
- puzzle-like environmental elements;
- a third-person “follow camera” which stays focused on the player character and whose rotation is typically controlled by the human player via the right joypad stick (on a console) or the mouse (on a PC—note that while there are a number of popular third-person shooters on a PC, the platformer genre exists almost exclusively on consoles); and
- a complex camera collision system for ensuring that the view point never “clips” through background geometry or dynamic foreground objects.

1.4.3 Fighting Games

Fighting games are typically two-player games involving humanoid characters pummeling each other in a ring of some sort. The genre is typified by games like *Soul Calibur* and *Tekken 3* (see Figure 1.5). The Wikipedia page http://en.wikipedia.org/wiki/Fighting_game provides an overview of this genre.

Traditionally games in the fighting genre have focused their technology efforts on:

- 丰富的战斗动画；
- 准确的命中检测；
- 能够检测复杂按钮和操纵杆组合的用户输入系统；以及
- 人群，但背景相对静止。

由于这些游戏中的3D世界很小，并且摄像机始终以动作为中心，因此从历史上看，这些游戏很少或根本不需要世界细分或遮挡剔除。同样，它们也不会采用先进的三维音频传播模型。

现代格斗游戏，如 EA 的《Fight Night Round 4》和 NetherRealm Studios 的《Injustice 2》（图 1.6），已经通过以下功能提高了技术水平：

- 高清人物图形；
- 具有地下散射和汗液效果的逼真皮肤着色器；
- 逼真的灯光和粒子效果；
- 高保真角色动画；以及

图 1.5. Namco 出品的《铁拳 3》（PlayStation）。（参见彩色图 IV。）

- a rich set of fighting animations;
- accurate hit detection;
- a user input system capable of detecting complex button and joystick combinations; and
- crowds, but otherwise relatively static backgrounds.

Since the 3D world in these games is small and the camera is centered on the action at all times, historically these games have had little or no need for world subdivision or occlusion culling. They would likewise not be expected to employ advanced three-dimensional audio propagation models, for example.

Modern fighting games like EA's *Fight Night Round 4* and NetherRealm Studios' *Injustice 2* (Figure 1.6) have upped the technological ante with features like:

- high-definition character graphics;
- realistic skin shaders with subsurface scattering and sweat effects;
- photo-realistic lighting and particle effects;
- high-fidelity character animations; and

Figure 1.5. *Tekken 3* by Namco (PlayStation). (See Color Plate IV.)

• 基于物理的角色布料和头发模拟。

值得注意的是，一些格斗游戏，例如 Ninja Theory 的《天剑》和育碧蒙特利尔工作室的《荣耀战魂》，其背景设定在一个大规模的虚拟世界，而非一个封闭的竞技场。事实上，许多人认为这是一种独立的游戏类型，有时被称为“格斗游戏”。这类格斗游戏的技术要求可能更类似于第三人称射击游戏或策略游戏。

1.4.4 赛车游戏

赛车游戏涵盖所有以在赛道上驾驶汽车或其他车辆为主要任务的游戏。该类型有许多子类别。专注于模拟的赛车游戏（“模拟游戏”）旨在提供尽可能逼真的驾驶体验（例如，Gran Turismo）。街机赛车手更喜欢过度的乐趣而不是现实主义（例如，San Francisco Rush、Cruis'n USA、Hydro Thunder）。其中一个子类型使用经过改装的消费车辆探索街头赛车的亚文化（例如，Need for Speed、Juiced）。卡丁车赛车是一个子类别，其中平台游戏中的流行角色或电视中的卡通人物被重新塑造成古怪车辆的驾驶员（例如，Mario Kart、Jak X、Freaky Flyers）。赛车游戏并不总是需要涉及基于时间的竞争。例如，一些卡丁车赛车游戏提供

图 1.6. NetherRealm Studios 出品的《不义联盟 2》（PlayStation 4、Xbox One、安卓、iOS、Microsoft Windows）。（参见彩色图版 V。）

- physics-based cloth and hair simulations for the characters.

It's important to note that some fighting games like Ninja Theory's *Heavenly Sword* and *For Honor* by Ubisoft Montreal take place in a large-scale virtual world, not a confined arena. In fact, many people consider this to be a separate genre, sometimes called a *brawler*. This kind of fighting game can have technical requirements more akin to those of a third-person shooter or a strategy game.

1.4.4 Racing Games

The racing genre encompasses all games whose primary task is driving a car or other vehicle on some kind of track. The genre has many subcategories. Simulation-focused racing games ("sims") aim to provide a driving experience that is as realistic as possible (e.g., *Gran Turismo*). Arcade racers favor over-the-top fun over realism (e.g., *San Francisco Rush*, *Cruis'n USA*, *Hydro Thunder*). One subgenre explores the subculture of street racing with tricked out consumer vehicles (e.g., *Need for Speed*, *Juiced*). Kart racing is a subcategory in which popular characters from platformer games or cartoon characters from TV are re-cast as the drivers of whacky vehicles (e.g., *Mario Kart*, *Jak X*, *Freaky Flyers*). Racing games need not always involve time-based competition. Some kart racing games, for example, offer

Figure 1.6. *Injustice 2* by NetherRealm Studios (PlayStation 4, Xbox One, Android, iOS, Microsoft Windows). (See Color Plate V.)

图 1.7. Polyphony Digital 出品的《Gran Turismo Sport》（PlayStation 4）。（参见彩色图版 VI。）

玩家可以在其中互相射击、收集战利品或参与各种其他限时或不限时的任务。有关此类游戏的讨论，请参阅 http://en.wikipedia.org/wiki/Racing_game。

赛车游戏通常非常线性，很像老式的 FPS 游戏。然而，行驶速度通常比 FPS 快得多。因此，更多的焦点放在非常长的基于走廊的赛道或环形赛道上，有时还会有各种替代路线和秘密捷径。赛车游戏通常将所有图形细节都集中在车辆、赛道和周围环境上。作为一个例子，图 1.7 展示了著名的 Gran Turismo 赛车游戏系列的最新一期 Gran Turismo Sport 的屏幕截图，该游戏由 Polyphony Digital 开发，由 Sony Interactive Entertainment 发行。然而，卡丁车赛车手也将大量的渲染和动画带宽投入到驾驶车辆的角色上。

典型赛车游戏的一些技术特性包括以下技术：

- 渲染远处的背景元素时会使用各种“技巧”，例如使用二维卡片来表示树木、丘陵和山脉。
- 赛道通常被分解成相对简单的二维区域，称为“扇区”。这些数据结构用于优化渲染和可见性确定，辅助人工智能和非人为控制车辆的路径查找，以及解决许多其他技术问题。
- 摄像机通常跟随车辆后方，以便第三人称视角

Figure 1.7. *Gran Turismo Sport* by Polyphony Digital (PlayStation 4). (See Color Plate VI.)

modes in which players shoot at one another, collect loot or engage in a variety of other timed and untimed tasks. For a discussion of this genre, see http://en.wikipedia.org/wiki/Racing_game.

A racing game is often very linear, much like older FPS games. However, travel speed is generally much faster than in an FPS. Therefore, more focus is placed on very long corridor-based tracks, or looped tracks, sometimes with various alternate routes and secret short-cuts. Racing games usually focus all their graphic detail on the vehicles, track and immediate surroundings. As an example of this, Figure 1.7 shows a screenshot from the latest installment in the well-known *Gran Turismo* racing game series, *Gran Turismo Sport*, developed by Polyphony Digital and published by Sony Interactive Entertainment. However, kart racers also devote significant rendering and animation bandwidth to the characters driving the vehicles.

Some of the technological properties of a typical racing game include the following techniques:

- Various “tricks” are used when rendering distant background elements, such as employing two-dimensional cards for trees, hills and mountains.
- The track is often broken down into relatively simple two-dimensional regions called “sectors.” These data structures are used to optimize rendering and visibility determination, to aid in artificial intelligence and path finding for non-human-controlled vehicles, and to solve many other technical problems.
- The camera typically follows behind the vehicle for a third-person per-

图 1.8。Ensemble Studios 出品的《帝国时代》（Windows 系统）。（参见彩色图 VII。）

视角，或者有时位于驾驶舱内的第一人称视角。

- 当赛道涉及隧道和其他“狭窄”空间时，通常需要付出很大努力来确保摄像机不会与背景几何形状发生碰撞。

1.4.5 策略游戏

现代策略游戏类型可以说是由《沙丘 II：王朝的建立》（1992 年）定义的。该类型的其他游戏包括《魔兽争霸》、《命令与征服》、《帝国时代》和《星际争霸》。在该类型中，玩家在广阔的战场上战略性地部署其武器库中的战斗单位，试图压倒对手。游戏世界通常以倾斜的自上而下的视角呈现。回合制策略游戏和实时策略 (RTS) 之间通常会有所区别。有关此类型的讨论，请参阅 https://en.wikipedia.org/wiki/Strategy_video_game。

策略游戏玩家通常无法为了远距离观察而大幅改变视角。这种限制使得开发者能够在策略游戏的渲染引擎中运用各种优化。

Figure 1.8. *Age of Empires* by Ensemble Studios (Windows). (See Color Plate VII.)

spective, or is sometimes situated inside the cockpit first-person style.

- When the track involves tunnels and other “tight” spaces, a good deal of effort is often put into ensuring that the camera does not collide with background geometry.

1.4.5 Strategy Games

The modern strategy game genre was arguably defined by *Dune II: The Building of a Dynasty* (1992). Other games in this genre include *Warcraft*, *Command & Conquer*, *Age of Empires* and *Starcraft*. In this genre, the player deploys the battle units in his or her arsenal strategically across a large playing field in an attempt to overwhelm his or her opponent. The game world is typically displayed at an oblique top-down viewing angle. A distinction is often made between turn-based strategy games and real-time strategy (RTS). For a discussion of this genre, see https://en.wikipedia.org/wiki/Strategy_video_game.

The strategy game player is usually prevented from significantly changing the viewing angle in order to see across large distances. This restriction permits developers to employ various optimizations in the rendering engine of a strategy game.

图 1.9. Creative Assembly 出品的《全面战争：战锤 2》（Windows 系统）。（参见彩色图版 VIII。）

该类型的早期游戏采用基于网格（基于单元）的世界构建，并使用正交投影来大大简化渲染器。例如，图 1.8 展示了经典策略游戏《帝国时代》的屏幕截图。

现代策略游戏有时会使用透视投影和真实的 3D 世界，但它们仍可能采用网格布局系统来确保单位和背景元素（例如建筑物）彼此正确对齐。

一个流行的例子是《全面战争：战锤 2》，如图 1.9 所示。

策略游戏中其他一些常见做法包括以下技巧：

- 每个单位的分辨率相对较低，因此游戏可以同时在屏幕上支持大量单位。
- 高度场地形通常是游戏设计和玩游戏的画布。
- 除了部署自己的部队之外，玩家通常还可以在地形上建造新建筑。
- 用户交互通常是通过单击和基于区域的单位选择，以及包含命令、设备、单位类型、建筑类型等的菜单或工具栏。

Figure 1.9. *Total War: Warhammer 2* by Creative Assembly (Windows). (See Color Plate VIII.)

Older games in the genre employed a grid-based (cell-based) world construction, and an orthographic projection was used to greatly simplify the renderer. For example, Figure 1.8 shows a screenshot from the classic strategy game *Age of Empires*.

Modern strategy games sometimes use perspective projection and a true 3D world, but they may still employ a grid layout system to ensure that units and background elements, such as buildings, align with one another properly. A popular example, *Total War: Warhammer 2*, is shown in Figure 1.9.

Some other common practices in strategy games include the following techniques:

- Each unit is relatively low-res, so that the game can support large numbers of them on-screen at once.
- Height-field terrain is usually the canvas upon which the game is designed and played.
- The player is often allowed to build new structures on the terrain in addition to deploying his or her forces.
- User interaction is typically via single-click and area-based selection of units, plus menus or toolbars containing commands, equipment, unit types, building types, etc.

图 1.10. 暴雪娱乐的《魔兽世界》（Windows、MacOS）。（参见彩色图 IX。）

1.4.6 大型多人在线游戏（MMOG）

大型多人在线游戏（MMOG 或简称 MMO）类型的典型代表包括《激战 2》（AreaNet/NCsoft）、《无尽的任务》（989 Studios/SOE）、《魔兽世界》（暴雪）和《星球大战：星系》（SOE/Lucas Arts）等。MMO 是指任何支持大量玩家（数千至数十万）同时在线的游戏，通常所有玩家都在一个非常庞大且持久的虚拟世界中游戏（即，一个内部状态会持续很长时间，远远超过任何一个玩家的游戏时间）。除此之外，MMO 的游戏体验通常与小型多人游戏相似。该类型的子类别包括大型多人在线角色扮演游戏 (MMORPG)、大型多人在线实时战略游戏 (MMORTS) 和大型多人在线第一人称射击游戏 (MMOFPS)。有关该类型的讨论，请参阅 http://en.wikipedia.org/wiki/MMOG。图 1.10 展示了广受欢迎的 MMORPG《魔兽世界》的屏幕截图。

所有 MMOG 的核心都是一组非常强大的服务器。这些服务器维护着游戏世界的权威状态，管理用户的登录和退出，提供用户间聊天或 IP 语音 (VoIP) 服务等等。几乎所有 MMOG 都要求用户定期支付某种费用

Figure 1.10. *World of Warcraft* by Blizzard Entertainment (Windows, MacOS). (See Color Plate IX.)

1.4.6 Massively Multiplayer Online Games (MMOG)

The massively multiplayer online game (MMOG or just MMO) genre is typified by games like *Guild Wars 2* (AreaNet/NCsoft), *EverQuest* (989 Studios/SOE), *World of Warcraft* (Blizzard) and *Star Wars Galaxies* (SOE/Lucas Arts), to name a few. An MMO is defined as any game that supports huge numbers of simultaneous players (from thousands to hundreds of thousands), usually all playing in one very large, *persistent* virtual world (i.e., a world whose internal state persists for very long periods of time, far beyond that of any one player's gameplay session). Otherwise, the gameplay experience of an MMO is often similar to that of their small-scale multiplayer counterparts. Subcategories of this genre include MMO role-playing games (MMORPG), MMO real-time strategy games (MMORTS) and MMO first-person shooters (MMOFPS). For a discussion of this genre, see http://en.wikipedia.org/wiki/MMOG. Figure 1.10 shows a screenshot from the hugely popular MMORPG *World of Warcraft*.

At the heart of all MMOGs is a very powerful battery of servers. These servers maintain the authoritative state of the game world, manage users signing in and out of the game, provide inter-user chat or voice-over-IP (VoIP) services and more. Almost all MMOGs require users to pay some kind of regular

游戏开发者需要支付订阅费才能玩游戏，他们还可能在游戏世界内或游戏外提供微交易。因此，中央服务器最重要的作用或许是处理计费和微交易，而这正是游戏开发者的主要收入来源。

由于大型多人在线游戏 (MMO) 的世界规模巨大且支持的用户数量极其庞大，因此这类游戏的图形保真度几乎总是低于非大型多人在线游戏。

图 1.11 展示了 Bungie 最新 FPS 游戏《命运 2》的屏幕截图。这款游戏被称为 MMOFPS，因为它融合了 MMO 类型的一些元素。然而，Bungie 更喜欢称之为“共享世界”游戏，因为与传统的 MMO（玩家可以看到并与特定服务器上的任何其他玩家互动）不同，《命运》提供“即时匹配”功能。这允许玩家只与服务器已匹配的其他玩家互动；这个匹配系统在《命运 2》中得到了显著改进。此外，与传统的 MMO 不同的是，《命运 2》的图形保真度与第一人称和第三人称射击游戏相当。

这里需要指出的是，《绝地求生》（PUBG）这款游戏最近催生出了一个叫做“大逃杀”的子游戏类型。这类游戏模糊了普通多人射击游戏和大型多人在线游戏之间的界限，因为它们通常让大约100名玩家在一个在线世界中相互对抗，采用基于生存的“最后一人存活”的游戏风格。

图 1.11。Bungie 出品的《命运 2》，© 2018 Bungie Inc.（Xbox One、PlayStation 4、PC）（参见彩色图版 X。）

subscription fee in order to play, and they may offer micro-transactions within the game world or out-of-game as well. Hence, perhaps the most important role of the central server is to handle the billing and micro-transactions which serve as the game developer's primary source of revenue.

Graphics fidelity in an MMO is almost always lower than its non-massively multiplayer counterparts, as a result of the huge world sizes and extremely large numbers of users supported by these kinds of games.

Figure 1.11 shows a screen from Bungie's latest FPS game, *Destiny 2*. This game has been called an MMOFPS because it incorporates some aspects of the MMO genre. However, Bungie prefers to call it a "shared world" game because unlike a traditional MMO, in which a player can see and interact with literally any other player on a particular server, *Destiny* provides "on-the-fly match-making." This permits the player to interact only with the other players with whom they have been matched by the server; this matchmaking system has been significantly improved for *Destiny 2*. Also unlike a traditional MMO, the graphics fidelity in *Destiny 2* is on par with first- and third-person shooters.

We should note here that the game *Player Unknown's Battlegrounds* (PUBG) has recently popularized a subgenre known as *battle royale*. This type of game blurs the line between regular multiplayer shooters and massively multiplayer online games, because they typically pit on the order of 100 players against each other in an online world, employing a survival-based "last man standing" gameplay style.

Figure 1.11. *Destiny 2* by Bungie, © 2018 Bungie Inc. (Xbox One, PlayStation 4, PC) (See Color Plate X.)

1.4.7 玩家创作的内容

随着社交媒体的兴起，游戏的协作性越来越强。游戏设计的最新趋势是玩家创作内容。例如，Media Molecule 的《小小大星球》（LittleBigPlanet™）、《小小大星球 2》（LittleBigPlanet™ 2）（图 1.12）和《小小大星球 3：回家之旅》（LittleBigPlanet™ 3: The Journey Home）从技术上来说属于益智平台游戏，但它们最显著和独特的特点是鼓励玩家创造、发布和分享自己的游戏世界。Media Molecule 在这一引人入胜的游戏类型中的最新作品是 PlayStation 4 版《梦想》（Dreams）（图 1.13）。

如今，在玩家创作内容类型中，最受欢迎的游戏或许是《我的世界》（Minecraft，图 1.14）。这款游戏的精彩之处在于它的简洁性：《我的世界》游戏世界由简单的立方体素状元素构成，这些元素映射着低分辨率纹理，以模拟各种材质。方块可以是实心的，也可以包含火把、铁砧、告示牌、栅栏和玻璃板等物品。游戏世界中居住着一个或多个玩家角色、鸡和猪等动物，以及各种“怪物”——好人（例如村民）和坏人（例如僵尸和无处不在的爬行者），它们会偷偷靠近毫无戒心的玩家并爆炸（在用引信燃烧的“嘶嘶声”警告玩家后不久）。

玩家可以在《我的世界》中创建一个随机世界，然后在生成的地形中挖掘隧道和洞穴。他们还可以建造自己的建筑，从简单的地形和植被到广阔复杂的建筑，应有尽有。

图 1.12。Media Molecule 出品的《小小大星球 2》，© 2014 Sony Interactive Entertainment（PlayStation 3）。（参见彩色图 XI。）

1.4.7 Player-Authored Content

As social media takes off, games are becoming more and more collaborative in nature. A recent trend in game design is toward *player-authored content*. For example, Media Molecule's *LittleBigPlanet*,™ *LittleBigPlanet*™ 2 (Figure 1.12) and *LittleBigPlanet*™ *3: The Journey Home* are technically *puzzle platformers*, but their most notable and unique feature is that they encourage players to create, publish and share their own game worlds. Media Molecule's latest installment in this engaging genre is *Dreams* for the PlayStation 4 (Figure 1.13).

Perhaps the most popular game today in the player-created content genre is *Minecraft* (Figure 1.14). The brilliance of this game lies in its simplicity: *Minecraft* game worlds are constructed from simple cubic voxel-like elements mapped with low-resolution textures to mimic various materials. Blocks can be solid, or they can contain items such as torches, anvils, signs, fences and panes of glass. The game world is populated with one or more player characters, animals such as chickens and pigs, and various "mobs"—good guys like villagers and bad guys like zombies and the ubiquitous *creepers* who sneak up on unsuspecting players and explode (only scant moments after warning the player with the "hiss" of a burning fuse).

Players can create a randomized world in *Minecraft* and then dig into the generated terrain to create tunnels and caverns. They can also construct their own structures, ranging from simple terrain and foliage to vast and complex

Figure 1.12. *LittleBigPlanet*™ *2* by Media Molecule, © 2014 Sony Interactive Entertainment (PlayStation 3). (See Color Plate XI.)

图 1.13。Media Molecule 作品《Dreams》，© 2017 Sony Computer Computer Europe（PlayStation 4）。（参见彩色图版 XII。）

建筑和机械。或许，Minecraft 中最巧妙的创意莫过于红石。这种材料充当“线路”，让玩家能够铺设电路，控制活塞、漏斗、矿车和其他游戏中的动态元素。因此，玩家几乎可以创造任何他们能想象到的东西，然后通过搭建服务器邀请好友在线畅玩，与他们分享他们的世界。

图 1.14. Markus “Notch” Persson / Mojang AB 出品的 Minecraft（Windows、MacOS、Xbox 360、PlayStation 3、PlayStation Vita、iOS）。（参见彩色图版 XIII。）

Figure 1.13. *Dreams* by Media Molecule, © 2017 Sony Computer Computer Europe (PlayStation 4). (See Color Plate XII.)

buildings and machinery. Perhaps the biggest stroke of genius in *Minecraft* is *redstone*. This material serves as “wiring,” allowing players to lay down circuitry that controls pistons, hoppers, mine carts and other dynamic elements in the game. As a result, players can create virtually anything they can imagine, and then share their worlds with their friends by hosting a server and inviting them to play online.

Figure 1.14. *Minecraft* by Markus “Notch” Persson / Mojang AB (Windows, MacOS, Xbox 360, PlayStation 3, PlayStation Vita, iOS). (See Color Plate XIII.)

1.4.8 虚拟、增强和混合现实

虚拟现实、增强现实和混合现实是令人兴奋的新技术，旨在让观看者沉浸在完全由计算机生成或由计算机生成的图像增强的3D世界中。这些技术在游戏行业之外有着广泛的应用，但它们也已成为各种游戏内容的可行平台。

1.4.8.1 虚拟现实

虚拟现实 (VR) 可以定义为一种沉浸式多媒体或计算机模拟现实，它模拟用户在现实世界或虚拟世界中的身临其境。计算机生成虚拟现实 (CG VR) 是这项技术的一个子集，其中虚拟世界完全由计算机图形生成。用户通过佩戴头戴式设备（例如 HTC Vive、Oculus Rift、索尼 PlayStation VR、三星 Gear VR 或 Google Daydream View）来观看虚拟环境。头戴式设备将内容直接显示在用户眼前；系统还会跟踪头戴式设备在现实世界中的运动，从而使虚拟摄像头的运动与佩戴者的运动完美匹配。用户通常手持设备，系统可以跟踪每只手的运动。这使得用户可以在虚拟世界中进行交互：

例如，可以推、捡起或扔物体。

1.4.8.2 增强现实和混合现实

增强现实 (AR) 和混合现实 (MR) 这两个术语经常被混淆或互换使用。这两种技术都向用户呈现现实世界的视图，但都使用了计算机图形来增强体验。在这两种技术中，智能手机、平板电脑或增强现实眼镜等观看设备都会显示现实世界场景的实时或静态视图，并将计算机图形叠加在该图像上。在实时 AR 和 MR 系统中，观看设备中的加速度计允许虚拟摄像头的运动跟踪设备的运动，从而产生一种错觉，即设备只是一个供我们观察现实世界的窗口，从而使叠加的计算机图形具有很强的真实感。

有些人为了区分这两种技术，用“增强现实”来描述将计算机图形叠加在实时、直接或间接的现实世界视图上，但又不将其固定在现实世界的技术。而“混合现实”则更常被这样称呼。

1.4.8 Virtual, Augmented and Mixed Reality

Virtual, augmented and mixed reality are exciting new technologies that aim to immerse the viewer in a 3D world that is either entirely generated by a computer, or is augmented by computer-generated imagery. These technologies have many applications outside the game industry, but they have also become viable platforms for a wide range of gaming content.

1.4.8.1 Virtual Reality

Virtual reality (VR) can be defined as an immersive multimedia or computer-simulated reality that simulates the user's presence in an environment that is either a place in the real world or in an imaginary world. Computer-generated VR (CG VR) is a subset of this technology in which the virtual world is exclusively generated via computer graphics. The user views this virtual environment by donning a headset such as HTC Vive, Oculus Rift, Sony PlayStation VR, Samsung Gear VR or Google Daydream View. The headset displays the content directly in front of the user's eyes; the system also tracks the movement of the headset in the real world, so that the virtual camera's movements can be perfectly matched to those of the person wearing the headset. The user typically holds devices in his or her hands which allow the system to track the movements of each hand. This allows the user to interact in the virtual world: Objects can be pushed, picked up or thrown, for example.

1.4.8.2 Augmented and Mixed Reality

The terms *augmented reality* (AR) and *mixed reality* (MR) are often confused or used interchangeably. Both technologies present the user with a view of the real world, but with computer graphics used to enhance the experience. In both technologies, a viewing device like a smart phone, tablet or tech-enhanced pair of glasses displays a real-time or static view of a real-world scene, and computer graphics are overlaid on top of this image. In real-time AR and MR systems, accelerometers in the viewing device permit the virtual camera's movements to track the movements of the device, producing the illusion that the device is simply a window through which we are viewing the actual world, and hence giving the overlaid computer graphics a strong sense of realism.

Some people make a distinction between these two technologies by using the term "augmented reality" to describe technologies in which computer graphics are overlaid on a live, direct or indirect view of the real world, but are not anchored to it. The term "mixed reality," on the other hand, is more often

指利用计算机图形渲染与现实世界紧密相连、看似真实存在的虚拟物体。然而，这种区别并未被普遍接受。

以下是 AR 技术实际应用的几个示例：

• 美国陆军使用一种被称为“战术增强现实”（TAR）的系统为其士兵提供增强的战术意识——它将一个类似视频游戏的平视显示器（HUD）与一个小地图和物体标记叠加到士兵对现实世界的视野上（https://youtu.be/x8p19j8C6VI）。

• 2015 年，迪士尼展示了一些很酷的 AR 技术，该技术将 3D 卡通人物渲染在一张真实纸张上，并在纸张上用蜡笔绘制出 2D 版本的角色（https://youtu.be/SWzurBQ81CM）。

• 百事公司还用一个带有增强现实技术的公交车站恶作剧，逗乐了伦敦的通勤者。坐在公交车站围栏里的人们看到了增强现实技术的图像，其中包括一只徘徊的老虎、一颗陨石坠落，以及一条外星人的触手抓住街上不知情的路人 (https://youtu.be/Go9rf9GmYpM)。

以下是 MR 的一些示例：

• 从 Android 8.1 开始，Pixel 1 和 Pixel 2 上的相机应用程序支持 AR Stickers，这是一项有趣的功能，允许用户将动画 3D 对象和角色放入视频和照片中。

• 微软的 HoloLens 是混合现实的另一个例子。它将基于世界的图形叠加到实时视频图像上，可用于教育培训、工程、医疗保健和娱乐等广泛的应用领域。

1.4.8.3 VR/AR/MR游戏

游戏行业目前正在尝试VR和AR/MR技术，并试图在这些新媒体中找到立足点。一些传统的3D游戏已被“移植”到VR，带来了非常有趣的体验，尽管并非特别具有创新性。但或许更令人兴奋的是，全新的游戏类型正在涌现，它们提供的游戏体验是VR或AR/MR无法实现的。

例如，Owlchemy Labs 的 Job Simulator 让用户进入一个由机器人运营的虚拟工作博物馆，并要求他们利用游戏机制，以玩笑的方式模拟各种现实世界的工作

applied to the use of computer graphics to render imaginary objects which are anchored to the real world and appear to exist within it. However, this distinction is by no means universally accepted.

Here are a few examples of AR technology in action:

- The U.S. Army provides its soldiers with improved tactical awareness using a system dubbed “tactical augmented reality” (TAR)—it overlays a video-game-like heads-up display (HUD) complete with a mini-map and object markers onto the soldier’s view of the real world (https://youtu.be/x8p19j8C6VI).
- In 2015, Disney demonstrated some cool AR technology that renders a 3D cartoon character on top of a sheet of real-world paper on which a 2D version of the character is colored with a crayon (https://youtu.be/SWzurBQ81CM).
- PepsiCo also pranked commuting Londoners with an AR-enabled bus stop. People sitting in the bus stop enclosure were treated to AR images of a prowling tiger, a meteor crashing, and an alien tentacle grabbing unwitting passers by off the street (https://youtu.be/Go9rf9GmYpM).

And here are a few examples of MR:

- Starting with Android 8.1, the camera app on the Pixel 1 and Pixel 2 supports *AR Stickers*, a fun feature that allows users to place animated 3D objects and characters into videos and photos.
- Microsoft’s HoloLens is another example of mixed reality. It overlays world-anchored graphics onto a live video image, and can be used for a wide range of applications including education and training, engineering, health care, and entertainment.

1.4.8.3 VR/AR/MR Games

The game industry is currently experimenting with VR and AR/MR technologies, and is trying to find its footing within these new media. Some traditional 3D games have been “ported” to VR, yielding very interesting, if not particularly innovative, experiences. But perhaps more exciting, entirely new game genres are starting to emerge, offering gameplay experiences that could not be achieved without VR or AR/MR.

For example, *Job Simulator* by Owlchemy Labs plunges the user into a virtual job museum run by robots, and asks them to perform tongue-in-cheek approximations of various real-world jobs, making use of game mechanics

这在非VR平台上根本行不通。Owlchemy的下一部作品《假期模拟器》将同样异想天开的幽默感和艺术风格运用到了《工作模拟器》的世界里，机器人会邀请玩家放松身心并执行各种任务。图1.15展示了另一款创新（且略带不安！）的HTC Vive游戏《会计》的截图，这款游戏由《瑞克和莫蒂》和《史丹利的寓言》的创作者打造。

1.4.8.4 VR游戏引擎

VR 游戏引擎在技术上与第一人称射击游戏引擎在很多方面相似，事实上，许多支持 FPS 的引擎（例如 Unity 和 Unreal Engine）都"开箱即用"地支持 VR。然而，VR 游戏与 FPS 游戏在许多重要方面有所不同：

- 立体渲染。VR 游戏需要渲染场景两次，每只眼睛渲染一次。这会使需要渲染的图形基元数量翻倍，尽管由于两眼距离较近，图形流水线的其他方面（例如可见性剔除）每帧只需执行一次。因此，VR 游戏的渲染成本并不像分屏多人模式那样高，但从两个（略微）不同的虚拟摄像机渲染每帧两次的原理是相同的。

- 帧率非常高。研究表明，VR 的帧率低于 90

图 1.15. Squanchtendo 和 Crows Crows Crows (HTC Vive) 的《会计》。（参见彩色图版 XIV。）

that simply wouldn't work on a non-VR platform. Owlchemy's next installment, *Vacation Simulator*, applies the same whimsical sense of humour and art style to a world in which the robots of Job Simulator invite the player to relax and perform various tasks. Figure 1.15 shows a screenshot from another innovative (and somewhat disturbing!) game for HTC Vive called *Accounting*, from the creators of "Rick & Morty" and *The Stanley Parable*.

1.4.8.4 VR Game Engines

VR game engines are technologically similar in many respects to first-person shooter engines, and in fact many FPS-capable engines such as Unity and Unreal Engine support VR "out of the box." However, VR games differ from FPS games in a number of significant ways:

- *Stereoscopic rendering*. A VR game needs to render the scene twice, once for each eye. This doubles the number of graphics primitives that must be rendered, although other aspects of the graphics pipeline such as visibility culling can be performed only once per frame, since the eyes are reasonably close together. As such, a VR game isn't quite as expensive to render as the same game would be to render in split-screen multiplayer mode, but the principle of rendering each frame twice from two (slightly) different virtual cameras is the same.
- *Very high frame rate*. Studies have shown that VR running at below 90

Figure 1.15. *Accounting* by Squanchtendo and Crows Crows Crows (HTC Vive). (See Color Plate XIV.)

每秒帧数过高可能会导致用户出现方向感丧失、恶心等不良反应。这意味着 VR 系统不仅需要每帧渲染两次场景，而且还需要以 90 FPS 以上的帧率进行渲染。正因如此，VR 游戏和应用程序通常需要在高性能 CPU 和 GPU 硬件上运行。

• 导航问题。在 FPS 游戏中，玩家只需使用游戏手柄或 WASD 键即可在游戏世界中自由行走。在 VR 游戏中，用户可以通过在现实世界中实际行走来实现少量移动，但安全的物理游戏区域通常很小（相当于小型浴室或衣柜的大小）。“飞行”式移动也容易引起恶心，因此大多数游戏选择点击式传送机制，将虚拟玩家/摄像机移动到更远的距离。此外，各种现实世界设备也已被设计出来，允许 VR 用户用脚“原地行走”，以便在 VR 世界中移动。

当然，VR 在某种程度上弥补了这些限制，因为它实现了传统电子游戏中无法实现的全新用户交互模式。例如，

• 用户可以在现实世界中触摸、拾取和投掷虚拟世界中的物体；

• 玩家可以通过在现实世界中躲避物理攻击来躲避虚拟世界中的攻击；

• 可能出现新的用户界面机会，例如将浮动菜单附加到虚拟手上，或在虚拟世界的白板上看到游戏积分；

• 玩家甚至可以拿起一副虚拟 VR 眼镜并将其戴在头上，从而将它们带入“嵌套”的 VR 世界 - 这种效果最好称为“VR 感知”。

1.4.8.5 线下娱乐

像《Pokémon Go》这样的游戏既不会将图形叠加到现实世界的图像上，也不会生成完全沉浸式的虚拟世界。然而，用户在《Pokémon Go》电脑生成的世界中的视角会根据用户手机或平板电脑的移动做出反应，就像360度全景视频一样。而且游戏能够感知你在现实世界中的实际位置，引导你在附近的公园、商场和餐厅寻找宝可梦。这类游戏实际上不能被称为AR/MR，但也不属于VR范畴。这类游戏或许更适合被描述为一种基于位置的体验。

frames per second is likely to induce disorientation, nausea, and other negative user effects. This means that not only do VR systems need to render the scene twice per frame, they need to do so at 90+ FPS. This is why VR games and applications are generally required to run on high-powered CPU and GPU hardware.

- *Navigation issues*. In an FPS game, the player can simply walk around the game world with the joypad or the WASD keys. In a VR game, a small amount of movement can be realized by the user physically walking around in the real world, but the safe physical play area is typically quite small (the size of a small bathroom or closet). Travelling by “flying” tends to induce nausea as well, so most games opt for a point-and-click teleportation mechanism to move the virtual player/camera across larger distances. Various real-world devices have also been conceived that allow a VR user to “walk” in place with their feet in order to move around in a VR world.

Of course, VR makes up for these limitations somewhat by enabling new user interaction paradigms that aren’t possible in traditional video games. For example,

- users can reach in the real world to touch, pick up and throw objects in the virtual world;
- a player can dodge an attack in the virtual world by dodging physically in the real world;
- new user interface opportunities are possible, such as having floating menus attached to one’s virtual hands, or seeing a game’s credits written on a whiteboard in the virtual world;
- a player can even pick up a pair of virtual VR goggles and place them onto his or her head, thereby transporting them into a “nested” VR world—an effect that might best be called “VR-ception.”

1.4.8.5 Location-Based Entertainment

Games like *Pokémon Go* neither overlay graphics onto an image of the real world, nor do they generate a completely immersive virtual world. However, the user’s view of the computer-generated world of Pokémon Go does react to movements of the user’s phone or tablet, much like a 360-degree video. And the game is aware of your actual location in the real world, prompting you to go searching for Pokémon in nearby parks, malls and restaurants. This kind of game can’t really be called AR/MR, but neither does it fall into the VR category. Such a game might be better described as a form of *location-based*

娱乐，尽管有些人确实使用 AR 来指代这类游戏。

1.4.9 其他类型

当然，还有很多其他游戏类型，我们在此就不深入介绍了。例如：

- 体育，每个主要运动项目（足球、棒球、足球、高尔夫等）都有子类别；
- 角色扮演游戏（RPG）；
- 上帝游戏，例如《Populous》和《Black & White》；
- 环境/社会模拟游戏，如《模拟城市》或《模拟人生》；
- 像俄罗斯方块这样的益智游戏；
- 非电子游戏的转换，如象棋、纸牌游戏、围棋等；
- 网络游戏，例如 Electronic Arts 的 Pogo 网站提供的游戏；

等等。

我们已经看到，每种游戏类型都有其独特的技术要求。这解释了为什么不同游戏类型的引擎传统上存在很大差异。然而，不同游戏类型之间也存在大量的技术重叠，尤其是在单一硬件平台上。随着越来越强大的硬件的出现，由于优化问题而导致的游戏类型之间的差异开始消失。因此，在不同游戏类型甚至不同硬件平台上重用相同的引擎技术变得越来越可能。

1.5 游戏引擎调查

1.5.1 Quake 引擎系列

第一款 3D 第一人称射击 (FPS) 游戏普遍被认为是《德军总部 3D》（1992 年）。这款游戏由德克萨斯州的 id Software 公司为 PC 平台开发，引领游戏行业走向了一个全新而激动人心的方向。id Software 随后开发了《毁灭战士》、《雷神之锤》、《雷神之锤 II》和《雷神之锤 III》。所有这些引擎在架构上都非常相似，我将它们统称为 Quake 引擎系列。Quake 技术已被用于开发许多其他游戏，甚至其他引擎。例如，PC 平台的《荣誉勋章》的系列如下：

entertainment, although some people do use the AR moniker for these kinds of games.

1.4.9 Other Genres

There are of course many other game genres which we won't cover in depth here. Some examples include:

- sports, with subgenres for each major sport (football, baseball, soccer, golf, etc.);
- role-playing games (RPG);
- God games, like *Populous* and *Black & White*;
- environmental/social simulation games, like *SimCity* or *The Sims*;
- puzzle games like *Tetris*;
- conversions of non-electronic games, like chess, card games, go, etc.;
- web-based games, such as those offered at Electronic Arts' Pogo site;

and the list goes on.

We have seen that each game genre has its own particular technological requirements. This explains why game engines have traditionally differed quite a bit from genre to genre. However, there is also a great deal of technological overlap between genres, especially within the context of a single hardware platform. With the advent of more and more powerful hardware, differences between genres that arose because of optimization concerns are beginning to evaporate. It is therefore becoming increasingly possible to reuse the same engine technology across disparate genres, and even across disparate hardware platforms.

1.5 Game Engine Survey

1.5.1 The Quake Family of Engines

The first 3D first-person shooter (FPS) game is generally accepted to be *Castle Wolfenstein 3D* (1992). Written by id Software of Texas for the PC platform, this game led the game industry in a new and exciting direction. id Software went on to create *Doom*, *Quake*, *Quake II* and *Quake III*. All of these engines are very similar in architecture, and I will refer to them as the Quake family of engines. Quake technology has been used to create many other games and even other engines. For example, the lineage of *Medal of Honor* for the PC platform goes something like this:

- Quake III（id Software）；
- 罪孽（仪式）；
- FAKK 2（仪式）；
  《荣誉勋章：联合进攻》（2015 年与梦工厂互动工作室合作）；
- 荣誉勋章：太平洋突袭（电子艺界，洛杉矶）

许多其他基于Quake技术的游戏也经历了同样曲折的历程，并由许多不同的游戏和工作室开发。事实上，Valve的Source引擎（用于开发《半条命》系列游戏）也与Quake技术有着深厚的渊源。

Quake 和 Quake II 的源代码均可免费获取，而且原版 Quake 引擎架构合理且“简洁”（尽管它们略显过时，且完全用 C 语言编写）。这些代码库堪称工业级游戏引擎构建的绝佳范例。Quake 和 Quake II 的完整源代码可在 https://github.com/id-Software/Quake-2 获取。

如果您拥有《雷神之锤》和/或《雷神之锤 II》游戏，您实际上可以使用 Microsoft Visual Studio 构建代码，并使用磁盘中的真实游戏资源在调试器下运行游戏。这将非常有益。您可以设置断点，运行游戏，然后通过单步执行代码来分析引擎的实际工作方式。我强烈建议您下载其中一个或两个引擎，并以这种方式分析源代码。

1.5.2 虚幻引擎

Epic Games, Inc. 于 1998 年凭借其传奇游戏《虚幻》(Unreal) 一举进军 FPS 领域。自那时起，虚幻引擎就成为了 Quake 技术在 FPS 领域的主要竞争对手。虚幻引擎 2 (UE2) 是《虚幻竞技场 2004》(UT2004) 的基础，并已用于无数的“模组”、大学项目和商业游戏。虚幻引擎 4 (UE4) 是最新的进化版，拥有业内一些最优秀的工具和最丰富的引擎功能集，包括一个用于创建着色器的便捷而强大的图形用户界面，以及一个用于游戏逻辑编程的图形用户界面，称为蓝图（以前称为 Kismet）。

虚幻引擎以其丰富的功能集和简洁易用的工具而闻名。虚幻引擎并非完美无缺，大多数开发者都会以各种方式对其进行修改，以便在特定硬件平台上以最佳方式运行游戏。然而，虚幻引擎是一款极其强大的原型设计工具和商业游戏开发平台，几乎可以用于构建任何 3D 第一人称或第三人称游戏（更不用说

- *Quake III* (id Software);
- *Sin* (Ritual);
- *F.A.K.K. 2* (Ritual);
- *Medal of Honor: Allied Assault* (2015 & Dreamworks Interactive); and
- *Medal of Honor: Pacific Assault* (Electronic Arts, Los Angeles).

Many other games based on Quake technology follow equally circuitous paths through many different games and studios. In fact, Valve's Source engine (used to create the *Half-Life* games) also has distant roots in Quake technology.

The *Quake* and *Quake II* source code is freely available, and the original Quake engines are reasonably well architected and "clean" (although they are of course a bit outdated and written entirely in C). These code bases serve as great examples of how industrial-strength game engines are built. The full source code to *Quake* and *Quake II* is available at https://github.com/id-Software/Quake-2.

If you own the Quake and/or Quake II games, you can actually build the code using Microsoft Visual Studio and run the game under the debugger using the real game assets from the disk. This can be incredibly instructive. You can set breakpoints, run the game and then analyze how the engine actually works by stepping through the code. I highly recommend downloading one or both of these engines and analyzing the source code in this manner.

1.5.2 Unreal Engine

Epic Games, Inc. burst onto the FPS scene in 1998 with its legendary game *Unreal*. Since then, the Unreal Engine has become a major competitor to Quake technology in the FPS space. Unreal Engine 2 (UE2) is the basis for *Unreal Tournament 2004* (UT2004) and has been used for countless "mods," university projects and commercial games. Unreal Engine 4 (UE4) is the latest evolutionary step, boasting some of the best tools and richest engine feature sets in the industry, including a convenient and powerful graphical user interface for creating shaders and a graphical user interface for game logic programming called *Blueprints* (previously known as Kismet).

The Unreal Engine has become known for its extensive feature set and cohesive, easy-to-use tools. The Unreal Engine is not perfect, and most developers modify it in various ways to run their game optimally on a particular hardware platform. However, Unreal is an incredibly powerful prototyping tool and commercial game development platform, and it can be used to build virtually any 3D first-person or third-person game (not to mention games in

UE4 还支持多种类型的游戏（以及其他类型的游戏）。许多不同类型的精彩游戏都是使用 UE4 开发的，包括 Tequila Works 的《Rime》、Radiation Blue 的《Genesis: Alpha One》、Hazelight Studios 的《A Way Out》以及 Microsoft Studios 的《Crackdown 3》。

虚幻开发者网络 (UDN) 提供了丰富的文档和其他信息，涵盖虚幻引擎所有已发布的版本（请参阅 http://udn.epicgames.com/Main/WebHome.html）。部分文档可免费获取。然而，最新版本虚幻引擎的完整文档通常仅限于引擎授权用户访问。此外，还有许多其他实用的网站和 wiki 介绍虚幻引擎。其中一个热门网站是 http://www.beyondunreal.com。

值得庆幸的是，Epic 现在提供虚幻引擎 4 的完整访问权限，包括源代码等，只需支付低廉的月费，游戏发行后还能获得一定比例的利润分成。这使得 UE 4 成为小型独立游戏工作室的可行选择。

1.5.3 半条命源引擎

Source 是驱动著名游戏《半条命 2》及其续集《半条命 2：第一章》和《半条命 2：第二章》以及《军团要塞 2》和《传送门》（合称《橙盒》）的游戏引擎。Source 是一款高品质引擎，在图形处理能力和工具集方面可与虚幻引擎 4 相媲美。

1.5.4 DICE 的 Frostbite

寒霜引擎源于 DICE 于 2006 年为《战地：叛逆连队》打造的游戏引擎。自那时起，寒霜引擎已成为美国艺电 (EA) 内部采用最广泛的引擎；EA 的许多主要系列游戏都采用了它，包括《质量效应》、《战地》、《极品飞车》、《龙腾世纪》和《星球大战：前线 II》。寒霜引擎拥有强大的统一资源创建工具 FrostEd、强大的工具管道（称为“后端服务”）以及强大的运行时游戏引擎。由于寒霜引擎是专有引擎，因此很遗憾，EA 以外的开发者无法使用它。

1.5.5 Rockstar 高级游戏引擎 (RAGE)

RAGE 是驱动风靡全球的《侠盗猎车手 V》的引擎。该引擎由 Rockstar Games 旗下 Rockstar 圣地亚哥工作室的 RAGE Technology Group 部门开发，Rockstar Games 的内部工作室已使用 RAGE 开发 PlayStation 4、Xbox One、PlayStation 3、Xbox 360、Wii、Windows 和 MacOS 平台的游戏。其他基于该专有引擎开发的游戏包括《侠盗猎车手 IV》、《荒野大镖客：救赎》和《马克思佩恩 3》。

other genres as well). Many exciting games in all sorts of genres have been developed with UE4, including *Rime* by Tequila Works, *Genesis: Alpha One* by Radiation Blue, *A Way Out* by Hazelight Studios, and *Crackdown 3* by Microsoft Studios.

The Unreal Developer Network (UDN) provides a rich set of documentation and other information about all released versions of the Unreal Engine (see http://udn.epicgames.com/Main/WebHome.html). Some documentation is freely available. However, access to the full documentation for the latest version of the Unreal Engine is generally restricted to licensees of the engine. There are plenty of other useful websites and wikis that cover the Unreal Engine. One popular one is http://www.beyondunreal.com.

Thankfully, Epic now offers full access to Unreal Engine 4, source code and all, for a low monthly subscription fee plus a cut of your game's profits if it ships. This makes UE4 a viable choice for small independent game studios.

1.5.3 The Half-Life Source Engine

Source is the game engine that drives the well-known *Half-Life 2* and its sequels *HL2: Episode One* and *HL2: Episode Two*, *Team Fortress 2* and *Portal* (shipped together under the title *The Orange Box*). Source is a high-quality engine, rivaling Unreal Engine 4 in terms of graphics capabilities and tool set.

1.5.4 DICE's Frostbite

The Frostbite engine grew out of DICE's efforts to create a game engine for *Battlefield Bad Company* in 2006. Since then, the Frostbite engine has become the most widely adopted engine within Electronic Arts (EA); it is used by many of EA's key franchises including *Mass Effect*, *Battlefield*, *Need for Speed*, *Dragon Age*, and *Star Wars Battlefront II*. Frostbite boasts a powerful unified asset creation tool called FrostEd, a powerful tools pipeline known as Backend Services, and a powerful runtime game engine. It is a proprietary engine, so it's unfortunately unavailable for use by developers outside EA.

1.5.5 Rockstar Advanced Game Engine (RAGE)

RAGE is the engine that drives the insanely popular *Grand Theft Auto V*. Developed by RAGE Technology Group, a division of Rockstar Games' Rockstar San Diego studio, RAGE has been used by Rockstar Games' internal studios to develop games for PlayStation 4, Xbox One, PlayStation 3, Xbox 360, Wii, Windows, and MacOS. Other games developed on this proprietary engine include *Grand Theft Auto IV*, *Red Dead Redemption* and *Max Payne 3*.

1.5.6 冷冻发动机

Crytek 最初开发了名为 CRYENGINE 的强大游戏引擎，作为 NVIDIA 的技术演示。当该技术的潜力得到认可后，Crytek 将演示变成了一款完整的游戏，《孤岛惊魂》由此诞生。从那时起，许多游戏都使用 CRYENGINE 制作，包括《孤岛危机》、《代号王国》、《崛起：罗马之子》和《万众狂欢》。多年来，该引擎已发展成为 Crytek 现在的最新产品 CRYENGINE V。这个强大的游戏开发平台提供了一套强大的资产创建工具和一个功能丰富的运行时引擎，具有高质量的实时图形。CRYENGINE 可用于制作针对各种平台的游戏，包括 Xbox One、Xbox 360、PlayStation 4、PlayStation 3、Wii U、Linux、iOS 和 Android。

1.5.7 索尼的PhyreEngine

为了让索尼的 PlayStation 3 平台更容易开发游戏，索尼在 2008 年的游戏开发者大会 (GDC) 上推出了 PhyreEngine。截至 2013 年，PhyreEngine 已发展成为一个强大且功能齐全的游戏引擎，支持一系列令人印象深刻的功能，包括高级照明和延迟渲染。许多工作室使用它来构建 90 多个已发布的游戏，包括 thatgamecompany 的热门游戏 flOw、Flower 和 Journey 以及 Coldwood Interactive 的 Unravel。PhyreEngine 现在支持索尼的 PlayStation 4、PlayStation 3、PlayStation 2、PlayStation Vita 和 PSP 平台。PhyreEngine 让开发者能够使用 PS3 上高度并行的 Cell 架构的强大功能和 PS4 的高级计算功能，以及简化的新世界编辑器和其他强大的游戏开发工具。作为 PlayStation SDK 的一部分，任何获得索尼授权的开发者都可以免费使用它。

1.5.8 微软的XNA游戏工作室

微软的 XNA Game Studio 是一个易于使用且高度可访问的游戏开发平台，基于 C# 语言和公共语言运行时 (CLR)，旨在鼓励玩家创建自己的游戏并与在线游戏社区分享，就像 YouTube 鼓励创建和分享自制视频一样。

不管怎样，微软已于 2014 年正式停用 XNA。不过，开发者可以通过名为 MonoGame 的 XNA 开源实现将他们的 XNA 游戏移植到 iOS、Android、Mac OS X、Linux 和 Windows 8 Metro 平台。更多详情，请访问 https://www.windowscentral.com/xnadead-long-live-xna。

1.5.6 CRYENGINE

Crytek originally developed their powerful game engine known as CRYENGINE as a tech demo for NVIDIA. When the potential of the technology was recognized, Crytek turned the demo into a complete game and *Far Cry* was born. Since then, many games have been made with CRYENGINE including *Crysis*, *Codename Kingdoms*, *Ryse: Son of Rome*, and *Everyone's Gone to the Rapture*. Over the years the engine has evolved into what is now Crytek's latest offering, CRYENGINE V. This powerful game development platform offers a powerful suite of asset-creation tools and a feature-rich runtime engine featuring high-quality real-time graphics. CRYENGINE can be used to make games targeting a wide range of platforms including Xbox One, Xbox 360, PlayStation 4, PlayStation 3, Wii U, Linux, iOS and Android.

1.5.7 Sony's PhyreEngine

In an effort to make developing games for Sony's PlayStation 3 platform more accessible, Sony introduced PhyreEngine at the Game Developer's Conference (GDC) in 2008. As of 2013, PhyreEngine has evolved into a powerful and full-featured game engine, supporting an impressive array of features including advanced lighting and deferred rendering. It has been used by many studios to build over 90 published titles, including thatgamecompany's hits *flOw*, *Flower* and *Journey*, and Coldwood Interactive's *Unravel*. PhyreEngine now supports Sony's PlayStation 4, PlayStation 3, PlayStation 2, PlayStation Vita and PSP platforms. PhyreEngine gives developers access to the power of the highly parallel Cell architecture on PS3 and the advanced compute capabilities of the PS4, along with a streamlined new world editor and other powerful game development tools. It is available free of charge to any licensed Sony developer as part of the PlayStation SDK.

1.5.8 Microsoft's XNA Game Studio

Microsoft's XNA Game Studio is an easy-to-use and highly accessible game development platform based on the C# language and the Common Language Runtime (CLR), and aimed at encouraging players to create their own games and share them with the online gaming community, much as YouTube encourages the creation and sharing of home-made videos.

For better or worse, Microsoft officially retired XNA in 2014. However, developers can port their XNA games to iOS, Android, Mac OS X, Linux and Windows 8 Metro via an open-source implementation of XNA called MonoGame. For more details, see https://www.windowscentral.com/xnadead-long-live-xna.

1.5.9 统一

Unity 是一个强大的跨平台游戏开发环境和运行时引擎，支持多种平台。使用 Unity，开发者可以将游戏部署到移动平台（例如 Apple iOS、Google Android）、游戏主机（例如 Microsoft Xbox 360 和 Xbox One、Sony PlayStation 3 和 PlayStation 4，以及 Nintendo Wii 和 Wii U）、掌上游戏平台（例如 Playstation Vita、Nintendo Switch）、台式电脑（例如 Microsoft Windows、Apple Macintosh 和 Linux）、电视盒（例如 Android TV 和 tvOS）以及虚拟现实 (VR) 系统（例如 Oculus Rift、Steam VR、Gear VR）。

Unity 的主要设计目标是简化开发和跨平台游戏部署。因此，Unity 提供了一个易于使用的集成编辑器环境，您可以在其中创建和操作构成游戏世界的资源和实体，并直接在编辑器中或目标硬件上快速预览游戏的运行效果。Unity 还提供了一套强大的工具，用于在每个目标平台上分析和优化您的游戏，一个全面的资源调节管道，以及在每个部署平台上独特地管理性能与质量权衡的能力。Unity 支持使用 JavaScript、C# 或 Boo 编写脚本；一个强大的动画系统，支持动画重定向（能够在完全不同的角色上播放为一个角色创作的动画）；并支持联网多人游戏。

Unity 已被用于创作众多已发行游戏，包括 N-Fusion/Eidos Montreal 的《杀出重围：陨落》、Team Cherry 的《空洞骑士》以及 StudioMDHR 的颠覆性复古风格游戏《茶杯头》。威比奖获奖短片《亚当》也是使用 Unity 实时渲染的。

1.5.10 其他商业游戏引擎

市面上还有很多其他的商业游戏引擎。虽然独立开发者可能没有足够的预算购买引擎，但许多这类产品都拥有丰富的在线文档和/或 wiki，可以作为游戏引擎和游戏编程方面的宝贵信息来源。例如，可以看看 Terathon Software 的 Tombstone 引擎 (http://tombstoneengine.com/)、LeadWerks 引擎 (https://www.leadwerks.com/) 以及 Idea Fabrik, PLC 的 HeroEngine (http://www.heroengine.com/)。

1.5.11 专有内部引擎

许多公司会构建并维护专有的内部游戏引擎。艺电 (Electronic Arts) 的许多 RTS 游戏都基于 Westwood Studios 开发的专有引擎 Sage 构建。顽皮狗 (Naughty Dog) 的《古惑狼》(Crash Bandicoot) 和

1.5.9 Unity

Unity is a powerful cross-platform game development environment and runtime engine supporting a wide range of platforms. Using Unity, developers can deploy their games on mobile platforms (e.g., Apple iOS, Google Android), consoles (Microsoft Xbox 360 and Xbox One, Sony PlayStation 3 and PlayStation 4, and Nintendo Wii, Wii U), handheld gaming platforms (e.g., Playstation Vita, Nintendo Switch), desktop computers (Microsoft Windows, Apple Macintosh and Linux), TV boxes (e.g., Android TV and tvOS) and virtual reality (VR) systems (e.g., Oculus Rift, Steam VR, Gear VR).

Unity's primary design goals are ease of development and cross-platform game deployment. As such, Unity provides an easy-to-use integrated editor environment, in which you can create and manipulate the assets and entities that make up your game world and quickly preview your game in action right there in the editor, or directly on your target hardware. Unity also provides a powerful suite of tools for analyzing and optimizing your game on each target platform, a comprehensive asset conditioning pipeline, and the ability to manage the performance-quality trade-off uniquely on each deployment platform. Unity supports scripting in JavaScript, C# or Boo; a powerful animation system supporting animation retargeting (the ability to play an animation authored for one character on a totally different character); and support for networked multiplayer games.

Unity has been used to create a wide variety of published games, including *Deus Ex: The Fall* by N-Fusion/Eidos Montreal, *Hollow Knight* by Team Cherry, and the subversive retro-style *Cuphead* by StudioMDHR. The Webby Award winning short film *Adam* was rendered in real time using Unity.

1.5.10 Other Commercial Game Engines

There are lots of other commercial game engines out there. Although indie developers may not have the budget to purchase an engine, many of these products have great online documentation and/or wikis that can serve as a great source of information about game engines and game programming in general. For example, check out the Tombstone engine (http://tombstoneengine.com/) by Terathon Software, the LeadWerks engine (https://www.leadwerks.com/), and HeroEngine by Idea Fabrik, PLC (http://www.heroengine.com/).

1.5.11 Proprietary In-House Engines

Many companies build and maintain proprietary in-house game engines. Electronic Arts built many of its RTS games on a proprietary engine called Sage, developed at Westwood Studios. Naughty Dog's *Crash Bandicoot* and

《杰克与达斯特》系列游戏基于专为 PlayStation 和 PlayStation 2 定制的专有引擎打造。顽皮狗为《神秘海域》系列游戏开发了一款全新的引擎，专为 PlayStation 3 硬件定制。该引擎经过改进，最终用于打造顽皮狗在 PlayStation 3 和 PlayStation 4 平台上的《最后生还者》系列游戏，以及其最新发布的《神秘海域 4：盗贼末路》和《神秘海域：失落的遗产》。当然，大多数获得商业授权的游戏引擎，例如 Quake、Source、虚幻引擎 4 和 CRYENGINE，最初都是顽皮狗的专有内部引擎。

1.5.12 开源引擎

开源 3D 游戏引擎是由业余和专业游戏开发者构建并在线免费提供的引擎。“开源”一词通常意味着源代码可免费获取，并且采用某种程度上开放的开发模式，这意味着几乎任何人都可以贡献代码。许可证（如果有的话）通常遵循 GNU 公共许可证 (GPL) 或宽 GNU 公共许可证 (LGPL)。前者允许任何人自由使用代码，只要其代码也免费提供；后者甚至允许在专有的营利性应用程序中使用代码。开源项目还有许多其他免费和半免费的许可方案。

网络上开源引擎的数量多得惊人。有些相当不错，有些则平庸无奇，还有一些简直糟透了！http://en.wikipedia.org/wiki/List_of_game_engines 提供的在线游戏引擎列表能让你感受到引擎的庞大数量。（http://www.worldofleveldesign.com/categories/level_design_tutorials/recommended-game-engines.php 上的列表更容易理解。）这两个列表都包含了开源和商业游戏引擎。

OGRE 是一款架构精良、易于学习和使用的 3D 渲染引擎。它拥有功能齐全的 3D 渲染器，包括先进的光照和阴影处理、优秀的骨骼角色动画系统、用于抬头显示器和图形用户界面的二维叠加系统，以及用于实现诸如光晕等全屏效果的后处理系统。OGRE 的开发者自己也承认，它并非一个完整的游戏引擎，但它提供了几乎所有游戏引擎所需的许多基础组件。

这里列出了一些其他著名的开源引擎：

- Panda3D 是一款基于脚本的引擎。该引擎的主要接口是 Python 自定义脚本语言。它旨在使原型设计

Jak and Daxter franchises were built on a proprietary engine custom tailored to the PlayStation and PlayStation 2. For the *Uncharted* series, Naughty Dog developed a brand new engine custom tailored to the PlayStation 3 hardware. This engine evolved and was ultimately used to create Naughty Dog's *The Last of Us* series on the PlayStation 3 and PlayStation 4, as well as its most recent releases, *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy*. And of course, most commercially licensed game engines like Quake, Source, Unreal Engine 4 and CRYENGINE all started out as proprietary in-house engines.

1.5.12 Open Source Engines

Open source 3D game engines are engines built by amateur and professional game developers and provided online for free. The term "open source" typically implies that source code is freely available and that a somewhat open development model is employed, meaning almost anyone can contribute code. Licensing, if it exists at all, is often provided under the Gnu Public License (GPL) or Lesser Gnu Public License (LGPL). The former permits code to be freely used by anyone, as long as their code is also freely available; the latter allows the code to be used even in proprietary for-profit applications. Lots of other free and semi-free licensing schemes are also available for open source projects.

There are a staggering number of open source engines available on the web. Some are quite good, some are mediocre and some are just plain awful! The list of game engines provided online at http://en.wikipedia.org/wiki/List_of_game_engines will give you a feel for the sheer number of engines that are out there. (The list at http://www.worldofleveldesign.com/categories/level_design_tutorials/recommended-game-engines.php is a bit more digestible.) Both of these lists include both open-source and commercial game engines.

OGRE is a well-architected, easy-to-learn and easy-to-use 3D rendering engine. It boasts a fully featured 3D renderer including advanced lighting and shadows, a good skeletal character animation system, a two-dimensional overlay system for heads-up displays and graphical user interfaces, and a post-processing system for full-screen effects like bloom. OGRE is, by its authors' own admission, not a full game engine, but it does provide many of the foundational components required by pretty much any game engine.

Some other well-known open source engines are listed here:

- Panda3D is a script-based engine. The engine's primary interface is the Python custom scripting language. It is designed to make prototyping

3D游戏和虚拟世界方便快捷。

• Yake 是一个基于 OGRE 构建的游戏引擎。

• Crystal Space 是一款具有可扩展模块化架构的游戏引擎。

• Torque 和 Irrlicht 也是著名的开源游戏引擎。

• 虽然从技术上讲 Lumberyard 引擎并非开源，但它确实向其开发者提供了源代码。它是一款由亚马逊开发的免费跨平台引擎，基于 CRYENGINE 架构。

1.5.13 面向非程序员的 2D 游戏引擎

随着苹果 iPhone/iPad 和谷歌 Android 等平台上休闲网页游戏和移动游戏的蓬勃发展，二维游戏也变得异常流行。许多流行的游戏/多媒体创作工具包应运而生，使小型游戏工作室和独立开发者能够为这些平台创作 2D 游戏。这些工具包注重易用性，允许用户使用图形用户界面 (GUI) 来创建游戏，而无需使用编程语言。观看此 YouTube 视频，了解您可以使用这些工具包创建哪些类型的游戏：https://www.youtube.com/watch?v= 3Zq1yo0lxOU

• Multimedia Fusion 2 (http://www.clickteam.com/website/world) 是由 Clickteam 开发的一款 2D 游戏/多媒体创作工具包。业内专业人士使用 Fusion 来创建游戏、屏幕保护程序和其他多媒体应用程序。PlanetBravo (http://www.planetbravo.com) 等教育机构也使用 Fusion 及其更简单的版本 The Games Factory 2 来教授儿童游戏开发和编程/逻辑概念。Fusion 支持 iOS、Android、Flash 和 Java 平台。

• Game Salad Creator (http://gamesalad.com/creator) 是另一款针对非程序员的图形游戏/多媒体创作工具包，在许多方面与 Fusion 相似。

• Scratch (http://scratch.mit.edu) 是一款创作工具包和图形化编程语言，可用于创建交互式演示和简单游戏。它是年轻人学习条件语句、循环和事件驱动编程等编程概念的绝佳途径。Scratch 由麻省理工学院媒体实验室的 Mitchel Resnick 领导的终身幼儿园小组于 2003 年开发。

3D games and virtual worlds convenient and fast.

- Yake is a game engine built on top of OGRE.
- Crystal Space is a game engine with an extensible modular architecture.
- Torque and Irrlicht are also well-known open-source game engines.
- While not technically open-source, the Lumberyard engine does provide source code to its developers. It is a free cross-platform engine developed by Amazon, and based on the CRYENGINE architecture.

1.5.13 2D Game Engines for Non-programmers

Two-dimensional games have become incredibly popular with the recent explosion of casual web gaming and mobile gaming on platforms like Apple iPhone/iPad and Google Android. A number of popular game/multimedia authoring toolkits have become available, enabling small game studios and independent developers to create 2D games for these platforms. These toolkits emphasize ease of use and allow users to employ a graphical user interface to create a game rather than requiring the use of a programming language. Check out this YouTube video to get a feel for the kinds of games you can create with these toolkits: https://www.youtube.com/watch?v= 3Zq1yo0lxOU

- *Multimedia Fusion 2* (http://www.clickteam.com/website/world is a 2D game/multimedia authoring toolkit developed by Clickteam. Fusion is used by industry professionals to create games, screen savers and other multimedia applications. Fusion and its simpler counterpart, The Games Factory 2, are also used by educational camps like PlanetBravo (http://www.planetbravo.com) to teach kids about game development and programming/logic concepts. Fusion supports the iOS, Android, Flash, and Java platforms.
- *Game Salad Creator* (http://gamesalad.com/creator) is another graphical game/multimedia authoring toolkit aimed at non-programmers, similar in many respects to Fusion.
- *Scratch* (http://scratch.mit.edu) is an authoring toolkit and graphical programming language that can be used to create interactive demos and simple games. It is a great way for young people to learn about programming concepts such as conditionals, loops and event-driven programming. Scratch was developed in 2003 by the Lifelong Kindergarten group, led by Mitchel Resnick at the MIT Media Lab.

1.6 运行时引擎架构

游戏引擎通常由工具套件和运行时组件组成。我们将首先探讨运行时组件的架构，然后在下一节中探讨工具架构。

图 1.16 展示了构成典型 3D 游戏引擎的所有主要运行时组件。没错，它很大！而且这张图甚至还没有涵盖所有工具。游戏引擎绝对是大型软件系统。

与所有软件系统一样，游戏引擎也是分层构建的。通常，上层依赖于下层，但反之则不然。当下层依赖于上层时，我们称之为循环依赖。任何软件系统中都应避免依赖循环，因为它们会导致系统间不良耦合，使软件难以测试，并阻碍代码复用。对于像游戏引擎这样的大型系统尤其如此。

以下是图 1.16 中所示组件的简要概述。本书的其余部分将更深入地研究每个组件，并学习它们通常如何集成到一个功能整体中。

1.6.1 目标硬件

目标硬件层代表游戏将在其上运行的计算机系统或主机。典型的平台包括基于 Microsoft Windows、Linux 和 MacOS 的 PC；移动平台，例如 Apple iPhone 和 iPad、Android 智能手机和平板电脑、索尼的 PlayStation Vita 和亚马逊的 Kindle Fire（等等）；以及游戏主机，例如 Microsoft 的 Xbox、Xbox 360 和 Xbox One，索尼的 PlayStation、PlayStation 2、PlayStation 3 和 PlayStation 4，以及任天堂的 DS、GameCube、Wii、Wii U 和 Switch。本书中的大多数主题与平台无关，但我们也会涉及一些与 PC 或主机开发相关的设计考虑因素，这些因素与平台无关。

1.6.2 设备驱动程序

设备驱动程序是由操作系统或硬件供应商提供的低级软件组件。驱动程序负责管理硬件资源，并屏蔽操作系统和上层引擎与各种硬件设备通信的细节。

1.6 Runtime Engine Architecture

A game engine generally consists of a tool suite and a runtime component. We'll explore the architecture of the runtime piece first and then get into tool architecture in the following section.

Figure 1.16 shows all of the major runtime components that make up a typical 3D game engine. Yeah, it's *big!* And this diagram doesn't even account for all the tools. Game engines are definitely large software systems.

Like all software systems, game engines are built in *layers*. Normally upper layers depend on lower layers, but not vice versa. When a lower layer depends upon a higher layer, we call this a *circular dependency*. Dependency cycles are to be avoided in any software system, because they lead to undesirable coupling between systems, make the software untestable and inhibit code reuse. This is especially true for a large-scale system like a game engine.

What follows is a brief overview of the components shown in the diagram in Figure 1.16. The rest of this book will be spent investigating each of these components in a great deal more depth and learning how these components are usually integrated into a functional whole.

1.6.1 Target Hardware

The target hardware layer represents the computer system or console on which the game will run. Typical platforms include Microsoft Windows, Linux and MacOS-based PCs; mobile platforms like the Apple iPhone and iPad, Android smart phones and tablets, Sony's PlayStation Vita and Amazon's Kindle Fire (among others); and game consoles like Microsoft's Xbox, Xbox 360 and Xbox One, Sony's PlayStation, PlayStation 2, PlayStation 3 and PlayStation 4, and Nintendo's DS, GameCube, Wii, Wii U and Switch. Most of the topics in this book are platform-agnostic, but we'll also touch on some of the design considerations peculiar to PC or console development, where the distinctions are relevant.

1.6.2 Device Drivers

Device drivers are low-level software components provided by the operating system or hardware vendor. Drivers manage hardware resources and shield the operating system and upper engine layers from the details of communicating with the myriad variants of hardware devices available.

图 1.16.运行时游戏引擎架构。

Figure 1.16. Runtime game engine architecture.

1.6.3 操作系统

在 PC 上，操作系统 (OS) 始终处于运行状态。它协调单台计算机上多个程序的执行，其中之一就是您的游戏。像 Microsoft Windows 这样的操作系统采用时间片式方法，让多个正在运行的程序共享硬件，这被称为抢占式多任务处理。这意味着 PC 游戏永远不能假设自己完全控制了硬件——它必须与系统中的其他程序“友好配合”。

在早期的游戏机上，操作系统（如果存在的话）只是一个薄薄的库层，直接编译到游戏可执行文件中。在这些早期系统中，游戏在运行时“拥有”整台机器。然而，在现代游戏机上，情况已不再如此。Xbox 360、PlayStation 3、Xbox One 和 PlayStation 4 上的操作系统可以中断游戏执行或接管某些系统资源，例如显示在线消息，或允许玩家暂停游戏并调出 PS4 的“XMB”用户界面或 Xbox One 的仪表盘。在 PS4 和 Xbox One 上，操作系统会持续运行后台任务，例如录制游戏过程视频以便您决定通过 PS4 的“分享”按钮分享，或者下载游戏、补丁和 DLC，以便您在等待游戏时享受游戏的乐趣。因此，游戏机和 PC 开发之间的差距正在逐渐缩小（无论好坏）。

1.6.4 第三方 SDK 和中间件

大多数游戏引擎都利用了许多第三方软件开发工具包 (SDK) 和中间件，如图 1.17 所示。SDK 提供的函数式或基于类的接口通常称为应用程序编程接口 (API)。我们将看几个例子。

图 1.17. 第三方 SDK 层。

1.6.4.1 数据结构和算法

与任何软件系统一样，游戏严重依赖容器数据结构和算法来操作它们。以下是一些提供此类服务的第三方库的示例：

1.6.3 Operating System

On a PC, the operating system (OS) is running all the time. It orchestrates the execution of multiple programs on a single computer, one of which is your game. Operating systems like Microsoft Windows employ a time-sliced approach to sharing the hardware with multiple running programs, known as preemptive multitasking. This means that a PC game can never assume it has full control of the hardware—it must “play nice” with other programs in the system.

On early consoles, the operating system, if one existed at all, was just a thin library layer that was compiled directly into your game executable. On those early systems, the game “owned” the entire machine while it was running. However, on modern consoles this is no longer the case. The operating system on the Xbox 360, PlayStation 3, Xbox One and PlayStation 4 can interrupt the execution of your game, or take over certain system resources, in order to display online messages, or to allow the player to pause the game and bring up the PS4’s “XMB” user interface or the Xbox One’s dashboard, for example. On the PS4 and Xbox One, the OS is continually running background tasks, such as recording video of your playthrough in case you decide to share it via the PS4’s Share button, or downloading games, patches and DLC, so you can have fun playing a game while you wait. So the gap between console and PC development is gradually closing (for better or for worse).

1.6.4 Third-Party SDKs and Middleware

Most game engines leverage a number of third-party software development kits (SDKs) and middleware, as shown in Figure 1.17. The functional or class-based interface provided by an SDK is often called an application programming interface (API). We will look at a few examples.

Figure 1.17. Third-party SDK layer.

1.6.4.1 Data Structures and Algorithms

Like any software system, games depend heavily on *container* data structures and algorithms to manipulate them. Here are a few examples of third-party libraries that provide these kinds of services:

• Boost。Boost 是一个强大的数据结构和算法库，其设计风格借鉴了标准 C++ 库及其前身标准模板库 (STL)。（Boost 的在线文档也是学习计算机科学知识的好地方！）

• Folly 。Folly 是 Facebook 使用的库，其目标是通过各种有用的功能扩展标准 C++ 库和 Boost，重点是最大限度地提高代码性能。

• Loki . Loki 是一个功能强大的通用编程模板库，它会让你的大脑痛苦不堪！

C++ 标准库和 STL

C++ 标准库也提供了很多与 Boost 等第三方库中相同的功能。标准库中实现泛型容器类（例如 std::vector 和 std::list）的子集通常被称为标准模板库 (STL)，尽管从技术角度来看，这有点用词不当：标准模板库是由 Alexander Stepanov 和 David Musser 在 C++ 语言标准化之前编写的。该库的大部分功能被吸收到现在的 C++ 标准库中。本书中使用的术语“STL”通常是指提供泛型容器类的 C++ 标准库子集，而不是原始的 STL。

1.6.4.2 图形

大多数游戏渲染引擎都是建立在硬件接口库之上的，例如：

• Glide 是适用于旧版 Voodoo 显卡的 3D 图形 SDK。该 SDK 在 DirectX 7 开启的硬件变换和光照（硬件 T&L）时代之前就很流行。

• OpenGL 是一种广泛使用的便携式 3D 图形 SDK。

• DirectX 是 Microsoft 的 3D 图形 SDK，也是 OpenGL 的主要竞争对手。

• libgcm 是 PlayStation 3 的 RSX 图形硬件的低级直接接口，由索尼提供，作为 OpenGL 的更高效替代方案。

• Edge 是一款功能强大且高效的渲染和动画引擎，由顽皮狗和索尼为 PlayStation 3 制作，并被许多第一方和第三方游戏工作室使用。

- *Boost*. Boost is a powerful data structures and algorithms library, designed in the style of the standard C++ library and its predecessor, the standard template library (STL). (The online documentation for Boost is also a great place to learn about computer science in general!)
- *Folly*. Folly is a library used at Facebook whose goal is to extend the standard C++ library and Boost with all sorts of useful facilities, with an emphasis on maximizing code performance.
- *Loki*. Loki is a powerful generic programming template library which is exceedingly good at making your brain hurt!

The C++ Standard Library and STL

The C++ standard library also provides many of the same kinds of facilities found in third-party libraries like Boost. The subset of the standard library that implements generic container classes such as `std::vector` and `std::list` is often referred to as the *standard template library* (STL), although this is technically a bit of a misnomer: The standard template library was written by Alexander Stepanov and David Musser in the days before the C++ language was standardized. Much of this library's functionality was absorbed into what is now the C++ standard library. When we use the term STL in this book, it's usually in the context of the subset of the C++ standard library that provides generic container classes, not the original STL.

1.6.4.2 Graphics

Most game rendering engines are built on top of a hardware interface library, such as the following:

- *Glide* is the 3D graphics SDK for the old Voodoo graphics cards. This SDK was popular prior to the era of hardware transform and lighting (hardware T&L) which began with DirectX 7.
- *OpenGL* is a widely used portable 3D graphics SDK.
- *DirectX* is Microsoft's 3D graphics SDK and primary rival to OpenGL.
- *libgcm* is a low-level direct interface to the PlayStation 3's RSX graphics hardware, which was provided by Sony as a more efficient alternative to OpenGL.
- *Edge* is a powerful and highly efficient rendering and animation engine produced by Naughty Dog and Sony for the PlayStation 3 and used by a number of first- and third-party game studios.

• Vulkan 是由 KhronosTM Group 创建的低级库，它使游戏程序员能够将渲染批次和 GPGPU 计算作业以命令列表的形式直接提交给 GPU，并让他们能够对 CPU 和 GPU 之间共享的内存和其他资源进行精细控制。（有关 GPGPU 编程的更多信息，请参阅第 4.11 节。）

1.6.4.3 碰撞和物理

碰撞检测和刚体动力学（在游戏开发社区中简称为“物理”）由以下知名 SDK 提供：

• Havok 是一款流行的工业强度物理和碰撞引擎。
• PhysX 是另一种流行的工业强度物理和碰撞引擎，可从 NVIDIA 免费下载。
• Open Dynamics Engine (ODE) 是一个著名的开源物理/碰撞包。

1.6.4.4 角色动画

有许多商业动画包，包括但不限于以下内容：

• Granny。Rad Game Tools 广受欢迎的 Granny 工具包包含强大的 3D 模型和动画导出器，适用于所有主流 3D 建模和动画软件，例如 Maya、3D Studio MAX 等；一个用于读取和处理导出模型和动画数据的运行时库；以及一个强大的运行时动画系统。在我看来，Granny SDK 拥有我见过的所有动画 API（无论是商业的还是专有的），设计最精良、逻辑最清晰，尤其是在时间处理方面。

• Havok 动画。随着角色变得越来越逼真，物理和动画之间的界限变得越来越模糊。开发广受欢迎的 Havok 物理 SDK 的公司决定创建一个免费的动画 SDK，这将使弥合物理和动画之间的差距变得前所未有的容易。

• OrbisAnim。SN Systems 与顽皮狗的 ICE 和游戏团队、索尼互动娱乐的工具和技术部门以及索尼欧洲先进技术部门合作，为 PS4 制作了 OrbisAnim 库，其中包含一个强大而高效的动画引擎和一个用于渲染的高效几何处理引擎。

- *Vulkan* is a low-level library created by the Khronos™ Group which enables game programmers to submit rendering batches and GPGPU compute jobs directly to the GPU as command lists, and provides them with fine-grained control over memory and other resources that are shared between the CPU and GPU. (See Section 4.11 for more on GPGPU programming.)

1.6.4.3 Collision and Physics

Collision detection and rigid body dynamics (known simply as “physics” in the game development community) are provided by the following well-known SDKs:

- *Havok* is a popular industrial-strength physics and collision engine.
- *PhysX* is another popular industrial-strength physics and collision engine, available for free download from NVIDIA.
- *Open Dynamics Engine (ODE)* is a well-known open source physics/collision package.

1.6.4.4 Character Animation

A number of commercial animation packages exist, including but certainly not limited to the following:

- *Granny*. Rad Game Tools’ popular Granny toolkit includes robust 3D model and animation exporters for all the major 3D modeling and animation packages like Maya, 3D Studio MAX, etc., a runtime library for reading and manipulating the exported model and animation data, and a powerful runtime animation system. In my opinion, the Granny SDK has the best-designed and most logical animation API of any I’ve seen, commercial or proprietary, especially its excellent handling of time.
- *Havok Animation*. The line between physics and animation is becoming increasingly blurred as characters become more and more realistic. The company that makes the popular Havok physics SDK decided to create a complimentary animation SDK, which makes bridging the physics-animation gap much easier than it ever has been.
- *OrbisAnim*. The OrbisAnim library produced for the PS4 by SN Systems in conjunction with the ICE and game teams at Naughty Dog, the Tools and Technology group of Sony Interactive Entertainment, and Sony’s Advanced Technology Group in Europe includes a powerful and efficient animation engine and an efficient geometry-processing engine for rendering.

1.6.4.5 生物力学角色模型

- Endorphin 和 Euphoria。这些动画包使用逼真的人体运动的高级生物力学模型来制作角色动作。

正如我们之前提到的，角色动画和物理之间的界限正开始变得模糊。像 Havok Animation 这样的软件试图以传统的方式将物理和动画结合起来，由人类动画师通过 Maya 之类的工具提供大部分动作，并在运行时由物理引擎增强这些动作。但一家名为 Natural Motion Ltd. 的公司推出了一款产品，试图重新定义游戏和其他数字媒体中角色动作的处理方式。

其首款产品 Endorphin 是一款 Maya 插件，允许动画师对角色进行完整的生物力学模拟，并将生成的动画导出，如同手工制作的动画一样。该生物力学模型考虑了角色的重心、体重分布，以及真实人类在重力和其他力量作用下如何保持平衡和移动的细节。

其第二款产品 Euphoria 是 Endorphin 的实时版本，旨在在运行时在不可预测的力量的影响下产生物理和生物力学上准确的角色运动。

1.6.5 平台独立层

大多数游戏引擎都需要能够在多个硬件平台上运行。例如，像 Electronic Arts 和 ActivisionBlizzard Inc. 这样的公司总是将他们的游戏定位于各种各样的平台，因为这样可以让他们的游戏接触到尽可能多的市场。通常，唯一没有为每个游戏至少瞄准两个不同平台的游戏工作室是第一方工作室，例如索尼的顽皮狗和 Insomniac 工作室。因此，大多数游戏引擎都采用平台独立层构建，如图 1.18 所示。该层位于硬件、驱动程序、操作系统和其他第三方软件之上，通过将某些接口函数“包装”在自定义函数中，将引擎的其余部分与底层平台的大部分知识隔离开来，游戏开发者可以控制这些自定义函数在每个目标平台上。

将函数“包装”为游戏引擎平台独立层的一部分有两个主要原因：首先，一些应用程序编程接口（API），例如操作系统提供的接口，甚至是一些较旧的“标准”库（例如 C 标准库）中的函数，

1.6.4.5 Biomechanical Character Models

- *Endorphin and Euphoria*. These are animation packages that produce character motion using advanced biomechanical models of realistic human movement.

As we mentioned previously, the line between character animation and physics is beginning to blur. Packages like Havok Animation try to marry physics and animation in a traditional manner, with a human animator providing the majority of the motion through a tool like Maya and with physics augmenting that motion at runtime. But a firm called Natural Motion Ltd. has produced a product that attempts to redefine how character motion is handled in games and other forms of digital media.

Its first product, Endorphin, is a Maya plug-in that permits animators to run full biomechanical simulations on characters and export the resulting animations as if they had been hand animated. The biomechanical model accounts for center of gravity, the character's weight distribution, and detailed knowledge of how a real human balances and moves under the influence of gravity and other forces.

Its second product, Euphoria, is a real-time version of Endorphin intended to produce physically and biomechanically accurate character motion at runtime under the influence of unpredictable forces.

1.6.5 Platform Independence Layer

Most game engines are required to be capable of running on more than one hardware platform. Companies like Electronic Arts and ActivisionBlizzard Inc., for example, always target their games at a wide variety of platforms because it exposes their games to the largest possible market. Typically, the only game studios that do not target at least two different platforms per game are first-party studios, like Sony's Naughty Dog and Insomniac studios. Therefore, most game engines are architected with a platform independence layer, like the one shown in Figure 1.18. This layer sits atop the hardware, drivers, operating system and other third-party software and shields the rest of the engine from the majority of knowledge of the underlying platform by "wrapping" certain interface functions in custom functions over which you, the game developer, will have control on every target platform.

There are two primary reasons to "wrap" functions as part of your game engine's platform independence layer like this: First, some application programming interfaces (APIs), like those provided by the operating system, or even some functions in older "standard" libraries like the C standard library,

不同平台之间存在显著差异；封装这些函数可以为引擎的其余部分在所有目标平台上提供一致的 API。其次，即使使用像 Havok 这样的完全跨平台库，您也可能需要避免未来变更的影响，例如将来将引擎迁移到不同的碰撞/物理库。

图 1.18. 平台独立层。

1.6.6 核心系统

每个游戏引擎，以及实际上每个大型复杂的 C++ 软件应用程序，都需要一系列实用的软件工具。我们将这些工具归类为“核心系统”。图 1.19 展示了一个典型的核心系统层。

以下是核心层通常提供的设施的几个示例：

- 断言是插入的错误检查代码行，用于捕获逻辑错误以及违反程序员原始假设的情况。断言检查通常会从游戏的最终生产版本中删除。（断言将在 3.2.3.3 节中介绍。）

- 内存管理。几乎每个游戏引擎都实现了自己的自定义内存分配系统，以确保高速分配和释放，并限制内存碎片的负面影响（参见第 6.2.1 节）。

- 数学库。游戏本质上是高度数学密集型的。
  因此，每个游戏引擎至少都有一个（如果不是多个的话）数学库。这些库提供向量和矩阵数学、四元数旋转、三角学、直线、射线、球体、视锥体等的几何运算、样条函数操作、数值积分、方程组求解以及游戏程序员所需的任何其他功能。

- 自定义数据结构和算法。除非引擎设计者决定完全依赖第三方包（例如 Boost 和 Folly），否则通常需要一套用于管理基本数据结构（链表、动态数组、二叉树、哈希映射等）和算法（搜索、排序等）的工具。这些工具通常手工编写，以最大限度地减少或消除动态内存分配，并确保在目标平台上实现最佳的运行时性能。

differ significantly from platform to platform; wrapping these functions provides the rest of your engine with a consistent API across all of your targeted platforms. Second, even when using a fully cross-platform library such as Havok, you might want to insulate yourself from future changes, such as transitioning your engine to a different collision/physics library in the future.

Figure 1.18. Platform independence layer.

1.6.6 Core Systems

Every game engine, and really every large, complex C++ software application, requires a grab bag of useful software utilities. We'll categorize these under the label "core systems." A typical core systems layer is shown in Figure 1.19. Here are a few examples of the facilities the core layer usually provides:

- *Assertions* are lines of error-checking code that are inserted to catch logical mistakes and violations of the programmer's original assumptions. Assertion checks are usually stripped out of the final production build of the game. (Assertions are covered in Section 3.2.3.3.)
- *Memory management*. Virtually every game engine implements its own custom memory allocation system(s) to ensure high-speed allocations and deallocations and to limit the negative effects of memory fragmentation (see Section 6.2.1).
- *Math library*. Games are by their nature highly mathematics-intensive. As such, every game engine has at least one, if not many, math libraries. These libraries provide facilities for vector and matrix math, quaternion rotations, trigonometry, geometric operations with lines, rays, spheres, frusta, etc., spline manipulation, numerical integration, solving systems of equations and whatever other facilities the game programmers require.
- *Custom data structures and algorithms*. Unless an engine's designers decided to rely entirely on third-party packages such as Boost and Folly, a suite of tools for managing fundamental data structures (linked lists, dynamic arrays, binary trees, hash maps, etc.) and algorithms (search, sort, etc.) is usually required. These are often hand coded to minimize or eliminate dynamic memory allocation and to ensure optimal runtime performance on the target platform(s).

图 1.19.核心引擎系统。

有关最常见的核心引擎系统的详细讨论可在第二部分中找到。

1.6.7 资源管理器

资源管理器以某种形式存在于每个游戏引擎中，它提供了一个统一的接口（或一套接口），用于访问所有类型的游戏资源和其他引擎输入数据。一些引擎以高度集中和一致的方式实现这一点（例如，虚幻引擎的资源包、OGRE 的资源管理器类）。其他引擎则采用临时方法，通常让游戏程序员直接访问磁盘上的原始文件或压缩包（例如 Quake 的 PAK 文件）。图 1.20 描绘了一个典型的资源管理器层。

图 1.20。资源管理器。

1.6.8 渲染引擎

渲染引擎是任何游戏引擎中最大、最复杂的组件之一。渲染器的架构设计有很多种方式。目前还没有一种公认的方法，但正如我们将看到的，大多数现代渲染引擎都遵循一些基本的设计理念，这在很大程度上是由它们所依赖的 3D 图形硬件的设计所驱动的。

渲染引擎设计的一种常见且有效的方法是采用如下的分层架构。

Figure 1.19. Core engine systems.

A detailed discussion of the most common core engine systems can be found in Part II.

1.6.7 Resource Manager

Present in every game engine in some form, the resource manager provides a unified interface (or suite of interfaces) for accessing any and all types of game assets and other engine input data. Some engines do this in a highly centralized and consistent manner (e.g., Unreal's packages, OGRE's `ResourceManager` class). Other engines take an ad hoc approach, often leaving it up to the game programmer to directly access raw files on disk or within compressed archives such as Quake's PAK files. A typical resource manager layer is depicted in Figure 1.20.

Figure 1.20. Resource manager.

1.6.8 Rendering Engine

The rendering engine is one of the largest and most complex components of any game engine. Renderers can be architected in many different ways. There is no one accepted way to do it, although as we'll see, most modern rendering engines share some fundamental design philosophies, driven in large part by the design of the 3D graphics hardware upon which they depend.

One common and effective approach to rendering engine design is to employ a layered architecture as follows.

1.6.8.1 低级渲染器

低级渲染器（如图 1.21 所示）涵盖了引擎的所有原始渲染功能。在这一层级，设计重点在于尽可能快速且丰富地渲染一组几何图元，而不太考虑场景中哪些部分是可见的。该组件分为多个子组件，我们将在下文进行讨论。

图 1.21。低级渲染引擎。

图形设备接口

图形 SDK（例如 DirectX、OpenGL 或 Vulkan）需要编写相当多的代码，仅仅是为了枚举可用的图形设备、初始化它们、设置渲染表面（后台缓冲区、模板缓冲区等）等等。这些通常由一个我称之为图形设备接口的组件来处理（尽管每个引擎都有自己的术语）。

对于 PC 游戏引擎，您还需要编写代码将渲染器与 Windows 消息循环集成。通常，您需要编写一个“消息泵”，用于在 Windows 消息待处理时为其提供服务，否则，它会尽可能快地反复运行渲染循环。这会将游戏的键盘轮询循环与渲染器的屏幕更新循环绑定在一起。这种耦合并不理想，但只要付出一些努力，就可以最大限度地减少依赖关系。我们稍后会更深入地探讨这个主题。

其他渲染器组件

低级渲染器中的其他组件相互协作，以收集几何图元（有时称为渲染包）的提交，例如网格、线列表、点列表、粒子、地形块、文本字符串以及您想要绘制的任何其他内容，并尽快渲染它们。

1.6.8.1 Low-Level Renderer

The *low-level renderer*, shown in Figure 1.21, encompasses all of the raw rendering facilities of the engine. At this level, the design is focused on rendering a collection of geometric primitives as quickly and richly as possible, without much regard for which portions of a scene may be visible. This component is broken into various subcomponents, which are discussed below.

Figure 1.21. Low-level rendering engine.

Graphics Device Interface

Graphics SDKs, such as DirectX, OpenGL or Vulkan, require a reasonable amount of code to be written just to enumerate the available graphics devices, initialize them, set up render surfaces (back-buffer, stencil buffer, etc.) and so on. This is typically handled by a component that I'll call the *graphics device interface* (although every engine uses its own terminology).

For a PC game engine, you also need code to integrate your renderer with the Windows message loop. You typically write a “message pump” that services Windows messages when they are pending and otherwise runs your render loop over and over as fast as it can. This ties the game's keyboard polling loop to the renderer's screen update loop. This coupling is undesirable, but with some effort it is possible to minimize the dependencies. We'll explore this topic in more depth later.

Other Renderer Components

The other components in the low-level renderer cooperate in order to collect submissions of *geometric primitives* (sometimes called *render packets*), such as meshes, line lists, point lists, particles, terrain patches, text strings and whatever else you want to draw, and render them as quickly as possible.

图 1.22. 典型的场景图/空间细分层，用于剔除优化。

低级渲染器通常提供一个视口抽象，其中包含关联的相机到世界矩阵和 3D 投影参数，例如视野以及近裁剪平面和远裁剪平面的位置。低级渲染器还通过其材质系统和动态光照系统管理图形硬件和游戏着色器的状态。每个提交的图元都与一种材质相关联，并受 n 个动态光源的影响。材质描述了图元使用的纹理、需要生效的设备状态设置以及渲染图元时要使用的顶点和像素着色器。光源决定了如何将动态光照计算应用于图元。光照和着色是一个复杂的主题。我们将在第 11 章讨论基础知识，但这些主题在许多优秀的计算机图形学书籍中都有深入的介绍，包括 [16]、[49] 和 [2]。

1.6.8.2 场景图/剔除优化

低级渲染器会绘制所有提交给它的几何体，而不太考虑这些几何体是否可见（除了背面剔除和将三角形裁剪到相机视锥体之外）。通常需要一个更高级别的组件，根据某种形式的可见性判断来限制提交渲染的图元数量。该层如图 1.22 所示。

对于非常小的游戏世界，简单的视锥体剔除（即移除摄像机无法“看到”的物体）可能就足够了。对于更大的游戏世界，可以使用更高级的空间细分数据结构来快速确定物体的潜在可见集 (PVS)，从而提高渲染效率。空间细分可以采用多种形式，包括二叉空间分割树、四叉树、八叉树、k-dtree 或球体层次结构。空间细分有时也称为场景图，尽管从技术上讲，后者是一种特殊的数据结构，并不包含前者。门户或遮挡剔除方法也可能应用于渲染引擎的这一层。

理想情况下，低级渲染器应该完全不受所使用的空间细分或场景图类型的限制。这允许不同的游戏

Figure 1.22. A typical scene graph/spatial subdivision layer, for culling optimization.

The low-level renderer usually provides a viewport abstraction with an associated camera-to-world matrix and 3D projection parameters, such as field of view and the location of the near and far clip planes. The low-level renderer also manages the state of the graphics hardware and the game's shaders via its *material system* and its *dynamic lighting system*. Each submitted primitive is associated with a material and is affected by *n* dynamic lights. The material describes the texture(s) used by the primitive, what device state settings need to be in force, and which vertex and pixel shader to use when rendering the primitive. The lights determine how dynamic lighting calculations will be applied to the primitive. Lighting and shading is a complex topic. We'll discuss the fundamentals in Chapter 11, but these topics are covered in depth in many excellent books on computer graphics, including [16], [49] and [2].

1.6.8.2 Scene Graph/Culling Optimizations

The low-level renderer draws all of the geometry submitted to it, without much regard for whether or not that geometry is actually visible (other than back-face culling and clipping triangles to the camera frustum). A higher-level component is usually needed in order to limit the number of primitives submitted for rendering, based on some form of visibility determination. This layer is shown in Figure 1.22.

For very small game worlds, a simple *frustum cull* (i.e., removing objects that the camera cannot "see") is probably all that is required. For larger game worlds, a more advanced *spatial subdivision* data structure might be used to improve rendering efficiency by allowing the potentially visible set (PVS) of objects to be determined very quickly. Spatial subdivisions can take many forms, including a binary space partitioning tree, a quadtree, an octree, a *k*d-tree or a sphere hierarchy. A spatial subdivision is sometimes called a scene graph, although technically the latter is a particular kind of data structure and does not subsume the former. Portals or occlusion culling methods might also be applied in this layer of the rendering engine.

Ideally, the low-level renderer should be completely agnostic to the type of spatial subdivision or scene graph being used. This permits different game

图 1.23。视觉效果。

团队可以重复使用原始提交代码，但需要根据每个团队的游戏需求构建一个PVS判定系统。OGRE开源渲染引擎（http://www.ogre3d.org）的设计就是一个很好的例子。OGRE提供了一个即插即用的场景图架构。游戏开发者可以从众多预实现的场景图设计中进行选择，也可以提供自定义的场景图实现。

1.6.8.3 视觉效果

现代游戏引擎支持各种各样的视觉效果，如图1.23所示，包括：

- 粒子系统（用于烟雾、火焰、水花等）；
- 贴花系统（用于弹孔、脚印等）；
- 光照映射和环境映射；
- 动态阴影；以及
- 全屏后期效果，在 3D 场景渲染到屏幕外缓冲区后应用。

全屏后期效果的一些示例包括：

- 高动态范围（HDR）色调映射和光晕；
- 全屏抗锯齿（FSAA）；以及
- 色彩校正和色彩转换效果，包括漂白旁路、饱和度和去饱和度效果等。

游戏引擎通常会有一个特效系统组件，用于管理粒子、贴花和其他视觉效果的专门渲染需求。粒子和贴花系统通常是渲染引擎的不同组件，并作为低级渲染器的输入。另一方面

Figure 1.23. Visual effects.

teams to reuse the primitive submission code but to craft a PVS determination system that is specific to the needs of each team's game. The design of the OGRE open source rendering engine (http://www.ogre3d.org) is a great example of this principle in action. OGRE provides a plug-and-play scene graph architecture. Game developers can either select from a number of preimplemented scene graph designs, or they can provide a custom scene graph implementation.

1.6.8.3 Visual Effects

Modern game engines support a wide range of visual effects, as shown in Figure 1.23, including:

- particle systems (for smoke, fire, water splashes, etc.);
- decal systems (for bullet holes, foot prints, etc.);
- light mapping and environment mapping;
- dynamic shadows; and
- full-screen post effects, applied after the 3D scene has been rendered to an off-screen buffer.

Some examples of full-screen post effects include:

- high dynamic range (HDR) tone mapping and bloom;
- full-screen anti-aliasing (FSAA); and
- color correction and color-shift effects, including bleach bypass, saturation and desaturation effects, etc.

It is common for a game engine to have an *effects system* component that manages the specialized rendering needs of particles, decals and other visual effects. The particle and decal systems are usually distinct components of the rendering engine and act as inputs to the low-level renderer. On the other

图 1.24.前端图形。

手动、光照映射、环境映射和阴影通常在渲染引擎内部处理。全屏后期效果要么作为渲染器的一个组成部分实现，要么作为在渲染器输出缓冲区上运行的独立组件实现。

1.6.8.4 前端

大多数游戏都会将某种 2D 图形叠加在 3D 场景上，以实现各种目的。这些目的包括：

- 游戏的平视显示器（HUD）；
- 游戏内菜单、控制台和/或其他开发工具，可能与最终产品一起提供，也可能不提供；以及
- 可能是游戏中的图形用户界面 (GUI)，允许玩家操纵他或她的角色的库存、配置战斗单位或执行其他复杂的游戏内任务。

该层如图 1.24 所示。这类二维图形通常通过绘制带纹理的四边形（三角形对）并使用正交投影来实现。或者，它们也可以以全 3D 形式渲染，使四边形始终面向相机。

我们还在此层中加入了全动态视频 (FMV) 系统。该系统负责播放之前录制的全屏影片（使用游戏的渲染引擎或其他渲染包渲染）。

一个相关的系统是游戏内过场动画 (IGC) 系统。该组件通常允许在游戏内部以全 3D 形式编排过场动画序列。例如，当玩家穿过城市时，两个关键角色之间的对话可能会被实现为游戏内过场动画。IGC 可能会包含或不包含玩家角色。它们可以以玩家无法控制的特意切换的方式呈现，也可以巧妙地融入游戏中，甚至人类玩家都没有意识到。

Figure 1.24. Front end graphics.

hand, light mapping, environment mapping and shadows are usually handled internally within the rendering engine proper. Full-screen post effects are either implemented as an integral part of the renderer or as a separate component that operates on the renderer's output buffers.

1.6.8.4 Front End

Most games employ some kind of 2D graphics overlaid on the 3D scene for various purposes. These include:

- the game's *heads-up display* (HUD);
- in-game menus, a console and/or other *development tools*, which may or may not be shipped with the final product; and
- possibly an in-game *graphical user interface* (GUI), allowing the player to manipulate his or her character's inventory, configure units for battle or perform other complex in-game tasks.

This layer is shown in Figure 1.24. Two-dimensional graphics like these are usually implemented by drawing textured quads (pairs of triangles) with an orthographic projection. Or they may be rendered in full 3D, with the quads bill-boarded so they always face the camera.

We've also included the *full-motion video* (FMV) system in this layer. This system is responsible for playing full-screen movies that have been recorded earlier (either rendered with the game's rendering engine or using another rendering package).

A related system is the *in-game cinematics* (IGC) system. This component typically allows cinematic sequences to be choreographed within the game itself, in full 3D. For example, as the player walks through a city, a conversation between two key characters might be implemented as an in-game cinematic. IGCs may or may not include the player character(s). They may be done as a deliberate cut-away during which the player has no control, or they may be subtly integrated into the game without the human player even realizing that

正在进行 IGC。一些游戏，例如顽皮狗的《神秘海域 4：盗贼末路》，已经完全放弃了预渲染电影，而是将游戏中的所有过场动画都显示为实时 IGC。

图 1.25。分析和调试工具。

1.6.9 分析和调试工具

游戏是实时系统，因此游戏工程师通常需要分析游戏性能以优化性能。此外，内存资源通常很稀缺，因此开发人员也会大量使用内存分析工具。如图 1.25 所示，性能分析和调试层不仅包含这些工具，还包含游戏内调试功能，例如调试绘图、游戏内菜单系统或控制台，以及录制和回放游戏过程以进行测试和调试的功能。

有很多优秀的通用软件分析工具可用，包括：

- 英特尔的 VTune，
- IBM 的 Quantify 和 Purify（PurifyPlus 工具套件的一部分），
- Parasoft 的 Insure++，以及
- 由 Julian Seward 和 Valgrind 开发团队开发的 Valgrind。

然而，大多数游戏引擎还包含一套自定义的分析和调试工具。例如，它们可能包含以下一项或多项：

- 一种手动检测代码的机制，以便可以对代码的特定部分进行计时；
- 游戏运行时在屏幕上显示分析统计数据的功能；
- 将性能统计数据转储到文本文件或 Excel 电子表格的工具；
- 用于确定引擎和每个子系统使用了多少内存的工具，包括各种屏幕显示；
- 在游戏终止和/或游戏过程中转储内存使用情况、高水位和泄漏统计数据的能力；
- 允许在整个代码中插入调试打印语句的工具，以及打开或关闭不同类别的调试输出和控制输出详细程度的能力；以及

an IGC is taking place. Some games, such as Naughty Dog's *Uncharted 4: A Thief's End*, have moved away from pre-rendered movies entirely, and display *all* cinematic moments in the game as real-time IGCs.

Figure 1.25. Profiling and debugging tools.

1.6.9 Profiling and Debugging Tools

Games are real-time systems and, as such, game engineers often need to profile the performance of their games in order to optimize performance. In addition, memory resources are usually scarce, so developers make heavy use of memory analysis tools as well. The profiling and debugging layer, shown in Figure 1.25, encompasses these tools and also includes in-game debugging facilities, such as debug drawing, an in-game menu system or console and the ability to record and play back gameplay for testing and debugging purposes.

There are plenty of good general-purpose software profiling tools available, including:

- Intel's *VTune*,
- IBM's *Quantify* and *Purify* (part of the *PurifyPlus* tool suite),
- *Insure++* by Parasoft, and
- *Valgrind* by Julian Seward and the Valgrind development team.

However, most game engines also incorporate a suite of custom profiling and debugging tools. For example, they might include one or more of the following:

- a mechanism for manually instrumenting the code, so that specific sections of code can be timed;
- a facility for displaying the profiling statistics on-screen while the game is running;
- a facility for dumping performance stats to a text file or to an Excel spreadsheet;
- a facility for determining how much memory is being used by the engine, and by each subsystem, including various on-screen displays;
- the ability to dump memory usage, high water mark and leakage stats when the game terminates and/or during gameplay;
- tools that allow debug print statements to be peppered throughout the code, along with an ability to turn on or off different categories of debug output and control the level of verbosity of the output; and

图 1.26.碰撞和物理子系统。

- 能够录制游戏事件并进行回放。这很难做到完美，但如果操作得当，它可以成为追踪漏洞的非常有价值的工具。

PlayStation 4 提供了强大的核心转储功能，可帮助程序员调试崩溃问题。PlayStation 4 会持续录制最后 15 秒的游戏视频，以便玩家通过控制器上的“分享”按钮分享游戏体验。正因如此，PS4 的核心转储功能不仅会自动为程序员提供程序崩溃时的完整调用堆栈，还会提供崩溃瞬间的屏幕截图以及崩溃前 15 秒的视频片段。即使游戏已经发布，核心转储也可以在游戏崩溃时自动上传到游戏开发者的服务器。这些功能彻底改变了崩溃分析和修复的工作方式。

1.6.10 碰撞和物理

碰撞检测对每款游戏都至关重要。如果没有它，物体就会相互穿透，游戏就无法以任何合理的方式与虚拟世界互动。有些游戏还包含逼真或半逼真的动力学模拟。在游戏行业中，我们称之为“物理系统”，尽管“刚体动力学”这个术语实际上更为贴切，因为我们通常只关注刚体的运动（运动学）以及导致这种运动发生的力和扭矩（动力学）。该层如图 1.26 所示。

Figure 1.26. Collision and physics subsystem.

- the ability to record game events and then play them back. This is tough to get right, but when done properly it can be a very valuable tool for tracking down bugs.

The PlayStation 4 provides a powerful core dump facility to aid programmers in debugging crashes. The PlayStation 4 is always recording the last 15 seconds of gameplay video, to allow players to share their experiences via the Share button on the controller. Because of this, the PS4's core dump facility automatically provides programmers not only with a complete call stack of what the program was doing when it crashed, but also with a screenshot of the moment of the crash and 15 seconds of video footage showing what was happening just prior to the crash. Core dumps can be automatically uploaded to the game developer's servers whenever the game crashes, even after the game has shipped. These facilities revolutionize the tasks of crash analysis and repair.

1.6.10 Collision and Physics

Collision detection is important for every game. Without it, objects would interpenetrate, and it would be impossible to interact with the virtual world in any reasonable way. Some games also include a realistic or semi-realistic dynamics simulation. We call this the "physics system" in the game industry, although the term *rigid body dynamics* is really more appropriate, because we are usually only concerned with the motion (kinematics) of rigid bodies and the forces and torques (dynamics) that cause this motion to occur. This layer is depicted in Figure 1.26.

碰撞和物理通常紧密耦合。这是因为当检测到碰撞时，它们几乎总是作为物理集成和约束满足逻辑的一部分进行解析。如今，很少有游戏公司会编写自己的碰撞/物理引擎。相反，通常会将第三方 SDK 集成到引擎中。

- Havok 是当今业界的黄金标准。它功能丰富，性能全面卓越。
- NVIDIA 的 PhysX 是另一个出色的碰撞和动态引擎。它被集成到虚幻引擎 4 中，也可作为独立产品免费用于 PC 游戏开发。PhysX 最初设计为 Ageia 物理加速器芯片的接口。该 SDK 目前归 NVIDIA 所有并由 NVIDIA 发行，该公司已将 PhysX 适配到其最新的 GPU 上。

开源物理和碰撞引擎也已推出。其中最著名的或许是开放动力学引擎 (ODE)。更多信息，请访问 http://www.ode.org。I-Collide、V-Collide 和 RAPID 是其他一些流行的非商业碰撞检测引擎。这三个引擎均由北卡罗来纳大学 (UNC) 开发。更多信息，请访问 http://www.cs.unc.edu/~geom/I_COLLIDE/index.html 和 http://www.cs.unc。

edu/□geom/V_COLLIDE/index.html。

1.6.11 动画

任何包含有机或半有机角色（人类、动物、卡通人物甚至机器人）的游戏都需要动画系统。游戏中使用的动画基本类型有五种：

- 精灵/纹理动画，
- 刚体层次动画，
- 骨骼动画，
- 顶点动画，以及
- 变形目标。

骨骼动画允许动画师使用相对简单的骨骼系统来设置精细的 3D 角色网格的姿势。当骨骼移动时，3D 网格的顶点也会随之移动。虽然某些引擎也使用变形目标和顶点动画，但骨骼动画是当今游戏中最流行的动画方法；因此，本书将主要关注骨骼动画。典型的骨骼动画系统如图 1.27 所示。

Collision and physics are usually quite tightly coupled. This is because when collisions are detected, they are almost always resolved as part of the physics integration and constraint satisfaction logic. Nowadays, very few game companies write their own collision/physics engine. Instead, a third-party SDK is typically integrated into the engine.

- *Havok* is the gold standard in the industry today. It is feature-rich and performs well across the boards.
- *PhysX* by NVIDIA is another excellent collision and dynamics engine. It was integrated into Unreal Engine 4 and is also available for free as a stand-alone product for PC game development. PhysX was originally designed as the interface to Ageia's physics accelerator chip. The SDK is now owned and distributed by NVIDIA, and the company has adapted PhysX to run on its latest GPUs.

Open source physics and collision engines are also available. Perhaps the best-known of these is the Open Dynamics Engine (ODE). For more information, see http://www.ode.org. I-Collide, V-Collide and RAPID are other popular non-commercial collision detection engines. All three were developed at the University of North Carolina (UNC). For more information, see http://www.cs.unc.edu/~geom/I_COLLIDE/index.html and http://www.cs.unc.edu/~geom/V_COLLIDE/index.html.

1.6.11 Animation

Any game that has organic or semi-organic characters (humans, animals, cartoon characters or even robots) needs an animation system. There are five basic types of animation used in games:

- sprite/texture animation,
- rigid body hierarchy animation,
- skeletal animation,
- vertex animation, and
- morph targets.

Skeletal animation permits a detailed 3D character mesh to be posed by an animator using a relatively simple system of bones. As the bones move, the vertices of the 3D mesh move with them. Although morph targets and vertex animation are used in some engines, skeletal animation is the most prevalent animation method in games today; as such, it will be our primary focus in this book. A typical skeletal animation system is shown in Figure 1.27.

在图 1.16 中，您会注意到骨架网格渲染组件连接了渲染器和动画系统。两者之间有着紧密的协作，但接口定义得非常清晰。动画系统为骨架中的每个骨骼生成一个姿势，然后这些姿势以矩阵调色板的形式传递给渲染引擎。渲染器使用调色板中的一个或多个矩阵对每个顶点进行变换，以生成最终的混合顶点位置。这个过程被称为蒙皮。

使用布娃娃时，动画系统和物理系统之间也存在紧密的耦合。布娃娃是一个瘫软（通常是死亡的）的动画角色，其身体运动由物理系统模拟。物理系统将身体各个部位视为受约束的刚体系统，从而确定它们的位置和方向。动画系统计算渲染引擎所需的矩阵调色板，以便在屏幕上绘制角色。

1.6.12 人机接口设备（HID）

每个游戏都需要处理来自玩家的输入，这些输入来自各种人机界面设备 (HID)，包括：

- 键盘和鼠标，
- 游戏手柄，或
- 其他专用游戏控制器，如方向盘、钓鱼竿、跳舞毯、Wiimote 等。

我们有时将此组件称为播放器 I/O 组件，因为

图 1.28 播放器输入/输出系统，也称为人机接口设备 (HID) 层。

图 1.27。骨骼动画子系统。

You'll notice in Figure 1.16 that the skeletal mesh rendering component bridges the gap between the renderer and the animation system. There is a tight cooperation happening here, but the interface is very well defined. The animation system produces a pose for every bone in the skeleton, and then these poses are passed to the rendering engine as a palette of matrices. The renderer transforms each vertex by the matrix or matrices in the palette, in order to generate a final blended vertex position. This process is known as *skinning*.

There is also a tight coupling between the animation and physics systems when *rag dolls* are employed. A rag doll is a limp (often dead) animated character, whose bodily motion is simulated by the physics system. The physics system determines the positions and orientations of the various parts of the body by treating them as a constrained system of rigid bodies. The animation system calculates the palette of matrices required by the rendering engine in order to draw the character on-screen.

1.6.12 Human Interface Devices (HID)

Every game needs to process input from the player, obtained from various *human interface devices* (HIDs) including:

- the keyboard and mouse,
- a joypad, or
- other specialized game controllers, like steering wheels, fishing rods, dance pads, the Wiimote, etc.

We sometimes call this component the *player I/O* component, because

Figure 1.28. The player input/output system, also known as the human interface device (HID) layer.

Figure 1.27. Skeletal animation subsystem.

我们还可以通过 HID 向玩家提供输出，例如游戏手柄上的力反馈/震动，或者 Wiimote 产生的音频。典型的 HID 层如图 1.28 所示。

HID 引擎组件的架构有时会将特定硬件平台上游戏控制器的底层细节与高层游戏控制分离。它会处理来自硬件的原始数据，例如在每个游戏手柄摇杆中心点周围引入盲区、消除按键输入抖动、检测按键按下和弹起事件、解释和平滑加速度计输入（例如来自 PlayStation Dualshock 控制器的输入）等等。它通常提供一种机制，允许玩家自定义物理控制和逻辑游戏功能之间的映射。有时，它还会包含一个用于检测和弦（同时按下多个按钮）、序列（在一定时间限制内按顺序按下按钮）和手势（来自按钮、摇杆、加速度计等的输入序列）的系统。

1.6.13 音频

图 1.29。音频子系统。

在任何游戏引擎中，音频都和图形同等重要。然而，音频往往比渲染、物理、动画、AI 和游戏玩法更不受重视。举个例子：程序员经常在开发代码时关闭扬声器！（事实上，我认识不少游戏程序员，他们甚至连扬声器或耳机都没有。）然而，如果没有出色的音频引擎，任何优秀的游戏都称不上完美。音频层如图 1.29 所示。

音频引擎的复杂程度差异很大。Quake 的音频引擎非常基础，游戏团队通常会通过自定义功能对其进行增强，或将其替换为内部解决方案。虚幻引擎 4 提供了一个相当强大的 3D 音频渲染引擎（在 [45] 中有详细讨论），尽管它的功能有限，许多游戏团队可能希望对其进行增强和自定义，以提供高级的游戏专用功能。对于 DirectX 平台（PC、Xbox 360、Xbox One），微软提供了一个名为 XAudio2 的出色运行时音频引擎。美国艺电公司内部开发了一款名为 SoundR!OT 的先进高性能音频引擎。索尼互动娱乐 (SIE) 与顽皮狗等第一方工作室合作，提供了一个名为 Scream 的强大 3D 音频引擎，该引擎已用于多款 PS3 和 PS4 游戏，包括顽皮狗的《神秘海域 4：盗贼末路》和《最后生还者：重制版》。然而，即使游戏团队使用现有的音频引擎，每个游戏都需要大量的定制软件开发、集成工作、微调和对细节的关注，才能在最终产品中产生高质量的音频。

we may also provide *output* to the player through the HID, such as force-feedback/ rumble on a joypad or the audio produced by the Wiimote. A typical HID layer is shown in Figure 1.28.

The HID engine component is sometimes architected to divorce the low-level details of the game controller(s) on a particular hardware platform from the high-level game controls. It massages the raw data coming from the hardware, introducing a dead zone around the center point of each joypad stick, debouncing button-press inputs, detecting button-down and button-up events, interpreting and smoothing accelerometer inputs (e.g., from the PlayStation Dualshock controller) and more. It often provides a mechanism allowing the player to customize the mapping between physical controls and logical game functions. It sometimes also includes a system for detecting chords (multiple buttons pressed together), sequences (buttons pressed in sequence within a certain time limit) and gestures (sequences of inputs from the buttons, sticks, accelerometers, etc.).

1.6.13 Audio

Figure 1.29. Audio subsystem.

Audio is just as important as graphics in any game engine. Unfortunately, audio often gets less attention than rendering, physics, animation, AI and gameplay. Case in point: Programmers often develop their code with their speakers turned off! (In fact, I've known quite a few game programmers who didn't even *have* speakers or headphones.) Nonetheless, no great game is complete without a stunning audio engine. The audio layer is depicted in Figure 1.29.

Audio engines vary greatly in sophistication. Quake's audio engine is pretty basic, and game teams usually augment it with custom functionality or replace it with an in-house solution. Unreal Engine 4 provides a reasonably robust 3D audio rendering engine (discussed in detail in [45]), although its feature set is limited and many game teams will probably want to augment and customize it to provide advanced game-specific features. For DirectX platforms (PC, Xbox 360, Xbox One), Microsoft provides an excellent runtime audio engine called XAudio2. Electronic Arts has developed an advanced, high-powered audio engine internally called SoundR!OT. In conjunction with first-party studios like Naughty Dog, Sony Interactive Entertainment (SIE) provides a powerful 3D audio engine called Scream, which has been used on a number of PS3 and PS4 titles including Naughty Dog's *Uncharted 4: A Thief's End* and *The Last of Us: Remastered*. However, even if a game team uses a preexisting audio engine, every game requires a great deal of custom software development, integration work, fine-tuning and attention to detail in order to produce high-quality audio in the final product.

1.6.14 在线多人游戏/网络

许多游戏允许多名人类玩家在同一个虚拟世界中游戏。多人游戏至少有四种基本类型：

• 单屏多人游戏。两个或多个人机界面设备（游戏手柄、键盘、鼠标等）连接到一台街机、PC 或游戏主机。多个玩家角色居住在一个虚拟世界中，单个摄像头将所有玩家角色同时保持在画面中。这种多人游戏的例子包括《任天堂明星大乱斗》、《乐高星球大战》和《圣铠传说》。

• 分屏多人游戏。多个玩家角色居住在一个虚拟世界中，多个 HID 连接到一台游戏机，但每个 HID 都有自己的摄像头，屏幕被分成几个部分，以便每个玩家都可以看到自己的角色。

• 联网多人游戏。多台计算机或控制台联网在一起，每台机器承载一名玩家。

• 大型多人在线游戏 (MMOG)。实际上，数十万用户可以同时在一个由强大的中央服务器组成的庞大、持久的在线虚拟世界中玩游戏。

多层网络层如图1.30所示。

多人游戏在很多方面与单人游戏非常相似。然而，对多玩家的支持会对某些游戏引擎组件的设计产生深远的影响。游戏世界对象模型、渲染器、人机输入设备系统、玩家控制系统和动画系统都会受到影响。将多人游戏功能改造到现有的单人游戏引擎中并非不可能，尽管这可能是一项艰巨的任务。尽管如此，许多游戏团队已经成功做到了这一点。话虽如此，如果条件允许，通常最好从一开始就设计多人游戏功能。

图 1.30. 在线网络多人游戏工作子系统。

有趣的是，反过来，将多人游戏转换为单人游戏通常很容易。事实上，许多游戏引擎将单人模式视为多人游戏的一个特例，其中恰好只有一名玩家。Quake 引擎以其“客户端+服务器”模式而闻名，在这种模式下，在单人战役中，运行在一台 PC 上的单个可执行文件既充当客户端，又充当服务器。

1.6.15 游戏基础系统

游戏玩法指的是游戏中发生的动作、游戏发生的虚拟世界的规则、玩家的能力

1.6.14 Online Multiplayer/Networking

Many games permit multiple human players to play within a single virtual world. Multiplayer games come in at least four basic flavors:

- *Single-screen multiplayer*. Two or more human interface devices (joypads, keyboards, mice, etc.) are connected to a single arcade machine, PC or console. Multiple player characters inhabit a single virtual world, and a single camera keeps all player characters in frame simultaneously. Examples of this style of multiplayer gaming include *Smash Brothers*, *Lego Star Wars* and *Gauntlet*.
- *Split-screen multiplayer*. Multiple player characters inhabit a single virtual world, with multiple HIDs attached to a single game machine, but each with its own camera, and the screen is divided into sections so that each player can view his or her character.
- *Networked multiplayer*. Multiple computers or consoles are networked together, with each machine hosting one of the players.
- *Massively multiplayer online games (MMOG)*. Literally hundreds of thousands of users can be playing simultaneously within a giant, persistent, online virtual world hosted by a powerful battery of central servers.

The multiplayer networking layer is shown in Figure 1.30.

Multiplayer games are quite similar in many ways to their single-player counterparts. However, support for multiple players can have a profound impact on the design of certain game engine components. The game world object model, renderer, human input device system, player control system and animation systems are all affected. Retrofitting multiplayer features into a pre-existing single-player engine is certainly not impossible, although it can be a daunting task. Still, many game teams have done it successfully. That said, it is usually better to design multiplayer features from day one, if you have that luxury.

Figure 1.30. Online multiplayer networking subsystem.

It is interesting to note that going the other way—converting a multiplayer game into a single-player game—is typically trivial. In fact, many game engines treat single-player mode as a special case of a multiplayer game, in which there happens to be only one player. The Quake engine is well known for its *client-on-top-of-server* mode, in which a single executable, running on a single PC, acts both as the client and the server in single-player campaigns.

1.6.15 Gameplay Foundation Systems

The term *gameplay* refers to the action that takes place in the game, the rules that govern the virtual world in which the game takes place, the abilities of the

图 1.31.游戏基础系统。

玩家角色（称为玩家机制）以及世界中其他角色和物体的机制，以及玩家的目标和目的。游戏玩法通常使用引擎其余部分所使用的原生语言或高级脚本语言（有时两者兼而有之）实现。为了弥合游戏玩法代码与我们迄今为止讨论过的低级引擎系统之间的差距，大多数游戏引擎引入了一个层，我将其称为游戏玩法基础层（由于缺乏标准化名称）。如图 1.31 所示，该层提供了一套核心功能，可方便地在此基础上实现特定于游戏的逻辑。

1.6.15.1 游戏世界和对象模型

游戏基础层引入了游戏世界的概念，其中包含静态和动态元素。游戏世界的内容通常以面向对象的方式建模（通常（但并非总是）使用面向对象编程语言）。在本书中，构成游戏的对象类型集合称为游戏对象模型。游戏对象模型提供了对虚拟游戏世界中异构对象集合的实时模拟。

典型的游戏对象类型包括：

- 静态背景几何，如建筑物、道路、地形（通常是特殊情况）等；
- 动态刚体，例如岩石、汽水罐、椅子等；
- 玩家角色（PC）；
- 非玩家角色（NPC）；

Figure 1.31. Gameplay foundation systems.

player character(s) (known as *player mechanics*) and of the other characters and objects in the world, and the goals and objectives of the player(s). Gameplay is typically implemented either in the native language in which the rest of the engine is written or in a high-level scripting language—or sometimes both. To bridge the gap between the gameplay code and the low-level engine systems that we've discussed thus far, most game engines introduce a layer that I'll call the *gameplay foundations* layer (for lack of a standardized name). Shown in Figure 1.31, this layer provides a suite of core facilities, upon which game-specific logic can be implemented conveniently.

1.6.15.1 Game Worlds and Object Models

The gameplay foundations layer introduces the notion of a game world, containing both static and dynamic elements. The contents of the world are usually modeled in an object-oriented manner (often, but not always, using an object-oriented programming language). In this book, the collection of object types that make up a game is called the *game object model*. The game object model provides a real-time simulation of a heterogeneous collection of objects in the virtual game world.

Typical types of game objects include:

- static background geometry, like buildings, roads, terrain (often a special case), etc.;
- dynamic rigid bodies, such as rocks, soda cans, chairs, etc.;
- player characters (PC);
- non-player characters (NPC);

- 武器；
- 抛射物；
- 车辆；
- 灯光（可能在运行时出现在动态场景中，或者仅用于离线静态照明）；

- 相机；等等。

游戏世界模型与软件对象模型紧密相关，并且该模型最终会渗透到整个引擎中。“软件对象模型”是指用于实现面向对象软件的一组语言特性、策略和约定。在游戏引擎的语境中，软件对象模型可以回答以下问题：

- 您的游戏引擎是否采用面向对象的方式设计？
- 您将使用什么语言？C？C++？Java？OCaml？
- 静态类层次结构该如何组织？一个巨大的整体层次结构？还是一堆松散耦合的组件？
- 您会使用模板和基于策略的设计，还是传统的多态性？
- 对象是如何引用的？直接使用旧指针？还是智能指针？把手？
- 如何唯一标识对象？仅通过内存中的地址？按名称？按全局唯一标识符（GUID）？
- 如何管理游戏对象的生命周期？
- 如何模拟游戏对象随时间的状态？

我们将在第 16.2 节深入探讨软件对象模型和游戏对象模型。

1.6.15.2 事件系统

游戏对象总是需要相互通信。这可以通过各种方式实现。例如，发送消息的对象可能只是调用接收对象的成员函数。事件驱动架构与典型的图形用户界面非常相似，也是对象间通信的常用方法。在事件驱动系统中，发送方会创建一个称为事件或消息的小数据结构，其中包含消息的类型和要发送的任何参数数据。通过调用其事件处理程序函数将事件传递给接收对象。事件也可以存储在队列中以供将来某个时间处理。

- weapons;
- projectiles;
- vehicles;
- lights (which may be present in the dynamic scene at runtime, or only used for static lighting offline);
- cameras;

and the list goes on.

The game world model is intimately tied to a *software object model*, and this model can end up pervading the entire engine. The term software object model refers to the set of language features, policies and conventions used to implement a piece of object-oriented software. In the context of game engines, the software object model answers questions, such as:

- Is your game engine designed in an object-oriented manner?
- What language will you use? C? C++? Java? OCaml?
- How will the static class hierarchy be organized? One giant monolithic hierarchy? Lots of loosely coupled components?
- Will you use templates and policy-based design, or traditional polymorphism?
- How are objects referenced? Straight old pointers? Smart pointers? Handles?
- How will objects be uniquely identified? By address in memory only? By name? By a global unique identifier (GUID)?
- How are the lifetimes of game objects managed?
- How are the states of the game objects simulated over time?

We'll explore software object models and game object models in considerable depth in Section 16.2.

1.6.15.2 Event System

Game objects invariably need to communicate with one another. This can be accomplished in all sorts of ways. For example, the object sending the message might simply call a member function of the receiver object. An event-driven architecture, much like what one would find in a typical graphical user interface, is also a common approach to inter-object communication. In an event-driven system, the sender creates a little data structure called an *event* or *message*, containing the message's type and any argument data that are to be sent. The event is passed to the receiver object by calling its *event handler function*. Events can also be stored in a queue for handling at some future time.

1.6.15.3 脚本系统

许多游戏引擎都使用脚本语言，以便更轻松、更快速地开发特定于游戏的游戏规则和内容。如果没有脚本语言，每次更改引擎中使用的逻辑或数据结构时，您都必须重新编译并重新链接游戏可执行文件。但是，当脚本语言集成到引擎中时，可以通过修改和重新加载脚本代码来更改游戏逻辑和数据。有些引擎允许在游戏继续运行时重新加载脚本。其他引擎则要求在重新编译脚本之前关闭游戏。但无论哪种方式，周转时间仍然比重新编译和重新链接游戏可执行文件要快得多。

1.6.15.4 人工智能基础

传统上，人工智能完全属于游戏专用软件的范畴——它通常不被视为游戏引擎本身的一部分。然而，近年来，游戏公司已经意识到几乎每个人工智能系统都存在一些模式，这些基础正逐渐被纳入游戏引擎的范畴。

例如，一家名为 Kynogon 的公司开发了一个名为 Kynapse 的中间件 SDK，它提供了构建商业化游戏 AI 所需的大部分底层技术。这项技术已被 Autodesk 收购，并已被一款名为 Gameware Navigation 的全新 AI 中间件包所取代，该包由开发 Kynapse 的同一工程团队设计。该 SDK 提供了底层 AI 构建模块，例如导航网格生成、路径查找、静态和动态物体避让、游戏空间内漏洞识别（例如，可能存在伏击的敞开的窗户）以及 AI 与动画之间定义明确的接口。

1.6.16 游戏特定子系统

在游戏玩法基础层和其他底层引擎组件之上，游戏玩法程序员和设计师协作实现游戏本身的功能。游戏玩法系统通常数量众多、高度多样化，并且特定于正在开发的游戏。如图 1.32 所示，这些系统包括但不限于玩家角色的机制、各种游戏内摄像头系统、用于控制非玩家角色的人工智能、武器系统、车辆等等。如果可以在引擎和游戏之间划一条清晰的界线，

1.6.15.3 Scripting System

Many game engines employ a scripting language in order to make development of game-specific gameplay rules and content easier and more rapid. Without a scripting language, you must recompile and relink your game executable every time a change is made to the logic or data structures used in the engine. But when a scripting language is integrated into your engine, changes to game logic and data can be made by modifying and reloading the script code. Some engines allow script to be reloaded while the game continues to run. Other engines require the game to be shut down prior to script recompilation. But either way, the turnaround time is still much faster than it would be if you had to recompile and relink the game's executable.

1.6.15.4 Artificial Intelligence Foundations

Traditionally, artificial intelligence has fallen squarely into the realm of game-specific software—it was usually not considered part of the game engine per se. More recently, however, game companies have recognized patterns that arise in almost every AI system, and these foundations are slowly starting to fall under the purview of the engine proper.

For example, a company called Kynogon developed a middleware SDK named Kynapse, which provides much of the low-level technology required to build commercially viable game AI. This technology was purchased by Autodesk and has been superseded by a totally redesigned AI middleware package called Gameware Navigation, designed by the same engineering team that invented Kynapse. This SDK provides low-level AI building blocks such as nav mesh generation, path finding, static and dynamic object avoidance, identification of vulnerabilities within a play space (e.g., an open window from which an ambush could come) and a well-defined interface between AI and animation.

1.6.16 Game-Specific Subsystems

On top of the gameplay foundation layer and the other low-level engine components, gameplay programmers and designers cooperate to implement the features of the game itself. Gameplay systems are usually numerous, highly varied and specific to the game being developed. As shown in Figure 1.32, these systems include, but are certainly not limited to the mechanics of the player character, various in-game camera systems, artificial intelligence for the control of non-player characters, weapon systems, vehicles and the list goes on. If a clear line could be drawn between the engine and the game,

它位于游戏特定子系统和游戏玩法基础层之间。实际上，这条界线从来都不是完全清晰的。至少有一些游戏特定的知识总是会渗透到游戏玩法基础层，有时甚至延伸到引擎本身的核心。

1.7 工具和资产管道

任何游戏引擎都必须接收大量数据，这些数据以游戏资源、配置文件、脚本等等的形式存在。图 1.33 描绘了现代游戏引擎中常见的一些游戏资源类型。较粗的深灰色箭头表示数据如何从用于创建原始资源的工具一路流向游戏引擎本身。较细的浅灰色箭头表示不同类型的资源如何引用或使用其他资源。

1.7.1 数字内容创作工具

游戏本质上是多媒体应用程序。游戏引擎的输入数据形式多样，从 3D 网格数据到纹理位图、动画数据到音频文件。所有这些源数据都必须由艺术家创建和处理。艺术家使用的工具称为数字内容创作 (DCC) 应用程序。

DCC 应用程序通常专注于创建一种特定类型的数据，尽管有些工具可以生成多种数据类型。例如，Autodesk 的 Maya 和 3ds Max 以及 Pixologic 的 ZBrush 在 3D 网格和动画数据的创建中非常流行。Adobe 的 Photoshop 及其同类软件则致力于创建和编辑位图（纹理）。SoundForge 是一款流行的音频剪辑创建工具。某些类型的游戏数据无法使用现成的 DCC 应用程序创建。例如，大多数游戏引擎都提供了自定义编辑器来布局游戏世界。不过，有些引擎确实会使用

图 1.32. 游戏特定的子系统。

it would lie between the game-specific subsystems and the gameplay foundations layer. Practically speaking, this line is never perfectly distinct. At least some game-specific knowledge invariably seeps down through the gameplay foundations layer and sometimes even extends into the core of the engine itself.

1.7 Tools and the Asset Pipeline

Any game engine must be fed a great deal of data, in the form of game assets, configuration files, scripts and so on. Figure 1.33 depicts some of the types of game assets typically found in modern game engines. The thicker dark-grey arrows show how data flows from the tools used to create the original source assets all the way through to the game engine itself. The thinner light-grey arrows show how the various types of assets refer to or use other assets.

1.7.1 Digital Content Creation Tools

Games are multimedia applications by nature. A game engine's input data comes in a wide variety of forms, from 3D mesh data to texture bitmaps to animation data to audio files. All of this source data must be created and manipulated by artists. The tools that the artists use are called *digital content creation* (DCC) applications.

A DCC application is usually targeted at the creation of one particular type of data—although some tools can produce multiple data types. For example, Autodesk's Maya and 3ds Max and Pixologic's ZBrush are prevalent in the creation of both 3D meshes and animation data. Adobe's Photoshop and its ilk are aimed at creating and editing bitmaps (textures). SoundForge is a popular tool for creating audio clips. Some types of game data cannot be created using an off-the-shelf DCC app. For example, most game engines provide a custom editor for laying out game worlds. Still, some engines do make use of

Figure 1.32. Game-specific subsystems.

图 1.33。工具和资产管道。

现有的游戏世界布局工具。我见过一些游戏团队使用 3ds Max 或 Maya 作为世界布局工具，有时还会使用自定义插件来辅助用户。问问大多数游戏开发者，他们都会告诉你，他们还记得自己曾经使用简单的位图编辑器布局地形高度字段，或者直接手动将世界布局输入文本文件。工具不必美观——游戏团队会使用任何可用的工具来完成工作。话虽如此，如果游戏团队想要及时开发出一款高度精致的产品，工具必须相对易于使用，而且绝对必须可靠。

Figure 1.33. Tools and the asset pipeline.

preexisting tools for game world layout. I've seen game teams use 3ds Max or Maya as a world layout tool, with or without custom plug-ins to aid the user. Ask most game developers, and they'll tell you they can remember a time when they laid out terrain height fields using a simple bitmap editor, or typed world layouts directly into a text file by hand. Tools don't have to be pretty—game teams will use whatever tools are available and get the job done. That said, tools must be relatively *easy to use*, and they absolutely must be *reliable*, if a game team is going to be able to develop a highly polished product in a timely manner.

1.7.2 资产调节管道

数字内容创作 (DCC) 应用程序使用的数据格式很少适合直接在游戏中使用。主要有两个原因。

1. DCC 应用程序的内存数据模型通常比游戏引擎所需的复杂得多。例如，Maya 存储场景节点的有向无环图 (DAG)，并具有复杂的互连网络。它存储了对文件执行的所有编辑的历史记录。它将场景中每个对象的位置、方向和比例表示为完整的 3D 变换层次结构，分解为平移、旋转、缩放和剪切分量。游戏引擎通常只需要这些信息中的一小部分即可在游戏中渲染模型。

2. DCC 应用程序的文件格式在运行时读取速度通常太慢，并且在某些情况下它是一种封闭的专有格式。

因此，DCC 应用程序生成的数据通常会导出为更易于访问的标准化格式或自定义文件格式，以供游戏中使用。

数据从 DCC 应用导出后，通常需要进一步处理才能发送到游戏引擎。如果游戏工作室在多个平台上发布游戏，则中间文件可能会针对每个目标平台进行不同的处理。例如，3D 网格数据可能会导出为中间格式，例如 XML、JSON 或简单的二进制格式。然后，可能会对其进行处理，以合并使用相同材质的网格，或拆分引擎无法处理的过大网格。之后，网格数据可能会被组织并打包到适合在特定硬件平台上加载的内存映像中。

从 DCC 应用到游戏引擎的管道有时被称为资产调节管道 (ACP)。每个游戏引擎都以某种形式拥有这种管道。

1.7.2.1 3D 模型/网格数据

游戏中可见的几何体通常由三角形网格构成。一些老游戏也会使用被称为“画刷”的体积几何体。下文将简要讨论每种类型的几何数据。有关描述和渲染 3D 几何体的技术的深入讨论，请参阅第 11 章。

3D 模型（网格）

网格是由三角形和顶点组成的复杂形状。可渲染的几何体也可以由四边形或高阶细分构成

1.7.2 The Asset Conditioning Pipeline

The data formats used by digital content creation (DCC) applications are rarely suitable for direct use in-game. There are two primary reasons for this.

1. The DCC app's in-memory model of the data is usually much more complex than what the game engine requires. For example, Maya stores a directed acyclic graph (DAG) of scene nodes, with a complex web of interconnections. It stores a history of all the edits that have been performed on the file. It represents the position, orientation and scale of every object in the scene as a full hierarchy of 3D transformations, decomposed into translation, rotation, scale and shear components. A game engine typically only needs a tiny fraction of this information in order to render the model in-game.
2. The DCC application's file format is often too slow to read at runtime, and in some cases it is a closed proprietary format.

Therefore, the data produced by a DCC app is usually exported to a more accessible standardized format, or a custom file format, for use in-game.

Once data has been exported from the DCC app, it often must be further processed before being sent to the game engine. And if a game studio is shipping its game on more than one platform, the intermediate files might be processed differently for each target platform. For example, 3D mesh data might be exported to an intermediate format, such as XML, JSON or a simple binary format. Then it might be processed to combine meshes that use the same material, or split up meshes that are too large for the engine to digest. The mesh data might then be organized and packed into a memory image suitable for loading on a specific hardware platform.

The pipeline from DCC app to game engine is sometimes called the *asset conditioning pipeline* (ACP). Every game engine has this in some form.

1.7.2.1 3D Model/Mesh Data

The visible geometry you see in a game is typically constructed from triangle meshes. Some older games also make use of volumetric geometry known as *brushes*. We'll discuss each type of geometric data briefly below. For an in-depth discussion of the techniques used to describe and render 3D geometry, see Chapter 11.

3D Models (Meshes)

A mesh is a complex shape composed of triangles and vertices. Renderable geometry can also be constructed from quads or higher-order subdivision

表面。但是，当今的图形硬件几乎专门用于渲染光栅化三角形，因此所有形状最终都必须在渲染之前转换为三角形。

网格通常会应用一种或多种材质，以定义视觉表面属性（颜色、反射率、凹凸度、漫反射纹理等）。本书中，我将使用术语“网格”指代单个可渲染形状，使用“模型”指代可能包含多个网格以及动画数据和其他游戏元数据的复合对象。

网格通常在 3D 建模软件（例如 3ds Max、Maya 或 SoftImage）中创建。Pixologic 的一款强大且流行的工具 ZBrush 可以非常直观地构建超高分辨率网格，然后将其向下转换为具有法线贴图的低分辨率模型，以近似高频细节。

必须编写导出器来从数字内容创建 (DCC) 工具（例如 Maya、Max 等）中提取数据，并将其以引擎可处理的形式存储在磁盘上。DCC 应用提供了许多标准或半标准导出格式，但没有一种格式完全适合游戏开发（COLLADA 可能除外）。因此，游戏团队通常会创建自定义文件格式和自定义导出器来配合使用。

画笔几何

笔刷几何体被定义为凸包的集合，每个凸包由多个平面定义。笔刷通常直接在游戏世界编辑器中创建和编辑。这本质上是一种创建可渲染几何体的“老派”方法，但仍在某些引擎中使用。

优点：

- 快速且易于创建；
- 游戏设计师可以使用——通常用于“阻止”游戏级别以用于原型设计目的；
- 既可以作为碰撞体，也可以作为可渲染几何体。

缺点：

- 低分辨率；
- 难以创建复杂的形状；
- 无法支持铰接物体或动画角色。

1.7.2.2 骨骼动画数据

骨架网格体是一种特殊的网格体，它绑定到骨架层级结构，用于实现关节动画。这种网格体有时被称为

surfaces. But on today's graphics hardware, which is almost exclusively geared toward rendering rasterized triangles, all shapes must eventually be translated into triangles prior to rendering.

A mesh typically has one or more *materials* applied to it in order to define visual surface properties (color, reflectivity, bumpiness, diffuse texture, etc.). In this book, I will use the term "mesh" to refer to a single renderable shape, and "model" to refer to a composite object that may contain multiple meshes, plus animation data and other metadata for use by the game.

Meshes are typically created in a 3D modeling package such as 3ds Max, Maya or SoftImage. A powerful and popular tool by Pixologic called ZBrush allows ultra high-resolution meshes to be built in a very intuitive way and then down-converted into a lower-resolution model with normal maps to approximate the high-frequency detail.

Exporters must be written to extract the data from the digital content creation (DCC) tool (Maya, Max, etc.) and store it on disk in a form that is digestible by the engine. The DCC apps provide a host of standard or semi-standard export formats, although none are perfectly suited for game development (with the possible exception of COLLADA). Therefore, game teams often create custom file formats and custom exporters to go with them.

Brush Geometry

Brush geometry is defined as a collection of convex hulls, each of which is defined by multiple planes. Brushes are typically created and edited directly in the game world editor. This is essentially an "old school" approach to creating renderable geometry, but it is still used in some engines.

Pros:

- fast and easy to create;
- accessible to game designers—often used to "block out" a game level for prototyping purposes;
- can serve both as collision volumes and as renderable geometry.

Cons:

- low-resolution;
- difficult to create complex shapes;
- cannot support articulated objects or animated characters.

1.7.2.2 Skeletal Animation Data

A *skeletal mesh* is a special kind of mesh that is bound to a skeletal hierarchy for the purposes of articulated animation. Such a mesh is sometimes called a

皮肤，因为它构成了包裹不可见底层骨架的皮肤。骨架网格的每个顶点都包含一个索引列表，指示它绑定到骨架中的哪些关节。顶点通常还包含一组关节权重，指定每个关节对顶点的影响程度。

为了渲染骨架网格，游戏引擎需要三种不同类型的数据：

1. **网格本身，**

2. 骨骼层次结构（关节名称、父子关系以及骨骼最初绑定到网格时的基本姿势），以及

3. 一个或多个动画剪辑，指定关节随时间如何移动。

网格和骨架通常从 DCC 应用程序中导出为单个数据文件。但是，如果多个网格绑定到单个骨架，则最好将骨架导出为单独的文件。动画通常单独导出，以便只允许正在使用的动画在任何给定时间加载到内存中。然而，有些游戏引擎允许将一组动画导出为单个文件，有些甚至将网格、骨架和动画合并为一个整体文件。

未优化的骨骼动画由 4×3 矩阵采样流定义，这些采样以至少每秒 30 帧的频率采集骨骼中的每个关节（对于逼真的人形角色，骨骼关节可能多达 500 个或更多）。因此，动画数据本质上占用大量内存。因此，动画数据几乎总是以高度压缩的格式存储。压缩方案因引擎而异，有些是专有的。

对于游戏动画数据，没有一种标准化的格式。

1.7.2.3 音频数据

音频片段通常从 Sound Forge 或其他音频制作工具导出，格式多样，数据采样率也各有不同。音频文件可以是单声道、立体声、5.1、7.1 或其他多声道配置。Wave 文件 (.wav) 很常见，但其他文件格式，例如 PlayStation ADPCM 文件 (.vag) 也很常见。音频片段通常会被组织成库 (Bank)，以便于整理、轻松加载到引擎中以及进行流式传输。

skin because it forms the skin that surrounds the invisible underlying skeleton. Each vertex of a skeletal mesh contains a list of indices indicating to which joint(s) in the skeleton it is bound. A vertex usually also includes a set of joint weights, specifying the amount of influence each joint has on the vertex.

In order to render a skeletal mesh, the game engine requires three distinct kinds of data:

1. the mesh itself,

2. the skeletal hierarchy (joint names, parent-child relationships and the base pose the skeleton was in when it was originally bound to the mesh), and

3. one or more animation clips, which specify how the joints should move over time.

The mesh and skeleton are often exported from the DCC application as a single data file. However, if multiple meshes are bound to a single skeleton, then it is better to export the skeleton as a distinct file. The animations are usually exported individually, allowing only those animations which are in use to be loaded into memory at any given time. However, some game engines allow a bank of animations to be exported as a single file, and some even lump the mesh, skeleton and animations into one monolithic file.

An unoptimized skeletal animation is defined by a stream of 4×3 matrix samples, taken at a frequency of at least 30 frames per second, for each of the joints in a skeleton (of which there can be 500 or more for a realistic humanoid character). Thus, animation data is inherently memory-intensive. For this reason, animation data is almost always stored in a highly compressed format. Compression schemes vary from engine to engine, and some are proprietary. There is no one standardized format for game-ready animation data.

1.7.2.3 Audio Data

Audio clips are usually exported from Sound Forge or some other audio production tool in a variety of formats and at a number of different data sampling rates. Audio files may be in mono, stereo, 5.1, 7.1 or other multi-channel configurations. Wave files (.wav) are common, but other file formats such as PlayStation ADPCM files (.vag) are also commonplace. Audio clips are often organized into banks for the purposes of organization, easy loading into the engine, and streaming.

1.7.2.4 粒子系统数据

现代游戏运用复杂的粒子效果。这些效果由专门从事视觉效果创作的艺术家创作。第三方工具（例如 Houdini）可以创作出电影级的效果；然而，大多数游戏引擎无法渲染 Houdini 所能创建的全部效果。因此，许多游戏公司创建了自定义粒子效果编辑工具，该工具仅显示引擎实际支持的效果。自定义工具还可以让艺术家看到与游戏实际效果完全相同的效果。

1.7.3 世界编辑器

游戏世界是游戏引擎中所有元素汇集的地方。据我所知，目前还没有市面上可用的游戏世界编辑器（例如 Maya 或 Max 的游戏世界编辑器）。不过，许多市面上的游戏引擎提供了优秀的世界编辑器：

- 大多数基于 Quake 技术的游戏引擎都使用 Radiant 游戏编辑器的一些变体。
- Half-Life 2 Source 引擎提供了一个名为 Hammer 的世界编辑器。
- UnrealEd 是虚幻引擎的世界编辑器。这个强大的工具还可以作为引擎可以使用的所有数据类型的资源管理器。
- Sandbox 是 CRYENGINE 中的世界编辑器。

编写一个好的世界编辑器很难，但它是任何优秀游戏引擎极其重要的一部分。

1.7.4 资源数据库

游戏引擎处理各种各样的资源类型，从可渲染几何体到材质和纹理，从动画数据到音频。这些资源部分由艺术家使用 Maya、Photoshop 或 SoundForge 等工具生成的原始数据定义。然而，每个资源也包含大量的元数据。例如，当动画师在 Maya 中创作动画剪辑时，元数据会向资源调节管道，并最终向游戏引擎提供以下信息：

- 在运行时识别动画剪辑的唯一 ID。
- 源 Maya（.ma 或 .mb）文件的名称和目录路径。
- 帧范围——动画开始和结束的帧。
- 动画是否循环播放。

1.7.2.4 Particle Systems Data

Modern games make use of complex particle effects. These are authored by artists who specialize in the creation of visual effects. Third-party tools, such as Houdini, permit film-quality effects to be authored; however, most game engines are not capable of rendering the full gamut of effects that can be created with Houdini. For this reason, many game companies create a custom particle effect editing tool, which exposes only the effects that the engine actually supports. A custom tool might also let the artist see the effect exactly as it will appear in-game.

1.7.3 The World Editor

The game world is where everything in a game engine comes together. To my knowledge, there are no commercially available game world editors (i.e., the game world equivalent of Maya or Max). However, a number of commercially available game engines provide good world editors:

- Some variant of the *Radiant* game editor is used by most game engines based on Quake technology.
- The *Half-Life 2* Source engine provides a world editor called *Hammer*.
- *UnrealEd* is the Unreal Engine's world editor. This powerful tool also serves as the asset manager for all data types that the engine can consume.
- *Sandbox* is the world editor in CRYENGINE.

Writing a good world editor is difficult, but it is an extremely important part of any good game engine.

1.7.4 The Resource Database

Game engines deal with a wide range of asset types, from renderable geometry to materials and textures to animation data to audio. These assets are defined in part by the raw data produced by the artists when they use a tool like Maya, Photoshop or SoundForge. However, every asset also carries with it a great deal of *metadata*. For example, when an animator authors an animation clip in Maya, the metadata provides the asset conditioning pipeline, and ultimately the game engine, with the following information:

- A unique id that identifies the animation clip at runtime.
- The name and directory path of the source Maya (.ma or .mb) file.
- The *frame range*—on which frame the animation begins and ends.
- Whether or not the animation is intended to loop.

图 1.34.独立工具架构。

• 动画师选择的压缩技术和级别。（有些资源即使高度压缩也不会明显降低质量，而有些资源则需要较低压缩甚至不压缩才能在游戏中正常显示。）

每个游戏引擎都需要某种数据库来管理与游戏资源相关的所有元数据。该数据库可以使用真正的关系型数据库（例如 MySQL 或 Oracle）实现，也可以以文本文件集合的形式实现，并由版本控制系统（例如 Subversion、Perforce 或 Git）进行管理。在本书中，我们将这些元数据称为资源数据库。

无论资源数据库采用何种格式存储和管理，都必须提供某种用户界面，以便用户创作和编辑数据。在顽皮狗，我们为此目的用 C# 编写了一个名为 Builder 的自定义 GUI。有关 Builder 和其他一些资源数据库用户界面的更多信息，请参阅第 7.2.1.3 节。

1.7.5 一些工具架构方法

游戏引擎的工具套件可以以多种方式构建。有些工具可能是独立的软件，如图 1.34 所示。有些工具可能构建在运行时引擎使用的某些较低层之上，如图 1.35 所示。有些工具可能内置于游戏本身。

Figure 1.34. Stand-alone tools architecture.

- The animator's choice of compression technique and level. (Some assets can be highly compressed without noticeably degrading their quality, while others require less or no compression in order to look right in-game.)

Every game engine requires some kind of database to manage all of the metadata associated with the game's assets. This database might be implemented using an honest-to-goodness relational database such as MySQL or Oracle, or it might be implemented as a collection of text files, managed by a revision control system such as Subversion, Perforce or Git. We'll call this metadata the *resource database* in this book.

No matter in what format the resource database is stored and managed, some kind of user interface must be provided to allow users to author and edit the data. At Naughty Dog, we wrote a custom GUI in C# called Builder for this purpose. For more information on Builder and a few other resource database user interfaces, see Section 7.2.1.3.

1.7.5 Some Approaches to Tool Architecture

A game engine's tool suite may be architected in any number of ways. Some tools might be stand-alone pieces of software, as shown in Figure 1.34. Some tools may be built on top of some of the lower layers used by the runtime engine, as Figure 1.35 illustrates. Some tools might be built into the game itself.

图 1.35.基于与游戏共享的框架构建的工具。

例如，Quake 和基于虚幻引擎的游戏都拥有游戏内置控制台，允许开发者和“模组制作者”在游戏运行过程中输入调试和配置命令。最后，基于 Web 的用户界面在某些类型的工具中越来越受欢迎。

图 1.36.虚幻引擎的工具架构。

Figure 1.35. Tools built on a framework shared with the game.

For example, Quake- and Unreal-based games both boast an in-game console that permits developers and “modders” to type debugging and configuration commands while running the game. Finally, web-based user interfaces are becoming more and more popular for certain kinds of tools.

Figure 1.36. Unreal Engine’s tool architecture.

举一个有趣而独特的例子，虚幻的世界编辑器和资源管理器 UnrealEd 直接内置于运行时游戏引擎中。要运行编辑器，您需要使用命令行参数“editor”来运行游戏。这种独特的架构风格如图 1.36 所示。它允许工具完全访问引擎使用的所有数据结构，并避免了每个数据结构必须有两种表示形式的常见问题——一种用于运行时引擎，一种用于工具。这也意味着在编辑器中运行游戏非常快（因为游戏实际上已经在运行了）。实时游戏内编辑通常是一项非常棘手的功能，但当编辑器成为游戏的一部分时，开发起来相对容易。然而，像这样的引擎内编辑器设计也存在一些问题。例如，当引擎崩溃时，工具也会变得无法使用。因此，引擎和资源创建工具之间的紧密耦合往往会降低生产速度。

1.7.5.1 基于 Web 的用户界面

基于 Web 的用户界面正迅速成为某些游戏开发工具的标配。在顽皮狗，我们使用了许多基于 Web 的 UI。顽皮狗的本地化工具是我们本地化数据库的前端门户。Tasker 是所有顽皮狗员工在制作过程中用于创建、管理、安排、沟通和协作游戏开发任务的 Web 界面。一个名为 Connector 的 Web 界面也作为我们查看游戏引擎在运行时发出的各种调试信息流的窗口。游戏将其调试文本输出到各种命名通道，每个通道都与不同的引擎系统（动画、渲染、AI、声音等）相关联。这些数据流由轻量级 Redis 数据库收集。基于浏览器的 Connector 界面允许用户以便捷的方式查看和过滤这些信息。

基于 Web 的 UI 比独立的 GUI 应用程序具有诸多优势。首先，与使用 Java、C # 或 C++ 等语言编写的独立应用程序相比，Web 应用程序的开发和维护通常更轻松快捷。Web 应用程序无需特殊安装——用户只需一个兼容的 Web 浏览器即可。Web 界面的更新无需安装即可推送给用户——用户只需刷新或重启浏览器即可接收更新。Web 界面也迫使我们使用客户端-服务器架构来设计工具。这使我们能够将工具分发给更广泛的受众。例如，顽皮狗的本地化工具可直接提供给全球的外包合作伙伴。

As an interesting and unique example, Unreal's world editor and asset manager, UnrealEd, is built right into the runtime game engine. To run the editor, you run your game with a command-line argument of "editor." This unique architectural style is depicted in Figure 1.36. It permits the tools to have total access to the full range of data structures used by the engine and avoids a common problem of having to have two representations of every data structure—one for the runtime engine and one for the tools. It also means that running the game from within the editor is very fast (because the game is actually already running). Live in-game editing, a feature that is normally very tricky to implement, can be developed relatively easily when the editor is a part of the game. However, an in-engine editor design like this does have its share of problems. For example, when the engine is crashing, the tools become unusable as well. Hence a tight coupling between engine and asset creation tools can tend to slow down production.

1.7.5.1 Web-Based User Interfaces

Web-based user interfaces are quickly becoming the norm for certain kinds of game development tools. At Naughty Dog, we use a number of web-based UIs. Naughty Dog's localization tool serves as the front-end portal into our localization database. *Tasker* is the web-based interface used by all Naughty Dog employees to create, manage, schedule, communicate and collaborate on game development tasks during production. A web-based interface known as *Connector* also serves as our window into the various streams of debugging information that are emitted by the game engine at runtime. The game spits out its debug text into various named channels, each associated with a different engine system (animation, rendering, AI, sound, etc.) These data streams are collected by a lightweight Redis database. The browser-based Connector interface allows users to view and filter this information in a convenient way.

Web-based UIs offer a number of advantages over stand-alone GUI applications. For one thing, web apps are typically easier and faster to develop and maintain than a stand-alone app written in a language like Java, C# or C++. Web apps require no special installation—all the user needs is a compatible web browser. Updates to a web-based interface can be pushed out to the users without the need for an installation step—they need only refresh or restart their browser to receive the update. Web interfaces also force us to design our tools using a client-server architecture. This opens up the possibility of distributing our tools to a wider audience. For example, Naughty Dog's localization tool is available directly to outsourcing partners around the globe

为我们提供语言翻译服务。当然，独立工具仍然有其用武之地，尤其是在需要 3D 可视化等专用 GUI 的情况下。但如果您的工具只需要向用户呈现可编辑的表单和表格数据，那么基于 Web 的工具可能是您的最佳选择。

who provide language translation services to us. Stand-alone tools still have their place of course, especially when specialized GUIs such as 3D visualization are required. But if your tool only needs to present the user with editable forms and tabular data, a web-based tool may be your best bet.

2
行业工具

在我们踏上探索游戏引擎架构的奇妙旅程之前，我们有必要先掌握一些基本的工具和准备。在接下来的两章中，我们将回顾旅程中所需的软件工程概念和实践。在第二章中，我们将探索大多数专业游戏工程师使用的工具。然后在第三章中，我们将回顾面向对象编程、设计模式和大规模 C++ 编程领域的一些关键主题，以完善我们的准备工作。

游戏开发是软件工程中要求最高、涉及范围最广的领域之一，所以相信我，如果我们想要安全地应对接下来有时充满危险的领域，就必须做好充分的准备。对于一些读者来说，本章和下一章的内容可能非常熟悉。然而，我鼓励你不要完全跳过这些章节。我希望它们能让你愉快地复习知识；说不定，你甚至还能学到一两个新的技巧。

2.1 版本控制

版本控制系统是一种允许多个用户共同处理一组文件的工具。它维护每个文件的历史记录，以便更改

2
Tools of the Trade

Before we embark on our journey across the fascinating landscape of game engine architecture, it is important that we equip ourselves with some basic tools and provisions. In the next two chapters, we will review the software engineering concepts and practices that we will need during our voyage. In Chapter 2, we'll explore the tools used by the majority of professional game engineers. Then in Chapter 3, we'll round out our preparations by reviewing some key topics in the realms of object-oriented programming, design patterns and large-scale C++ programming.

Game development is one of the most demanding and broad areas of software engineering, so believe me, we'll want to be well equipped if we are to safely navigate the sometimes-treacherous terrain we'll be covering. For some readers, the contents of this chapter and the next will be very familiar. However, I encourage you not to skip these chapters entirely. I hope that they will serve as a pleasant refresher; and who knows, you might even pick up a new trick or two.

2.1 Version Control

A *version control* system is a tool that permits multiple users to work on a group of files collectively. It maintains a history of each file so that changes

可以在必要时进行跟踪和还原。它允许多个用户同时修改文件（甚至是同一个文件），而不会每个人都破坏彼此的工作。版本控制因其跟踪文件版本历史的能力而得名。它有时被称为源代码控制，因为它主要由计算机程序员用来管理他们的源代码。但是，版本控制也可以用于其他类型的文件。版本控制系统通常最擅长管理文本文件，原因我们将在下面发现。但是，许多游戏工作室使用单一版本控制系统来管理源代码文件（文本）和游戏资产，如纹理、3D 网格、动画和音频文件（通常是二进制）。

2.1.1 为什么要使用版本控制?

无论何时，由多名工程师组成的团队开发软件时，版本控制都至关重要。版本控制

- 提供一个中央存储库，工程师可以从中共享源代码；
- 保存对每个源文件所做更改的历史记录；
- 提供允许标记和随后检索代码库特定版本的机制；以及
- 允许从主开发线中分支出代码版本，此功能通常用于制作演示或为旧版本的软件制作补丁。

源代码控制系统即使在单工程师项目中也很有用。
虽然它的多用户功能并不重要，但它的其他功能，例如维护更改历史、标记版本、为演示和补丁创建分支、跟踪错误等，仍然是无价的。

2.1.2 常见的版本控制系统

以下是您在游戏工程师职业生涯中可能会遇到的最常见的源代码控制系统。

- SCCS 和 RCS。源代码控制系统 (SCCS) 和修订控制系统 (RCS) 是两个最古老的版本控制系统。两者都采用命令行界面。它们主要在 UNIX 平台上流行。
- CVS 。并行版本系统 (CVS) 是一个基于命令行的专业级重型源代码控制系统，最初建立在

can be tracked and reverted if necessary. It permits multiple users to modify files—even the same file—simultaneously, without everyone stomping on each other's work. Version control gets its name from its ability to track the version history of files. It is sometimes called *source control*, because it is primarily used by computer programmers to manage their source code. However, version control can be used for other kinds of files as well. Version control systems are usually best at managing text files, for reasons we will discover below. However, many game studios use a single version control system to manage both source code files (which are text) and game assets like textures, 3D meshes, animations and audio files (which are usually binary).

2.1.1 Why Use Version Control?

Version control is crucial whenever software is developed by a team of multiple engineers. Version control

- provides a central repository from which engineers can share source code;
- keeps a history of the changes made to each source file;
- provides mechanisms allowing specific versions of the code base to be tagged and later retrieved; and
- permits versions of the code to be branched off from the main development line, a feature often used to produce demos or make patches to older versions of the software.

A source control system can be useful even on a single-engineer project. Although its multiuser capabilities won't be relevant, its other abilities, such as maintaining a history of changes, tagging versions, creating branches for demos and patches, tracking bugs, etc., are still invaluable.

2.1.2 Common Version Control Systems

Here are the most common source control systems you'll probably encounter during your career as a game engineer.

- *SCCS and RCS*. The Source Code Control System (SCCS) and the Revision Control System (RCS) are two of the oldest version control systems. Both employ a command-line interface. They are prevalent primarily on UNIX platforms.
- *CVS*. The Concurrent Version System (CVS) is a heavy-duty professional-grade command-line-based source control system, originally built on top

RCS 的版本（但现在已作为独立工具实现）。CVS 在 UNIX 系统上广泛使用，但也可用于其他开发平台，例如 Microsoft Windows。它是开源的，并遵循 GNU 通用公共许可证 (GPL)。CVSNT（也称为 WinCVS）是基于 CVS 并与之兼容的 Windows 原生实现。

• Subversion。Subversion 是一个开源版本控制系统，旨在取代并改进 CVS。由于它是开源且免费的，因此对于个人项目、学生项目和小型工作室来说，它是一个不错的选择。

• Git 。这是一个开源版本控制系统，已被用于许多历史悠久的项目，包括 Linux 内核。在 git 开发模型中，程序员对文件进行更改并将更改提交到分支。然后，程序员可以快速轻松地将他的更改合并到任何其他代码分支中，因为 git“知道”如何回退一系列差异并将它们重新应用到新的基础修订版本上 - git 将此过程称为变基。最终结果是版本控制系统在处理多个代码分支时非常高效且快速。Git 是一个分布式版本控制系统；个人程序员大部分时间可以在本地工作，但他们可以轻松地将他们的更改合并到共享代码库中。它在单人软件项目中也非常容易使用，因为无需担心任何服务器设置。有关 git 的更多信息，请访问 http://git-scm.com/。

• Perforce 。Perforce 是一款专业级的源代码控制系统，具有文本和 GUI 界面。Perforce 的一大亮点是其“变更列表”概念。变更列表是已修改的源文件的集合，这些文件作为一个逻辑单元。变更列表会以原子方式签入存储库——要么提交整个变更列表，要么全部不提交。许多游戏公司都在使用 Perforce，包括顽皮狗和 Electronic Arts 。

• NxN Alienbrain。Alienbrain 是一款功能强大且丰富的源代码控制系统，专为游戏行业设计。它最引以为豪的是支持包含文本源代码文件和二进制游戏美术资源的超大型数据库，并具有可定制的用户界面，可针对特定领域（例如艺术家、制作人或程序员）进行定制。

• ClearCase。Rational ClearCase 是一款专业级源代码控制系统，适用于大型软件项目。它功能强大，并采用独特的用户界面，扩展了 Windows 的功能。

of RCS (but now implemented as a stand-alone tool). CVS is prevalent on UNIX systems but is also available on other development platforms such as Microsoft Windows. It is open source and licensed under the Gnu General Public License (GPL). CVSNT (also known as WinCVS) is a native Windows implementation that is based on, and compatible with, CVS.

- *Subversion*. Subversion is an open source version control system aimed at replacing and improving upon CVS. Because it is open source and hence free, it is a great choice for individual projects, student projects and small studios.
- *Git*. This is an open source revision control system that has been used for many venerable projects, including the Linux kernel. In the git development model, the programmer makes changes to files and commits the changes to a branch. The programmer can then merge his changes into any other code branch quickly and easily, because git “knows” how to rewind a sequence of diffs and reapply them onto a new base revision—a process git calls *rebasing*. The net result is a revision control system that is highly efficient and fast when dealing with multiple code branches. Git is a distributed version control system; individual programmers can work locally much of the time, yet they can merge their changes into a shared codebase easily. It’s also very easy to use on a one-person software project, because there’s zero server set-up to worry about. More information on git can be found at http://git-scm.com/.
- *Perforce*. Perforce is a professional-grade source control system, with both text-based and GUI interfaces. One of Perforce’s claims to fame is its concept of *change lists*. A change list is a collection of source files that have been modified as a logical unit. Change lists are checked into the repository atomically—either the entire change list is submitted, or none of it is. Perforce is used by many game companies, including *Naughty Dog* and *Electronic Arts*.
- *NxN Alienbrain*. Alienbrain is a powerful and feature-rich source control system designed explicitly for the game industry. Its biggest claim to fame is its support for very large databases containing both text source code files and binary game art assets, with a customizable user interface that can be targeted at specific disciplines such as artists, producers or programmers.
- *ClearCase*. Rational ClearCase is a professional-grade source control system aimed at very large-scale software projects. It is powerful and employs a unique user interface that extends the functionality of Windows

资源管理器。我还没有看到 ClearCase 在游戏行业中得到广泛应用，也许是因为它是比较昂贵的版本控制系统之一。

• Microsoft Visual SourceSafe。SourceSafe 是一个轻量级源代码控制包，已成功用于一些游戏项目。

2.1.3 Subversion 和 TortoiseSVN 概述

我选择在本书中重点介绍 Subversion，原因如下。首先，它是免费的，这一点非常好。根据我的经验，它运行良好且可靠。Subversion 中央仓库的设置非常简单，而且正如我们将看到的，如果您不想费力自行设置，市面上已经有许多免费的仓库服务器。此外，还有许多优秀的 Windows 和 Mac Subversion 客户端，例如免费的 Windows 版 TortoiseSVN。因此，虽然 Subversion 可能并非大型商业项目的最佳选择（我个人更喜欢 Perforce 或 git），但我发现它非常适合小型个人和教育项目。让我们来看看如何在 Microsoft Windows PC 开发平台上设置和使用 Subversion。在此过程中，我们将回顾几乎适用于任何版本控制系统的核心概念。

Subversion 与大多数其他版本控制系统一样，采用客户端/服务器架构。服务器管理一个中央存储库，其中存储了版本控制的目录层次结构。客户端连接到服务器并请求操作，例如检出目录树的最新版本、提交一个或多个文件的新更改、标记修订版本、创建存储库分支等等。我们这里不讨论如何设置服务器；我们假设您已经拥有服务器，而是专注于设置和使用客户端。您可以通过阅读 [43] 的第 6 章来学习如何设置 Subversion 服务器。但是，您可能永远不需要这样做，因为您总能找到免费的 Subversion 服务器。例如，HelixTeamHub 在 http://info.perforce.com/try-perforce-helix-teamhub-free.html 上提供 Subversion 代码托管服务，对于用户数不超过 5 人且存储空间不超过 1 GB 的项目，免费。Beanstalk 是另一个不错的托管服务，但他们会收取少量月费。

2.1.4 设置代码存储库

开始使用 Subversion 最简单的方法是访问 HelixTeamHub 或类似的 SVN 托管服务网站，并设置一个 Subversion 仓库。创建一个帐户，就可以开始使用了。大多数托管网站都提供了简单易懂的说明。

Explorer. I haven't seen ClearCase used much in the game industry, perhaps because it is one of the more expensive version control systems.

- *Microsoft Visual SourceSafe*. SourceSafe is a lightweight source control package that has been used successfully on some game projects.

2.1.3 Overview of Subversion and TortoiseSVN

I have chosen to highlight Subversion in this book for a few reasons. First off, it's free, which is always nice. It works well and is reliable, in my experience. A Subversion central repository is quite easy to set up, and as we'll see, there are already a number of free repository servers out there if you don't want to go to the trouble of setting one up yourself. There are also a number of good Windows and Mac Subversion clients, such as the freely available TortoiseSVN for Windows. So while Subversion may not be the best choice for a large commercial project (I personally prefer Perforce or git for that purpose), I find it perfectly suited to small personal and educational projects. Let's take a look at how to set up and use Subversion on a Microsoft Windows PC development platform. As we do so, we'll review core concepts that apply to virtually any version control system.

Subversion, like most other version control systems, employs a client-server architecture. The server manages a central repository, in which a version-controlled directory hierarchy is stored. Clients connect to the server and request operations, such as checking out the latest version of the directory tree, committing new changes to one or more files, tagging revisions, branching the repository and so on. We won't discuss setting up a server here; we'll assume you have a server, and instead we will focus on setting up and using the client. You can learn how to set up a Subversion server by reading Chapter 6 of [43]. However, you probably will never need to do so, because you can always find free Subversion servers. For example, HelixTeamHub provides Subversion code hosting at http://info.perforce.com/try-perforce-helix-teamhub-free.html that's free for projects of 5 users or fewer and up to 1 GB of storage. Beanstalk is another good hosting service, but they do charge a nominal monthly fee.

2.1.4 Setting up a Code Repository

The easiest way to get started with Subversion is to visit the website of HelixTeamHub or a similar SVN hosting service, and set up a Subversion repository. Create an account and you're off to the races. Most hosting sites offer easy-to-follow instructions.

图 2.1. TortoiseSVN 初始签出对话框。

图 2.2. TortoiseSVN 用户身份验证对话框。

创建存储库后，通常可以在托管服务的网站上进行管理。您可以添加和删除用户、控制选项以及执行大量高级任务。但接下来您真正需要做的就是设置一个 Subversion 客户端并开始使用您的存储库。

2.1.5 安装 TortoiseSVN

TortoiseSVN 是一款流行的 Subversion 前端。它扩展了 Microsoft Windows 资源管理器的功能，通过便捷的右键菜单和叠加图标来显示版本控制文件和文件夹的状态。

要获取 TortoiseSVN，请访问 http://tortoisesvn.tigris.org/。从下载页面下载最新版本。双击下载的 .msi 文件并按照安装向导的说明进行安装。

TortoiseSVN 安装完成后，您可以转到 Windows 资源管理器中的任何文件夹并右键单击 - TortoiseSVN 的菜单扩展现在应该是可见的。要连接到现有的代码存储库（例如您在 HelixTeamHub 上创建的代码存储库），请在本地硬盘上创建一个文件夹，然后右键单击并选择“SVN Checkout...”。将出现如图 2.1 所示的对话框。在“存储库 URL”字段中，输入存储库的 URL。如果您使用的是 HelixTeamHub，则该 URL 应为 https://helixteamhub.cloud/mr3/projects/myprojectname/repositories/subversion/myrepository，其中 myprojectname 是您首次创建项目时命名的项目，myrepository 是您的 SVN 代码存储库的名称。

Figure 2.1. TortoiseSVN initial check-out dialog.

Figure 2.2. TortoiseSVN user authentication dialog.

Once you've created your repository, you can typically administer it on the hosting service's website. You can add and remove users, control options and perform a wealth of advanced tasks. But all you really need to do next is set up a Subversion client and start using your repository.

2.1.5 Installing TortoiseSVN

TortoiseSVN is a popular front end for Subversion. It extends the functionality of the Microsoft Windows Explorer via a convenient right-click menu and overlay icons to show you the status of your version-controlled files and folders.

To get TortoiseSVN, visit http://tortoisesvn.tigris.org/. Download the latest version from the download page. Install it by double-clicking the .msi file that you've downloaded and following the installation wizard's instructions.

Once TortoiseSVN is installed, you can go to any folder in Windows Explorer and right-click—TortoiseSVN's menu extensions should now be visible. To connect to an existing code repository (such as one you created on HelixTeamHub), create a folder on your local hard disk and then right-click and select "SVN Checkout…." The dialog shown in Figure 2.1 will appear. In the "URL of repository" field, enter your repository's URL. If you are using HelixTeamHub, it would be https://helixteamhub.cloud/mr3/projects/*myprojectname*/repositories/subversion/*myrepository*, where *myprojectname* is whatever you named your project when you first created it, and *myrepository* is the name of your SVN code repository.

图 2.3. 文件版本历史记录。

图 2.4.编辑版本控制文件的本地副本。

现在您应该会看到如图 2.2 所示的对话框。输入您的用户名和密码；勾选此对话框中的“保存身份验证”选项后，您无需再次登录即可使用您的仓库。仅当您在自己的计算机上工作时才选择此选项——切勿在多人共享的计算机上工作。

验证用户名后，TortoiseSVN 会将您仓库的全部内容下载（“签出”）到您的本地磁盘。如果您刚刚设置了仓库，那么……什么也没有！您创建的文件夹仍然为空。但现在它已连接到 HelixTeamHub（或您的服务器所在的任何位置）上的 Subversion 仓库。如果您刷新 Windows 资源管理器窗口（按 F5），您应该会在文件夹上看到一个绿白相间的小勾号。此图标表示该文件夹已通过 TortoiseSVN 连接到 Subversion 仓库，并且该仓库的本地副本已更新。

2.1.6 文件版本、更新和提交

正如我们所见，像 Subversion 这样的源代码控制系统的主要目的之一，就是通过在服务器上维护一个中央存储库或所有源代码的“主”版本，允许多名程序员在同一个软件代码库上工作。服务器会维护每个文件的版本历史记录，如图 2.3 所示。此功能对于大规模多程序员软件开发至关重要。例如，如果有人犯了错误并提交了“破坏构建”的代码，您可以轻松回溯到过去撤销这些更改（并查看日志以查明罪魁祸首！）。您还可以抓取任何时间点的代码快照，以便使用、演示或修补软件的先前版本。

Figure 2.3. File version histories.

Figure 2.4. Editing the local copy of a version-controlled file.

You should now see the dialog shown in Figure 2.2. Enter your user name and password; checking the “Save authentication” option on this dialog allows you to use your repository without ever having to log in again. Only select this option if you are working on your own personal machine—never on a machine that is shared by many users.

Once you’ve authenticated your user name, TortoiseSVN will download (“check out”) the entire contents of your repository to your local disk. If you have just set up your repository, this will be …nothing! The folder you created will still be empty. But now it is connected to your Subversion repository on HelixTeamHub (or wherever your server is located). If you refresh your Windows Explorer window (hit F5), you should now see a little green and white checkmark on your folder. This icon indicates that the folder is connected to a Subversion repository via TortoiseSVN and that the local copy of the repository is up to date.

2.1.6 File Versions, Updating and Committing

As we’ve seen, one of the key purposes of any source control system like Subversion is to allow multiple programmers to work on a single software code base by maintaining a central repository or “master” version of all the source code on a server. The server maintains a version history for each file, as shown in Figure 2.3. This feature is crucial to large-scale multiprogrammer software development. For example, if someone makes a mistake and checks in code that “breaks the build,” you can easily go back in time to undo those changes (and check the log to see who the culprit was!). You can also grab a snapshot of the code as it existed at any point in time, allowing you to work with, demonstrate or patch previous versions of the software.

图 2.5. TortoiseSVN 提交对话框。

每个程序员都会在自己的机器上获得一份代码的本地副本。使用 TortoiseSVN 时，您可以通过如上所述的“签出”代码库来获取初始工作副本。您应该定期更新本地副本，以反映其他程序员可能做出的任何更改。您可以通过右键单击文件夹，然后从弹出菜单中选择“SVN 更新”来执行此操作。

您可以在代码库的本地副本上工作，而不会影响团队中的其他程序员（图 2.4）。当您准备好与其他人共享更改时，您可以将更改提交到存储库（也称为提交或签入）。您可以通过右键单击要提交的文件夹，然后从弹出菜单中选择“SVN 提交...”来执行此操作。您将看到一个如图 2.5 所示的对话框，要求您确认更改。

在提交操作期间，Subversion 会生成每个文件的本地版本与存储库中同一文件的最新版本之间的差异。“diff”表示差异，通常通过对文件的两个版本进行逐行比较来生成。您可以在 TortoiseSVN 提交对话框（图 2.5）中双击任意文件，查看您的版本与服务器上最新版本之间的差异（即您所做的更改）。已更改的文件（即任何“有差异”的文件）将被提交。这会将存储库中的最新版本替换为本地版本，并添加

Figure 2.5. TortoiseSVN Commit dialog.

Each programmer gets a local copy of the code on his or her machine. In the case of TortoiseSVN, you obtain your initial working copy by “checking out” the repository, as described above. Periodically you should update your local copy to reflect any changes that may have been made by other programmers. You do this by right-clicking on a folder and selecting “SVN Update” from the pop-up menu.

You can work on your local copy of the code base without affecting the other programmers on the team (Figure 2.4). When you are ready to share your changes with everyone else, you *commit* your changes to the repository (also known as *submitting* or *checking in*). You do this by right-clicking on the folder you want to commit and selecting “SVN Commit…” from the pop-up menu. You will get a dialog like the one shown in Figure 2.5, asking you to confirm the changes.

During a commit operation, Subversion generates a *diff* between your local version of each file and the latest version of that same file in the repository. The term “diff” means difference, and it is typically produced by performing a line-by-line comparison of the two versions of the file. You can double-click on any file in the TortoiseSVN Commit dialog (Figure 2.5) to see the diffs between your version and the latest version on the server (i.e., the changes you made). Files that have changed (i.e., any files that “have diffs”) are committed. This replaces the latest version in the repository with your local version, adding a

图 2.6。 和-允许对存储库进行本地编辑。

该文件的版本历史记录中新增了一条记录。提交时，任何未更改的文件（即本地副本与存储库中的最新版本完全相同）都会被默认忽略。提交操作示例如图 2.6 所示。

如果您在提交之前创建了任何新文件，它们将在提交对话框中列为“未版本控制”。您可以勾选它们旁边的小复选框，以将它们添加到存储库。您在本地删除的任何文件同样会显示为“丢失”——如果您勾选了它们的复选框，它们将从存储库中删除。您还可以在提交对话框中输入注释。此注释将添加到存储库的历史记录中，以便您和团队中的其他人了解这些文件签入的原因。

2.1.7 多重检出、分支和合并

某些版本控制系统需要独占签出。这意味着您必须首先通过签出并锁定文件来表明您要修改文件的意图。签出的文件在您的本地磁盘上是可写的，其他任何人都不能签出。存储库中的所有其他文件在您的本地磁盘上都是只读的。一旦您完成文件编辑，您就可以签入它，这将释放锁定并将更改提交到存储库供其他所有人查看。独占锁定文件进行编辑的过程可确保没有两个人可以同时编辑同一个文件。

Subversion、CVS、Perforce 和许多其他高质量的版本控制系统也允许多次签出，即，您可以在其他人编辑同一个文件时对其进行编辑。任何先提交的用户的更改都将成为存储库中该文件的最新版本。其他用户的任何后续提交都要求程序员将其更改与先前提交的程序员所做的更改合并。

由于对同一个文件进行了多组更改（差异），版本控制系统必须合并这些更改才能生成文件的最终版本。这通常不是什么大问题，事实上许多冲突可以由版本控制系统自动解决。例如，如果您更改了函数 f()，而另一个程序员更改了函数 g()，那么您编辑的行数将与另一个程序员的编辑行数不同。在这种情况下，您的更改与他或她的更改之间的合并通常会自动解决，而不会出现任何冲突。但是，如果你们都对同一个函数 f() 进行了更改，那么第二个提交其更改的程序员将需要进行三方合并（参见图 2.7）。

为了使三方合并能够正常工作，版本控制服务器必须非常智能

Figure 2.6. Committing local edits to the repository.

new entry to the file's version history. Any files that have not changed (i.e., your local copy is identical to the latest version in the repository) are ignored by default during a commit. An example commit operation is shown in Figure 2.6.

If you created any new files prior to the commit, they will be listed as "non-versioned" in the Commit dialog. You can check the little check boxes beside them in order to add them to the repository. Any files that you deleted locally will likewise show up as "missing"—if you check their check boxes, they will be deleted from the repository. You can also type a comment in the Commit dialog. This comment is added to the repository's history log, so that you and others on your team will know why these files were checked in.

2.1.7 Multiple Check-Out, Branching and Merging

Some version control systems require *exclusive check-out*. This means that you must first indicate your intentions to modify a file by *checking it out* and *locking* it. The file(s) that are checked out to you are writable on your local disk and cannot be checked out by anyone else. All other files in the repository are read-only on your local disk. Once you're done editing the file, you can check it in, which releases the lock and commits the changes to the repository for everyone else to see. The process of exclusively locking files for editing ensures that no two people can edit the same file simultaneously.

Subversion, CVS, Perforce and many other high-quality version control systems also permit *multiple check-out*, i.e., you can edit a file while someone else is editing that same file. Whichever user's changes are committed first become the latest version of the file in the repository. Any subsequent commits by other users require that programmer to merge his or her changes with the changes made by the programmer(s) who committed previously.

Because more than one set of changes (diffs) have been made to the same file, the version control system must *merge* the changes in order to produce a final version of the file. This is often not a big deal, and in fact many conflicts can be resolved automatically by the version control system. For example, if you changed function `f()` and another programmer changed function `g()`, then your edits would have been to a different range of lines in the file than those of the other programmer. In this case, the merge between your changes and his or her changes will usually resolve automatically without any conflicts. However, if you were both making changes to the same function `f()`, then the second programmer to commit his or her changes will need to do a *three-way merge* (see Figure 2.7).

For three-way merges to work, the version control server has to be smart

图 2.7.由于两个不同用户的本地编辑而进行的三向合并。

足以追踪本地磁盘上当前每个文件的版本。这样，当你合并文件时，系统就能知道哪个版本是基础版本（即共同的祖先，例如图 2.7 中的版本 4）。

Subversion 允许多次签出，实际上它根本不需要你明确地签出文件。你只需在本地开始编辑文件即可——所有文件在本地磁盘上始终可写入。（顺便说一句，在我看来，这也是 Subversion 无法很好地扩展到大型项目的原因之一。为了确定哪些文件发生了更改，Subversion 必须搜索整个源文件树，这可能会很慢。像 Perforce 这样的版本控制系统会明确地跟踪你修改过的文件，在处理大量代码时通常更容易使用。但对于小型项目，Subversion 的方法就足够了。）

当您通过右键单击任何文件夹并从弹出菜单中选择“SVN 提交...”来执行提交操作时，系统可能会提示您将您的更改与其他人所做的更改合并。但是，如果自您上次更新本地副本以来没有人更改过该文件，那么您的更改

Figure 2.7. Three-way merge due to local edits by two different users.

enough to keep track of which version of each file you currently have on your local disk. That way, when you merge the files, the system will know which version is the base version (the common ancestor, such as version 4 in Figure 2.7).

Subversion permits multiple check-out, and in fact it doesn't require you to check out files explicitly at all. You simply start editing the files locally—all files are writable on your local disk at all times. (By the way, this is one reason that Subversion doesn't scale well to large projects, in my opinion. To determine which files you have changed, Subversion must search the entire tree of source files, which can be slow. Version control systems like Perforce, which explicitly keep track of which files you have modified, are usually easier to work with when dealing with large amounts of code. But for small projects, Subversion's approach works just fine.)

When you perform a commit operation by right-clicking on any folder and selecting "SVN Commit…" from the pop-up menu, you may be prompted to *merge* your changes with changes made by someone else. But if no one has changed the file since you last updated your local copy, then your changes

无需您进行任何进一步的操作即可提交。这是一个非常方便的功能，但也可能存在危险。建议您始终仔细检查您的提交，以确保没有提交任何您不想修改的文件。当 TortoiseSVN 显示“提交文件”对话框时，您可以双击单个文件，查看在点击“确定”按钮之前所做的修改。

2.1.8 删除文件

当文件从版本库中删除时，它并没有真正消失。该文件仍然存在于版本库中，但其最新版本仅被标记为“已删除”，以便用户在本地目录树中不再看到该文件。您仍然可以通过右键单击文件所在的文件夹，并从 TortoiseSVN 菜单中选择“显示日志”来查看和访问已删除文件的先前版本。

您可以通过将本地目录更新到文件被标记为已删除的版本之前的版本来恢复已删除的文件。然后，只需再次提交该文件即可。这会将文件的最新删除版本替换为删除之前的版本，从而有效地恢复文件。

2.2 编译器、链接器和 IDE

编译型语言（例如 C++）需要编译器和链接器才能将源代码转换为可执行程序。C++ 有许多可用的编译器/链接器，但对于 Microsoft Windows 平台，最常用的软件包可能是 Microsoft Visual Studio。该产品功能齐全的专业版和企业版可从 Microsoft 商店在线购买，其轻量级版本 Visual Studio 社区版（以前称为 Visual Studio Express）可从 https://www.visualstudio.com/downloads 免费下载。有关 Visual Studio 以及标准 C 和 C++ 库的文档可在 Microsoft 开发者网络 (MSDN) 网站 (https://msdn.microsoft) 在线获取。

com/en-us）。

Visual Studio 不仅仅是一个编译器和链接器。它是一个集成开发环境 (IDE)，包含一个功能齐全的源代码文本编辑器，以及一个强大的源代码级和机器级调试器。本书主要关注 Windows 平台，因此我们将深入探讨 Visual Studio。下文中学习的大部分内容也适用于其他编译器、链接器和调试器，例如 LLVM/Clang、gcc/gdb 等。

will be committed without any further action on your part. This is a very convenient feature, but it can also be dangerous. It's a good idea to always check your commits carefully to be sure you aren't committing any files that you didn't intend to modify. When TortoiseSVN displays its Commit Files dialog, you can double-click on an individual file in order to see the diffs you made prior to hitting the "OK" button.

2.1.8 Deleting Files

When a file is deleted from the repository, it's not really gone. The file still exists in the repository, but its latest version is simply marked "deleted" so that users will no longer see the file in their local directory trees. You can still see and access previous versions of a deleted file by right-clicking on the folder in which the file was contained and selecting "Show log" from the TortoiseSVN menu.

You can undelete a deleted file by updating your local directory to the version immediately before the version in which the file was marked deleted. Then simply commit the file again. This replaces the latest deleted version of the file with the version just prior to the deletion, effectively undeleting the file.

2.2 Compilers, Linkers and IDEs

Compiled languages, such as C++, require a *compiler* and *linker* in order to transform source code into an executable program. There are many compilers/linkers available for C++, but for the Microsoft Windows platform, the most commonly used package is probably Microsoft Visual Studio. The fully-featured Professional and Enterprise Editions of the product can be purchased online from the Microsoft store, and its lighter-weight cousin Visual Studio Community Edition (previously known as Visual Studio Express) is available for free download at https://www.visualstudio.com/downloads. Documentation on Visual Studio and the standard C and C++ libraries is available online at the Microsoft Developer Network (MSDN) site (https://msdn.microsoft.com/en-us).

Visual Studio is more than just a compiler and linker. It is an *integrated development environment* (IDE), including a slick and fully featured *text editor* for source code and a powerful source-level and machine-level *debugger*. In this book, our primary focus is the Windows platform, so we'll investigate Visual Studio in some depth. Much of what you learn below will be applicable to other compilers, linkers and debuggers such as LLVM/Clang, gcc/gdb, and

Intel C/C++ 编译器。所以，即使您不打算使用 Visual Studio，我也建议您仔细阅读本节。您将找到各种关于使用编译器、链接器和调试器的实用技巧。

2.2.1 源文件、头文件和翻译单元

用 C++ 编写的程序由源文件组成。这些源文件通常以 .c、.cc、.cxx 或 .cpp 为扩展名，包含程序的大部分源代码。源文件在技术上被称为翻译单元，因为编译器每次只会将一个源文件从 C++ 翻译成机器码。

一种称为头文件的特殊源文件通常用于在翻译单元之间共享信息，例如类型声明和函数原型。编译器看不到头文件。相反，C++ 预处理器会在将翻译单元发送给编译器之前，用相应头文件的内容替换每个 #include 语句。这是一个微妙但非常重要的区别。从程序员的角度来看，头文件是作为独立文件存在的——但由于预处理器的头文件扩展，编译器看到的只是翻译单元。

2.2.2 库、可执行文件和动态链接库

编译翻译单元后，生成的机器代码将被放置在目标文件中（在 Windows 系统中，文件扩展名为 .obj；在 UNIX 系统中，文件扩展名为 .o）。目标文件中的机器代码如下：

- 可重定位，这意味着代码驻留的内存地址尚未确定，并且
- 未链接，这意味着对在翻译单元之外定义的函数和全局数据的任何外部引用尚未得到解决。

目标文件可以归类到称为“库”的组中。库只是一个存档，类似于 ZIP 或 tar 文件，包含零个或多个目标文件。库的存在只是为了方便起见，允许将大量目标文件收集到一个易于使用的文件中。

目标文件和库由链接器链接成可执行文件。可执行文件包含完全解析的机器码，可由操作系统加载和运行。链接器的工作包括：

the Intel C/C++ compiler. So even if you're not planning on ever using Visual Studio, I suggest you read this section carefully. You'll find all sorts of useful tips on using compilers, linkers and debuggers in general.

2.2.1 Source Files, Headers and Translation Units

A program written in C++ is comprised of *source files*. These typically have a .c, .cc, .cxx or .cpp extension, and they contain the bulk of your program's source code. Source files are technically known as *translation units*, because the compiler *translates* one source file at a time from C++ into machine code.

A special kind of source file, known as a *header file*, is often used in order to share information, such as type declarations and function prototypes, between translation units. Header files are not seen by the compiler. Instead, the C++ *preprocessor* replaces each `#include` statement with the contents of the corresponding header file prior to sending the translation unit to the compiler. This is a subtle but very important distinction to make. Header files exist as distinct files from the point of view of the programmer—but thanks to the preprocessor's header file expansion, all the compiler ever sees are translation units.

2.2.2 Libraries, Executables and Dynamic Link Libraries

When a translation unit is compiled, the resulting machine code is placed in an object file (files with a .obj extension under Windows or .o under UNIX-based operating systems). The machine code in an object file is:

- *relocatable*, meaning that the memory addresses at which the code resides have not yet been determined, and
- *unlinked*, meaning that any external references to functions and global data that are defined outside the translation unit have not yet been resolved.

Object files can be collected into groups called *libraries*. A library is simply an archive, much like a ZIP or tar file, containing zero or more object files. Libraries exist merely as a convenience, permitting a large number of object files to be collected into a single easy-to-use file.

Object files and libraries are *linked* into an executable by the linker. The executable file contains fully resolved machine code that can be loaded and run by the operating system. The linker's jobs are:

- 计算所有机器代码的最终相对地址，因为它将在程序运行时出现在内存中，并且
- 确保每个翻译单元（目标文件）对函数和全局数据的所有外部引用都得到正确解析。

需要记住的是，可执行文件中的机器码仍然是可重定位的，这意味着文件中所有指令和数据的地址仍然是相对于任意基地址的，而不是绝对基地址。程序的最终绝对基地址直到程序实际加载到内存中（即将运行之前）才可知。

动态链接库 (DLL) 是一种特殊的库，其行为类似于常规静态库和可执行文件的混合体。DLL 的行为类似于库，因为它包含可被任意数量的不同可执行文件调用的函数。然而，DLL 的行为也类似于可执行文件，因为它可以由操作系统独立加载，并且它包含一些启动和关闭代码，其运行方式与 C++ 可执行文件中的 main() 函数非常相似。

使用 DLL 的可执行文件包含部分链接的机器码。大多数函数和数据引用在最终可执行文件中已完全解析，但任何对 DLL 中存在的外部函数或数据的引用仍保持未链接状态。运行可执行文件时，操作系统会解析所有未链接函数的地址，具体方法是：定位相应的 DLL，如果尚未加载，则将其加载到内存中，并修补必要的内存地址。动态链接库是一项非常有用的操作系统功能，因为可以在不更改使用它们的可执行文件的情况下更新单个 DLL。

2.2.3 项目与解决方案

现在我们了解了库、可执行文件和动态链接库 (DLL) 之间的区别，接下来让我们看看如何创建它们。在 Visual Studio 中，项目是源文件的集合，这些源文件在编译后会生成库、可执行文件或 DLL。自 VS 2013 以来的所有 Visual Studio 版本中，项目都存储在扩展名为 .vcxproj 的项目文件中。这些文件采用 XML 格式，因此人们可以轻松读取，甚至在必要时手动编辑。

自 Visual Studio 7 (Visual Studio 2003) 以来的所有版本都使用解决方案文件（扩展名为 .sln 的文件）来包含和管理项目集合。解决方案是用于构建一个或多个库、可执行文件和/或 DLL 的依赖和/或独立项目的集合。在 Visual Studio 图形用户界面中，解决方案资源管理器

- to calculate the final relative addresses of all the machine code, as it will appear in memory when the program is run, and
- to ensure that all external references to functions and global data made by each translation unit (object file) are properly resolved.

It's important to remember that the machine code in an executable file is still relocatable, meaning that the addresses of all instructions and data in the file are still *relative* to an arbitrary base address, not absolute. The final absolute base address of the program is not known until the program is actually loaded into memory, just prior to running it.

A *dynamic link library* (DLL) is a special kind of library that acts like a hybrid between a regular static library and an executable. The DLL acts like a library, because it contains functions that can be called by any number of different executables. However, a DLL also acts like an executable, because it can be loaded by the operating system independently, and it contains some start-up and shut-down code that runs much the way the `main()` function in a C++ executable does.

The executables that use a DLL contain *partially linked* machine code. Most of the function and data references are fully resolved within the final executable, but any references to external functions or data that exist in a DLL remain unlinked. When the executable is run, the operating system resolves the addresses of all unlinked functions by locating the appropriate DLLs, loading them into memory if they are not already loaded, and patching in the necessary memory addresses. Dynamically linked libraries are a very useful operating system feature, because individual DLLs can be updated without changing the executable(s) that use them.

2.2.3 Projects and Solutions

Now that we understand the difference between libraries, executables and dynamic link libraries (DLLs), let's see how to create them. In Visual Studio, a *project* is a collection of *source files* which, when compiled, produce a library, an executable or a DLL. In all versions of Visual Studio since VS 2013, projects are stored in project files with a .vcxproj extension. These files are in XML format, so they are reasonably easy for a human to read and even edit by hand if necessary.

All versions of Visual Studio since version 7 (Visual Studio 2003) employ *solution files* (files with a .sln extension) as a means of containing and managing collections of projects. A solution is a collection of dependent and/or independent projects intended to build one or more libraries, executables and/or DLLs. In the Visual Studio graphical user interface, the *Solution Explorer* is

通常显示在主窗口的右侧或左侧，如图2.8所示。

解决方案资源管理器是一个树形视图。解决方案本身位于根目录，项目是其直接子目录。源文件和头文件显示为每个项目的子目录。一个项目可以包含任意数量的用户定义文件夹，嵌套深度不限。文件夹仅用于组织目的，与文件在磁盘上的文件夹结构无关。但是，在设置项目文件夹时，通常的做法是模拟磁盘上的文件夹结构。

2.2.4 构建配置

C/C++ 预处理器、编译器和链接器提供了各种各样的选项来控制代码的构建方式。这些选项通常在编译器运行时在命令行中指定。例如，使用 Microsoft 编译器构建单个翻译单元的典型命令可能如下所示：

```
> cl /c foo.cpp /Fo foo.obj /Wall /Od /Zi
```

这告诉编译器/链接器编译但不链接（ /c ）名为 foo.cpp 的翻译单元，将结果输出到名为 foo.obj 的目标文件（ /Fo foo.obj ），启用所有警告（ /Wall ），关闭所有优化（ /Od ）并生成调试信息（ /Zi ）。LLVM/Clang 的大致等效命令行如下所示：

```
> clang --std=c++14 foo.cpp -o foo.o --Wall -O0 -g
```

图 2.8.VisualStudio 解决方案资源管理器窗口。

usually displayed along the right or left side of the main window, as shown in Figure 2.8.

The Solution Explorer is a tree view. The solution itself is at the root, with the projects as its immediate children. Source files and headers are shown as children of each project. A project can contain any number of user-defined folders, nested to any depth. Folders are for organizational purposes only and have nothing to do with the folder structure in which the files may reside on-disk. However, it is common practice to mimic the on-disk folder structure when setting up a project's folders.

2.2.4 Build Configurations

The C/C++ preprocessor, compiler and linker offer a wide variety of options to control how your code will be built. These options are normally specified on the command line when the compiler is run. For example, a typical command to build a single translation unit with the Microsoft compiler might look like this:

```
> cl /c foo.cpp /Fo foo.obj /Wall /Od /Zi
```

This tells the compiler/linker to compile but not link (`/c`) the translation unit named foo.cpp, output the result to an object file named foo.obj (`/Fo foo.obj`), turn on all warnings (`/Wall`), turn off all optimizations (`/Od`) and generate debugging information (`/Zi`). A roughly equivalent command line for LLVM/Clang would look something like this:

```
> clang --std=c++14 foo.cpp -o foo.o --Wall -O0 -g
```

Figure 2.8. The VisualStudio Solution Explorer window.

现代编译器提供的选项如此之多，以至于每次构建代码时都指定所有选项既不切实际，又容易出错。这时，构建配置就派上用场了。构建配置实际上只是与解决方案中特定项目相关的预处理器、编译器和链接器选项的集合。您可以定义任意数量的构建配置，随意命名，并在每个配置中以不同的方式配置预处理器、编译器和链接器选项。默认情况下，相同的选项会应用于项目中的每个翻译单元，但您可以根据单个翻译单元覆盖全局项目设置。（我建议尽可能避免这种情况，因为这样很难区分哪些 .cpp 文件有自定义设置，哪些没有。）

在 Visual Studio 中创建新项目/解决方案时，默认情况下会创建两个构建配置，分别为“调试”和“发布”。发布版本用于最终交付（发布）给客户的软件版本，而调试版本则用于开发目的。调试版本的运行速度比非调试版本慢，但它为程序员提供了开发和调试程序的宝贵信息。

专业软件开发者通常会为其软件设置两种以上的构建配置。为了理解原因，我们需要了解局部（编译时）和全局（链接时）优化的工作原理——我们将在 2.2.4.2 节中讨论这些优化。现在，我们先不讨论“发布构建”这个有点令人困惑的术语，而是使用“调试构建”（意味着禁用局部和全局优化）和“非调试构建”（意味着启用局部和/或全局优化）。

2.2.4.1 常用构建选项

本节列出了您希望通过游戏引擎项目的构建配置控制的一些最常见的选项。

预处理器设置

C++ 预处理器负责处理 #include 文件的扩展以及 #define 宏的定义和替换。所有现代 C++ 预处理器都具备一个极其强大的功能，那就是能够通过命令行选项（也就是通过构建配置）定义预处理器宏。以这种方式定义的宏就像是用 #define 语句写入源代码一样。对于大多数编译器来说，实现此功能的命令行选项是 -D 或 /D，并且可以使用任意数量的此类指令。

此功能允许您将各种构建选项传达给您的代码，

Modern compilers provide so many options that it would be impractical and error prone to specify all of them every time you build your code. That's where *build configurations* come in. A build configuration is really just a collection of preprocessor, compiler and linker options associated with a particular project in your solution. You can define any number of build configurations, name them whatever you want, and configure the preprocessor, compiler and linker options differently in each configuration. By default, the same options are applied to every translation unit in the project, although you can override the global project settings on an individual translation unit basis. (I recommend avoiding this if at all possible, because it becomes difficult to tell which .cpp files have custom settings and which do not.)

When you create a new project/solution in Visual Studio, by default it creates two build configurations named "Debug" and "Release." The release build is intended for the final version of your software that you'll ship (release) to customers, while the debug build is for development purposes. A debug build runs more slowly than a non-debug build, but it provides the programmer with invaluable information for developing and debugging the program.

Professional software developers often set up more than two build configurations for their software. To understand why, we'll need to understand how local (compile-time) and global (link-time) optimizations work—we'll discuss these kinds of optimizations in Section 2.2.4.2. For now, let's drop the somewhat confusing term "release build" and stick with the terms "debug build" (meaning local and global optimizations are disabled) and "non-debug build" (meaning local and/or global optimizations are enabled).

2.2.4.1 Common Build Options

This section lists some of the most common options you'll want to control via build configurations for a game engine project.

Preprocessor Settings

The C++ preprocessor handles the expansion of `#included` files and the definition and substitution of `#defined` macros. One extremely powerful feature of all modern C++ preprocessors is the ability to define preprocessor macros via command-line options (and hence via build configurations). Macros defined in this way act as though they had been written into your source code with a `#define` statement. For most compilers, the command line option for this is `-D` or `/D`, and any number of these directives can be used.

This feature allows you to communicate various build options to your code,

无需修改源代码本身。举一个常见的例子，符号 _DEBUG 始终为调试版本定义，而在非调试版本中，则定义符号 NDEBUG。源代码可以检查这些标志，从而“知道”它是在调试模式还是非调试模式下构建的。这被称为条件编译。例如，

```
void f()
{
#ifdef _DEBUG
    printf("Calling function f()\n");
#endif
    // ...
}
```

编译器还可以根据其对编译环境和目标平台的了解，自由地将“神奇的”预处理宏引入到您的代码中。例如，大多数 C/C++ 编译器在编译 C++ 文件时都会定义宏 __cplusplus。这使得编写的代码能够自动适应 C 或 C++ 的编译。

再举一个例子，每个编译器都通过一个“神奇的”预处理宏来在源代码中标识自己。在 Microsoft 编译器下编译代码时，会定义宏 _MSC_VER；在 GNU 编译器 (gcc) 下编译时，则会定义宏 __GNUC__，其他编译器也是如此。代码运行的目标平台同样也通过宏来标识。例如，在为 32 位 Windows 机器构建代码时，始终会定义符号 _WIN32。这些关键特性允许编写跨平台代码，因为它们使代码能够“知道”哪个编译器正在编译它，以及它将在哪个目标平台上运行。

编译器设置

最常见的编译器选项之一是控制编译器是否应在其生成的目标文件中包含调试信息。调试器会使用这些信息来单步执行代码、显示变量值等等。调试信息会使可执行文件占用的磁盘空间更大，也为黑客逆向代码打开了方便之门，因此，在最终发布的可执行文件中，调试信息总是会被剥离。然而，在开发过程中，调试信息非常重要，应该始终包含在构建过□□程中。

还可以告知编译器是否扩展内联函数。
当内联函数扩展关闭时，每个内联函数在内存中只出现一次，位于不同的地址。这使得跟踪

without having to modify the source code itself. As a ubiquitous example, the symbol `_DEBUG` is always defined for a debug build, while in non-debug builds, the symbol `NDEBUG` is defined instead. The source code can check these flags and in effect “know” whether it is being built in debug or non-debug mode. This is known as *conditional compilation*. For example,

```
void f()
{
#ifdef _DEBUG
    printf("Calling function f()\n");
#endif
    // ...
}
```

The compiler is also free to introduce “magic” preprocessor macros into your code, based on its knowledge of the compilation environment and target platform. For example, the macro `__cplusplus` is defined by most C/C++ compilers when compiling a C++ file. This allows code to be written that automatically adapts to being compiled for C or C++.

As another example, every compiler identifies itself to the source code via a “magic” preprocessor macro. When compiling code under the Microsoft compiler, the macro `_MSC_VER` is defined; when compiling under the GNU compiler (gcc), the macro `__GNUC__` is defined instead and so on for the other compilers. The target platform on which the code will be run is likewise identified via macros. For example, when building for a 32-bit Windows machine, the symbol `_WIN32` is always defined. These key features permit cross-platform code to be written, because they allow your code to “know” what compiler is compiling it and on which target platform it is destined to be run.

Compiler Settings

One of the most common compiler options controls whether or not the compiler should include *debugging information* with the object files it produces. This information is used by debuggers to step through your code, display the values of variables and so on. Debugging information makes your executables larger on disk and also opens the door for hackers to reverse-engineer your code, so it is always stripped from the final shipping version of your executable. However, during development, debugging information is invaluable and should always be included in your builds.

The compiler can also be told whether or not to expand *inline functions*. When inline function expansion is turned off, every inline function appears only once in memory, at a distinct address. This makes the task of tracing

通过调试器中的代码要简单得多，但显然是以牺牲通常通过内联实现的执行速度改进为代价的。

内联函数扩展只是广义代码转换（称为优化）的一个例子。编译器尝试优化代码的积极性及其使用的优化类型可以通过编译器选项进行控制。优化倾向于重新排序代码中的语句，还会导致变量从代码中完全剥离或移动，并且可能导致 CPU 寄存器在同一函数的后面被重新用于新用途。优化的代码通常会使大多数调试器感到困惑，导致它们以各种方式“欺骗”您，并使您难以或无法看到真正发生了什么。因此，在调试版本中通常会关闭所有优化。这允许仔细检查每个变量和每一行代码，就像它最初编码的那样。但是，当然，这样的代码运行速度会比完全优化的代码慢得多。

链接器设置

链接器还提供了多种选项。您可以控制要生成的输出文件类型——可执行文件或 DLL。您还可以指定要链接到可执行文件的外部库，以及要搜索哪些目录路径来查找它们。常见的做法是在构建调试可执行文件时链接调试库，而在非调试模式下构建时链接优化库。

链接器选项还控制诸如堆栈大小、程序在内存中的首选基地址、代码将在哪种类型的机器上运行（针对特定于机器的优化）、是否启用全局（链接时）优化以及其他一些我们在此不关心的细节。

2.2.4.2 局部和全局优化

优化编译器能够自动优化其生成的机器码。目前常用的所有 C/C++ 编译器都是优化编译器，包括 Microsoft Visual Studio、gcc、LLVM/Clang 和 Intel C/C++ 编译器。

优化可以有两种基本形式：

- 本地优化和
- 全局优化，

through the code in the debugger much simpler, but obviously comes at the expense of the execution speed improvements normally achieved by inlining.

Inline function expansion is but one example of generalized code transformations known as *optimizations*. The aggressiveness with which the compiler attempts to optimize your code, and the kinds of optimizations its uses, can be controlled via compiler options. Optimizations have a tendency to reorder the statements in your code, and they also cause variables to be stripped out of the code altogether, or moved around, and can cause CPU registers to be reused for new purposes later in the same function. Optimized code usually confuses most debuggers, causing them to “lie” to you in various ways, and making it difficult or impossible to see what’s really going on. As a result, all optimizations are usually turned off in a debug build. This permits every variable and every line of code to be scrutinized as it was originally coded. But, of course, such code will run much more slowly than its fully optimized counterpart.

Linker Settings

The linker also exposes a number of options. You can control what type of output file to produce—an executable or a DLL. You can also specify which external libraries should be linked into your executable, and which directory paths to search in order to find them. A common practice is to link with debug libraries when building a debug executable and with optimized libraries when building in non-debug mode.

Linker options also control things like stack size, the preferred base address of your program in memory, what type of machine the code will run on (for machine-specific optimizations), whether or not global (link-time) optimizations are enabled, and some other minutia with which we will not concern ourselves here.

2.2.4.2 Local and Global Optimizations

An optimizing compiler is one that can automatically optimize the machine code it generates. All of the C/C++ compilers commonly used today are optimizing compilers. These include Microsoft Visual Studio, gcc, LLVM/Clang and the Intel C/C++ compiler.

Optimizations can come in two basic forms:

- local optimizations and
- global optimizations,

尽管其他类型的优化也是可能的（例如，窥孔优化，它使优化器能够进行特定于平台或 CPU 的优化）。

局部优化仅针对称为基本块的小段代码进行操作。粗略地说，基本块是不涉及分支的汇编语言指令序列。局部优化包括以下内容：

- 代数简化，
- 运算符强度降低（例如，将 x / 2 转换为 x >> 1，因为移位运算符“强度较低”，因此比整数除法运算符成本更低），
- 代码内联，
- 常量折叠（识别编译时常量的表达式并用已知值替换这些表达式），
- 常量传播（用文字常量本身替换所有值为常量的变量实例），
- 循环展开（例如，将总是迭代四次的循环转换为循环内代码的四个副本，以消除条件分支），
- 死代码消除（删除没有效果的代码，例如删除赋值表达式 x = 5;，如果它后面紧跟着另一个对 x 的赋值，如 x = y + 1;），以及
- 指令重新排序（以尽量减少 CPU 管道停顿）。

全局优化的操作范围超出了基本代码块的范围——它们将程序的整个控制流图都考虑在内。这类优化的一个例子是公共子表达式消除。理想情况下，全局优化跨越翻译单元边界进行操作，因此由链接器而不是编译器执行。链接器执行的优化被称为链接时优化 (LTO)。

一些现代编译器（例如 LLVM/Clang）支持基于性能的优化 (PGO)。顾名思义，这些优化使用从软件先前运行中获得的分析信息，以迭代方式识别和优化其性能最关键的代码路径。PGO 和 LTO 优化可以带来显著的性能提升，但也存在成本。LTO 优化会大幅增加链接可执行文件所需的时间。而 PGO 优化本质上是迭代式的，需要运行软件（通过 QA 团队或自动化测试套件）才能生成驱动进一步优化的分析信息。

although other kinds of optimizations are possible (e.g., *peep-hole optimizations*, which enable the optimizer to make platform- or CPU-specific optimizations).

Local optimizations operate only on small chunks of code known as *basic blocks*. Roughly speaking, a basic block is a sequence of assembly language instructions that doesn't involve a branch. Local optimizations include things like:

- *algebraic simplification*,
- *operator strength reduction* (e.g., converting `x / 2` into `x >> 1` because the shift operator is "lower strength" and therefore less expensive than the integer division operator),
- *code inlining*,
- *constant folding* (recognizing expressions that are constant at compile time and replacing such expressions with their known value),
- *constant propagation* (replacing all instances of a variable whose value turns out to be constant with the literal constant itself),
- *loop unrolling* (e.g., converting a loop that always iterates exactly four times with four copies of the code within the loop, in order to eliminate conditional branching),
- *dead code elimination* (removal of code that has no effect, such as removing the assignment expression `x = 5;` if it is immediately followed by another assignment to `x` like `x = y + 1;`), and
- *instruction reordering* (in order to minimize CPU pipeline stalls).

Global optimizations operate beyond the scope of basic code blocks—they take the whole control flow graph of a program into account. An example of this kind of optimization would be *common sub-expression elimination*. Global optimizations ideally operate across translation unit boundaries, and hence are performed by the linker rather than the compiler. Aptly, optimizations performed by the linker are known as *link-time optimizations* (LTO).

Some modern compilers like LLVM/Clang support *profile-guided optimizations* (PGO). As the name implies, these optimizations use profiling information obtained from previous runs of your software to iteratively identify and optimize its most performance-critical code paths. PGO and LTO optimizations can produce an impressive performance boost, but they come with a cost. LTO optimizations greatly increase the amount of time required to link an executable. And PGO optimizations, being iterative in nature, require the software to be run (via the QA team or an automated test suite) in order to generate the profiling information that drives further optimizations.

大多数编译器都提供各种选项来控制其优化力度。您可以完全禁用优化（这对于调试版本非常有用，因为调试能力比性能更重要），或者可以提高优化的“级别”，直至达到预设的最大值。例如，gcc、Clang 和 Visual C++ 的优化级别范围从 -O0（表示不执行任何优化）到 -O3（启用所有优化）。您也可以通过其他命令行选项来启用或禁用单个优化。

2.2.4.3 典型的构建配置

游戏项目通常不止两种构建配置。以下是我在游戏开发中见过的一些常见配置。

• 调试。调试版本是程序的一个非常慢的版本，它关闭了所有优化，禁用了所有函数内联，并包含完整的调试信息。此版本用于测试全新代码，也用于调试开发过程中出现的所有（除了最细微的问题）问题。

• 开发版本。开发版本（或“dev build”）是程序的更快版本，启用了大部分或全部本地优化，但仍保留调试信息和断言。（有关断言的讨论，请参阅第 3.2.3.3 节。）这不仅能让您看到游戏以代表最终产品的速度运行，还能让您有机会调试问题。

• 发行版。发行版配置用于构建最终交付给客户的游戏。它有时也被称为“最终”版本或“磁盘”版本。与开发版本不同，发行版会剥离所有调试信息，删除大部分或所有断言，并进行全方位优化，包括全局优化（LTO 和 PGO）。发行版的调试非常棘手，但它是所有构建类型中速度最快、最精简的。

混合构建

混合构建是一种构建配置，其中大多数翻译单元在开发模式下构建，但其中一小部分在调试模式下构建。这使得当前正在审查的代码段可以轻松调试，而其余代码则继续以接近全速运行。

使用像 make 这样的基于文本的构建系统，可以很容易地设置混合构建，允许用户指定每个构建是否使用调试模式。

Most compilers offer various options controlling how aggressive its optimization efforts will be. Optimizations can be disabled entirely (useful for debug builds, where debuggability is more important than performance), or increasing “levels” of optimizations can be applied up to some predetermined maximum. For example, optimizations for gcc, Clang and Visual C++ all range from -O0 (meaning no optimizations are performed) to -O3 (all optimizations enabled). Individual optimizations can be turned on or off as well, via other command line options.

2.2.4.3 Typical Build Configurations

Game projects often have more than just two build configurations. Here are a few of the common configurations I've seen used in game development.

- *Debug*. A debug build is a very slow version of your program, with all optimizations turned off, all function inlining disabled, and full debugging information included. This build is used when testing brand new code and also to debug all but the most trivial problems that arise during development.
- *Development*. A development build (or “dev build”) is a faster version of your program, with most or all local optimizations enabled, but with debugging information and assertions still turned on. (See Section 3.2.3.3 for a discussion of assertions.) This allows you to see your game running at a speed representative of the final product, but it still gives you some opportunity to debug problems.
- *Ship*. A ship configuration is intended for building the final game that you will ship to your customers. It is sometimes called a “final” build or “disk” build. Unlike a development build, all debugging information is stripped out of a ship build, most or all assertions are compiled out, and optimizations are cranked all the way up, including global optimizations (LTO and PGO). A ship build is very tricky to debug, but it is the fastest and leanest of all build types.

Hybrid Builds

A hybrid build is a build configuration in which the majority of the translation units are built in development mode, but a small subset of them is built in debug mode. This permits the segment of code that is currently under scrutiny to be easily debugged, while the rest of the code continues to run at close to full speed.

With a text-based build system like `make`, it is quite easy to set up a hybrid build that permits users to specify the use of debug mode on a per-

翻译单元基础。简而言之，我们定义一个名为 $HYBRID_SOURCES 的 make 变量，它列出了混合构建中应在调试模式下编译的所有翻译单元（.cpp 文件）的名称。我们设置了用于编译每个翻译单元的调试和开发版本的构建规则，并将生成的目标文件（.obj/.o）放置在两个不同的文件夹中，一个用于调试，一个用于开发。最后的链接规则设置为与 $HYBRID_SOURCES 中列出的目标文件的调试版本以及所有其他目标文件的非调试版本进行链接。如果我们设置正确，make 的依赖规则将处理剩下的事情。

不幸的是，这在 Visual Studio 中并不容易，因为它的构建配置旨在基于每个项目应用，而不是每个翻译单元。问题的关键在于我们无法轻松定义要在调试模式下构建的翻译单元列表。解决此问题的一个方法是编写一个脚本（使用 Python 或其他合适的语言），该脚本会自动生成 Visual Studio .vcxproj 文件，并给出要在混合配置中以调试模式构建的 .cpp 文件列表。如果您的源代码已经组织成库，则另一种可行的替代方法是在解决方案级别设置“混合”构建配置，该配置根据每个项目（以及每个库）在调试和开发构建之间进行选择。这不像在每个翻译单元的基础上进行控制那样灵活，但如果您的库足够精细，它确实可以很好地工作。

构建配置和可测试性

项目支持的构建配置越多，测试就越困难。尽管各种配置之间的差异可能很小，但存在一个关键错误存在于其中一个配置中而其他配置中不存在的有限概率。因此，每个构建配置都必须进行同等彻底的测试。大多数游戏工作室不会正式测试其调试版本，因为调试配置主要用于功能初始开发期间的内部使用，以及调试在其他配置中发现的问题。但是，如果您的测试人员将大部分时间都花在测试开发版本配置上，那么您就不能简单地在黄金大师赛前一天晚上制作游戏的正式版本，并期望它具有与开发版本相同的错误配置文件。实际上，测试团队必须在 Alpha 和 Beta 测试期间对开发版本和正式版本进行同等测试，以确保正式版本中不会出现任何令人不快的意外。就可测试性而言，将构建配置保持在最低限度具有明显的优势，并且在

translation-unit basis. In a nutshell, we define a make variable called something like `$HYBRID_SOURCES`, which lists the names of all translation units (.cpp files) that should be compiled in debug mode for our hybrid build. We set up build rules for compiling both debug and development versions of every translation unit, and arrange for the resulting object files (.obj/.o) to be placed into two different folders, one for debug and one for development. The final link rule is set up to link with the debug versions of the object files listed in `$HYBRID_SOURCES` and with the non-debug versions of all other object files. If we've set it up properly, `make`'s dependency rules will take care of the rest.

Unfortunately, this is not so easy to do in Visual Studio, because its build configurations are designed to be applied on a per-project basis, not per-translation unit. The crux of the problem is that we cannot easily define a list of the translation units that we want to build in debug mode. One solution to this problem is to write a script (in Python or another suitable language) that automatically generates Visual Studio .vcxproj files, given a list of the .cpp files you want to be built in debug mode in your hybrid configuration. Another alternative that works if your source code is already organized into libraries is to set up a "Hybrid" build configuration at the solution level, which picks and chooses between debug and development builds on a per-project (and hence per-*library*) basis. This isn't as flexible as having control on a per-translation-unit basis, but it does work reasonably well if your libraries are sufficiently granular.

Build Configurations and Testability

The more build configurations your project supports, the more difficult testing becomes. Although the differences between the various configurations may be slight, there's a finite probability that a critical bug may exist in one of them but not in the others. Therefore, each build configuration must be tested equally thoroughly. Most game studios do not formally test their debug builds, because the debug configuration is primarily intended for internal use during initial development of a feature and for the debugging of problems found in one of the other configurations. However, if your testers spend most of their time testing your development build configuration, then you cannot simply make a ship build of your game the night before Gold Master and expect it to have an identical bug profile to that of the development build. Practically speaking, the test team must test both your development and ship builds equally throughout alpha and beta to ensure that there aren't any nasty surprises lurking in your ship build. In terms of testability, there is a clear advantage to keeping your build configurations to a minimum, and in

事实上，一些工作室由于这个原因没有单独的船舶建造 - 他们只是在经过彻底测试后才运送他们的开发版本（但删除了调试信息）。

2.2.4.4 项目配置教程

在解决方案资源管理器中右键单击任意项目，并从菜单中选择“属性...”，即可打开该项目的“属性页”对话框。左侧的树状视图显示了各种设置类别。其中，我们最常用的四个是：

- 配置属性/常规，
- 配置属性/调试，
- 配置属性/C++，以及
- 配置属性/链接器。

配置下拉组合框

请注意窗口左上角标有“配置：”的下拉组合框。这些属性页上显示的所有属性均分别应用于每个构建配置。如果您为调试配置设置了属性，并不一定意味着其他配置也存在相同的设置。

点击组合框下拉列表，您会发现可以选择单个配置或多个配置，包括“所有配置”。根据经验，尽量在选择“所有配置”的情况下完成大部分构建配置的编辑。这样，您就无需对每个配置进行多次相同的编辑，也不会因为意外设置错误而导致某个配置出现问题。但是，请注意，调试和开发配置中的某些设置需要有所不同。例如，函数内联和代码优化设置在调试和开发构建中当然应该有所不同。

通用属性页面

如图 2.9 所示，在常规属性页面上，最有用的字段如下：

- 输出目录。这定义了构建的最终产品将存放到哪里 —— 也就是编译器/链接器最终输出的可执行文件、库或 DLL。

fact some studios don't have a separate ship build for this reason—they simply ship their *development* build once it has been thoroughly tested (but with the debugging information stripped out).

2.2.4.4 Project Configuration Tutorial

Right-clicking on any project in the Solution Explorer and selecting "Properties…" from the menu brings up the project's "Property Pages" dialog. The tree view on the left shows various categories of settings. Of these, the four we will use most are:

- Configuration Properties/General,
- Configuration Properties/Debugging,
- Configuration Properties/C++, and
- Configuration Properties/Linker.

Configurations Drop-Down Combo Box

Notice the drop-down combo box labeled "Configuration:" at the top-left corner of the window. All of the properties displayed on these property pages apply separately to each build configuration. If you set a property for the debug configuration, this does not necessarily mean that the same setting exists for the other configuration(s).

If you click on the combo box to drop down the list, you'll find that you can select a single configuration or multiple configurations, including "All configurations." As a rule of thumb, try to do most of your build configuration editing with "All configurations" selected. That way, you won't have to make the same edits multiple times, once for each configuration—and you don't risk setting things up incorrectly in one of the configurations by accident. However, be aware that some settings *need* to be different between the debug and development configurations. For example, function inlining and code optimization settings should, of course, be different between debug and development builds.

General Property Page

On the *General* property page, shown in Figure 2.9, the most useful fields are the following:

- *Output directory*. This defines where the final product(s) of the build will go—namely, the executable, library or DLL that the compiler/linker ultimately outputs.

图 2.9. Visual Studio 项目属性页 - 常规页。

• 中间文件目录。这定义了构建过程中中间文件（主要是目标文件（.obj 扩展名））的存放位置。中间文件不会随最终程序一起发布——它们仅在构建可执行文件、库或 DLL 的过程中需要。因此，最好将中间文件与最终产品（.exe、.lib 或 .dll 文件）放在不同的目录中。

请注意，VisualStudio 提供了一个宏功能，可以在“项目属性页”对话框中指定目录和其他设置时使用。宏本质上是一个包含全局值的命名变量，可以在项目配置设置中引用。

宏的调用方式是将宏的名称括在括号中，并在其前面加上美元符号（例如，$(ConfigurationName)）。下面列出了一些常用的宏。

• $(TargetFileName) . 该项目构建的最终可执行文件、库或 DLL 文件的名称。

• $(TargetPath) . 包含最终可执行文件、库或 DLL 的文件夹的完整路径。

• $(ConfigurationName) . 构建配置的名称，在 Visual Studio 中默认为“Debug”或“Release”，但正如我们所说，一个真正的游戏项目可能会有多个配置，例如“Debug”、“Hybrid”、“Development”和“Ship”。

• $(OutDir).此对话框中指定的“输出目录”字段的值。

Figure 2.9. Visual Studio project property pages—General page.

- *Intermediate directory*. This defines where intermediate files, primarily object files (.obj extension), are placed during a build. Intermediate files are never shipped with your final program—they are only required during the process of building your executable, library or DLL. Hence, it is a good idea to place intermediate files in a different directory than the final products (.exe, .lib or .dll files).

Note that VisualStudio provides a macro facility, which may be used when specifying directories and other settings in the “Project Property Pages” dialog. A *macro* is essentially a named variable that contains a global value and that can be referred to in your project configuration settings.

Macros are invoked by writing the name of the macro enclosed in parentheses and prefixed with a dollar sign (e.g., `$(ConfigurationName)`). Some commonly used macros are listed below.

- `$(TargetFileName)`. The name of the final executable, library or DLL file being built by this project.
- `$(TargetPath)`. The full path of the folder containing the final executable, library or DLL.
- `$(ConfigurationName)`. The name of the build config, which will be “Debug” or “Release” by default in Visual Studio, although as we’ve said, a real game project will probably have multiple configurations such as “Debug,” “Hybrid,” “Development” and “Ship.”
- `$(OutDir)`. The value of the “Output Directory” field specified in this dialog.

- $(IntDir) . 此对话框中“中间目录”字段的值。

- $(VCInstallDir) . Visual Studio 的 C++ 标准库当前安装的目录。

使用宏而不是硬编码配置设置的好处是，只需更改全局宏的值，即可自动影响所有使用该宏的配置设置。此外，某些宏（例如 $(ConfigurationName)）会根据构建配置自动更改其值，因此使用它们可以让您在所有配置中使用相同的设置。

要查看所有可用宏的完整列表，请单击“常规”属性页上的“输出目录”字段或“中间目录”字段，单击文本字段右侧的小箭头，选择“编辑...”，然后单击出现的对话框中的“宏”按钮。

调试属性页

“调试”属性页用于指定要调试的可执行文件的名称和位置。在此页面上，您还可以指定在程序运行时应传递给程序的命令行参数。我们将在下文更深入地讨论如何调试程序。

C/C++ 属性页

C/C++ 属性页控制编译时语言设置，这些设置会影响源文件如何编译为目标文件（.obj 扩展名）。此页面上的设置不会影响目标文件如何链接到最终的可执行文件或 DLL。

我们鼓励您浏览 C/C++ 页面的各个子页面，了解有哪些可用的设置。一些最常用的设置包括：

- 常规属性页/附加包含目录。此字段列出了查找 #include 头文件时将搜索的磁盘目录。

  重要提示：最好使用相对路径和/或 Visual Studio 宏（例如 $(OutDir) 或 $(IntDir) ）来指定这些目录。这样，即使您将构建树移动到磁盘上的其他位置，或移动到另一台具有不同根文件夹的计算机，一切仍能正常工作。

- `$(IntDir)`. The value of the “Intermediate Directory” field in this dialog.
- `$(VCInstallDir)`. The directory in which Visual Studio’s C++ standard library is currently installed.

The benefit of using macros instead of hard-wiring your configuration settings is that a simple change of the global macro’s value will automatically affect all configuration settings in which the macro is used. Also, some macros like `$(ConfigurationName)` automatically change their values depending on the build configuration, so using them can permit you to use identical settings across all your configurations.

To see a complete list of all available macros, click in either the “Output Directory” field or the “Intermediate Directory” field on the “General” property page, click the little arrow to the right of the text field, select “Edit…” and then click the “Macros” button in the dialog that comes up.

Debugging Property Page

The “Debugging” property page is where the name and location of the executable to debug is specified. On this page, you can also specify the command-line argument(s) that should be passed to the program when it runs. We’ll discuss debugging your program in more depth below.

C/C++ Property Page

The C/C++ property page controls compile-time language settings—things that affect how your source files will be compiled into object files (.obj extension). The settings on this page do *not* affect how your object files are linked into a final executable or DLL.

You are encouraged to explore the various subpages of the C/C++ page to see what kinds of settings are available. Some of the most commonly used settings include the following:

- *General Property Page/Additional Include Directories*. This field lists the on-disk directories that will be searched when looking for `#included` header files.

  Important: It is always best to specify these directories using *relative paths* and/or with Visual Studio macros like `$(OutDir)` or `$(IntDir)`. That way, if you move your build tree to a different location on disk or to another computer with a different root folder, everything will continue to work properly.

• 常规属性页/调试信息格式。此字段控制是否生成调试信息以及生成格式。通常，调试和开发配置都包含调试信息，以便您在游戏开发过程中追踪问题。为了防止黑客入侵，飞船版本会删除所有调试信息。

• 预处理器属性页/预处理器定义。这个非常方便的字段列出了编译代码时应定义的任意数量的 C/C++ 预处理器符号。有关预处理器定义符号的讨论，请参见第 2.2.4.1 节中的“预处理器设置”。

链接器属性页

“链接器”属性页列出了一些属性，这些属性会影响目标代码文件如何链接到可执行文件或 DLL。同样，建议您探索各个子页面。一些常用的设置如下：

• 常规属性页/输出文件。此设置列出了构建的最终产品的名称和位置，通常是可执行文件或 DLL。

• 常规属性页/附加库目录。与 C/C++ 附加包含目录字段非常相似，此字段列出了在查找要链接到最终可执行文件的库和目标文件时将搜索的零个或多个目录。

• 输入属性页/附加依赖项。此字段列出了要链接到可执行文件或 DLL 的外部库。例如，如果您正在构建启用 OGRE 的应用程序，则 OGRE 库将列在此处。

请注意，Visual Studio 使用各种“魔法”来指定应链接到可执行文件的库。例如，源代码中可以使用特殊的 #pragma 指令指示链接器自动链接到特定库。因此，您可能无法在“附加依赖项”字段中看到所有实际链接到的库。（事实上，这就是它们被称为附加依赖项的原因。）例如，您可能已经注意到，Direct X 应用程序不会在其“附加依赖项”字段中手动列出所有 DirectX 库。现在您知道原因了。

2.2.5 调试代码

任何程序员都应该学习的最重要的技能之一就是如何有效地调试代码。本节提供了一些有用的调试技巧和

- *General Property Page/Debug Information Format*. This field controls whether or not debug information is generated and in what format. Typically both debug and development configurations include debugging information so that you can track down problems during the development of your game. The ship build will have all the debug info stripped out to prevent hacking.
- *Preprocessor Property Page/Preprocessor Definitions*. This very handy field lists any number of C/C++ preprocessor symbols that should be defined in the code when it is compiled. See *Preprocessor Settings* in Section 2.2.4.1 for a discussion of preprocessor-defined symbols.

Linker Property Page

The “Linker” property page lists properties that affect how your object code files will be linked into an executable or DLL. Again, you are encouraged to explore the various subpages. Some commonly used settings follow:

- *General Property Page/Output File*. This setting lists the name and location of the final product of the build, usually an executable or DLL.
- *General Property Page/Additional Library Directories*. Much like the C/C++ Additional Include Directories field, this field lists zero or more directories that will be searched when looking for libraries and object files to link into the final executable.
- *Input Property Page/Additional Dependencies*. This field lists external libraries that you want linked into your executable or DLL. For example, the OGRE libraries would be listed here if you are building an OGRE-enabled application.

Note that Visual Studio employs various “magic spells” to specify libraries that should be linked into an executable. For example, a special `#pragma` instruction in your source code can be used to instruct the linker to automatically link with a particular library. For this reason, you may not see all of the libraries you're actually linking to in the “Additional Dependencies” field. (In fact, that's why they are called *additional* dependencies.) You may have noticed, for example, that Direct X applications do not list all of the DirectX libraries manually in their “Additional Dependencies” field. Now you know why.

2.2.5 Debugging Your Code

One of the most important skills any programmer can learn is how to effectively debug code. This section provides some useful debugging tips and

这些技巧。有些适用于任何调试器，有些则特定于 Microsoft Visual Studio。不过，您通常可以在其他调试器中找到与 Visual Studio 调试功能等效的功能，因此即使您不使用 Visual Studio 调试代码，本节也应该很有用。

2.2.5.1 启动项目

Visual Studio 解决方案可以包含多个项目。其中一些项目用于构建可执行文件，而另一些项目则用于构建库或 DLL。单个解决方案中可以包含多个用于构建可执行文件的项目。Visual Studio 提供了一个称为“启动项目”的设置。对于调试器而言，该项目被视为“当前”项目。通常，程序员会通过设置单个启动项目来一次调试一个项目。但是，也可以同时调试多个项目（有关详细信息，请参阅 http://msdn.microsoft.com/en-us/library/0s590bew(v=vs.100).aspx）。

启动项目在解决方案资源管理器中以粗体突出显示。默认情况下，如果启动项目生成了可执行文件，则按下 F5 将运行由启动项目生成的 .exe 文件。（从技术上讲，F5 会运行你在“调试”属性页的“命令”字段中输入的任何命令，因此它并不局限于运行你的项目生成的 .exe 文件。）

2.2.5.2 断点

断点是代码调试的基础。断点指示程序在源代码的特定行停止，以便您检查正在发生的事情。

在 Visual Studio 中，选择一行并按 F9 键切换断点。运行程序时，如果包含断点的代码行即将执行，调试器将停止程序。我们称该断点为“命中”。一个小箭头会显示 CPU 的程序计数器当前位于哪一行代码。如图 2.10 所示。

2.2.5.3 单步执行代码

一旦设置了断点，您就可以按 F10 键单步执行代码。黄色的程序计数器箭头会移动，显示执行的行数。按 F11 键单步执行函数调用（即，您看到的下一行代码是被调用函数的第一行），而按 F10 键单步跳过该函数调用（即，调试器全速调用该函数，然后在调用后紧接着的行再次断点）。

tricks. Some are applicable to any debugger and some are specific to Microsoft Visual Studio. However, you can usually find an equivalent to Visual Studio's debugging features in other debuggers, so this section should prove useful even if you don't use Visual Studio to debug your code.

2.2.5.1 The Start-Up Project

A Visual Studio solution can contain more than one project. Some of these projects build executables, while others build libraries or DLLs. It's possible to have more than one project that builds an executable in a single solution. Visual Studio provides a setting known as the "Start-Up Project." This is the project that is considered "current" for the purposes of the debugger. Typically a programmer will debug one project at a time by setting a single start-up project. However, it is possible to debug multiple projects simultaneously (see http://msdn.microsoft.com/en-us/library/0s590bew(v=vs.100).aspx for details).

The start-up project is highlighted in bold in the Solution Explorer. By default, hitting F5 will run the .exe built by the start-up project, if the start-up project builds an executable. (Technically speaking, F5 runs whatever command you type into the Command field in the Debugging property page, so it's not limited to running the .exe built by your project.)

2.2.5.2 Breakpoints

Breakpoints are the bread and butter of code debugging. A breakpoint instructs the program to stop at a particular line in your source code so that you can inspect what's going on.

In Visual Studio, select a line and hit F9 to toggle a breakpoint. When you run your program and the line of code containing the breakpoint is about to be executed, the debugger will stop the program. We say that the breakpoint has been "hit." A little arrow will show you which line of code the CPU's program counter is currently on. This is shown in Figure 2.10.

2.2.5.3 Stepping through Your Code

Once a breakpoint has been hit, you can single-step your code by hitting the F10 key. The yellow program-counter arrow moves to show you the lines as they execute. Hitting F11 steps *into* a function call (i.e., the next line of code you'll see is the first line of the called function), while F10 steps *over* that function call (i.e., the debugger calls the function at full speed and then breaks again on the line right after the call).

图 2.10. 在 Visual Studio 中设置断点。

图 2.11. 调用堆栈窗口。

2.2.5.4 调用堆栈

调用堆栈窗口（如图 2.11 所示）显示了代码执行过程中任何特定时刻调用的函数堆栈。（有关程序堆栈的更多详细信息，请参阅第 3.3.5.2 节。）要显示调用堆栈（如果尚未显示），请转到主菜单栏上的“调试”菜单，选择“窗口”，然后选择“调用堆栈”。

一旦遇到断点（或程序手动暂停），您可以通过双击“调用堆栈”窗口中的条目来上下移动调用堆栈。这对于检查 main() 和当前代码行之间的函数调用链非常有用。例如，您可以追溯到父函数中某个 bug 的根本原因，该 bug 出现在深度嵌套的子函数中。

2.2.5.5 监视窗口

在单步执行代码并在调用堆栈中上下移动时，您需要能够检查程序中变量的值。这就是监视窗口的作用。要打开监视窗口，请转到“调试”菜单，选择“窗口...”，然后选择“监视...”，最后选择以下选项之一：

Figure 2.10. Setting a breakpoint in Visual Studio.

Figure 2.11. The call stack window.

2.2.5.4 The Call Stack

The *call stack* window, shown in Figure 2.11, shows you the stack of functions that have been called at any given moment during the execution of your code. (See Section 3.3.5.2 for more details on a program's stack.) To display the call stack (if it is not already visible), go to the “Debug” menu on the main menu bar, select “Windows” and then “Call Stack.”

Once a breakpoint has been hit (or the program is manually paused), you can move up and down the call stack by double-clicking on entries in the “Call Stack” window. This is very useful for inspecting the chain of function calls that were made between `main()` and the current line of code. For example, you might trace back to the root cause of a bug in a parent function that has manifested itself in a deeply nested child function.

2.2.5.5 The Watch Window

As you step through your code and move up and down the call stack, you will want to be able to inspect the values of the variables in your program. This is what *watch windows* are for. To open a watch window, go to the “Debug” menu, select “Windows…,” then select “Watch…,” and finally select one of

图 2.12. Visual Studio 的监视窗口。

“监视 1”到“监视 4”。（Visual Studio 允许您同时打开最多四个监视窗口。）打开监视窗口后，您可以在窗口中输入变量名称或直接从源代码中拖入表达式。

如图 2.12 所示，简单数据类型的变量会将其值列在其名称的右侧。复杂数据类型则以小型树状视图的形式显示，可以轻松展开以“深入”查看几乎任何嵌套结构。类的基类始终显示为派生类实例的第一个子类。这样，您不仅可以检查类的数据成员，还可以检查其基类的数据成员。

您可以在监视窗口中输入几乎任何有效的 C/C++ 表达式，Visual Studio 都会计算该表达式并尝试显示结果值。例如，您可以输入“ 5 + 3 ”，Visual Studio 将显示“8”。您可以使用 C 或 C++ 类型转换语法将变量从一种类型转换为另一种类型。例如，在监视窗口中输入“ (float)intVar1/(float)intVar2 ”将以浮点值的形式显示两个整数变量的比率。

您甚至可以从监视窗口内调用程序中的函数。
Visual Studio 会自动重新评估在监视窗口中输入的表达式，因此，如果您在监视窗口中调用某个函数，则每次遇到断点或单步执行代码时都会调用该函数。这允许您利用程序的功能，在尝试解释调试器中检查的数据时节省工作量。例如，假设您的游戏引擎提供了一个名为 quatToAngleDeg() 的函数，它将四元数转换为以度为单位的旋转角度。您可以在监视窗口中调用此函数，以便轻松地

Figure 2.12. Visual Studio's watch window.

“Watch 1” through “Watch 4.” (Visual Studio allows you to open up to four watch windows simultaneously.) Once a watch window is open, you can type the names of variables into the window or drag expressions in directly from your source code.

As you can see in Figure 2.12, variables with simple data types are shown with their values listed immediately to the right of their names. Complex data types are shown as little tree views that can be easily expanded to “drill down” into virtually any nested structure. The base class of a class is always shown as the first child of an instance of a derived class. This allows you to inspect not only the class' data members, but also the data members of its base class(es).

You can type virtually any valid C/C++ *expression* into the watch window, and Visual Studio will evaluate that expression and attempt to display the resulting value. For example, you could type “`5 + 3`” and Visual Studio will display “8.” You can cast variables from one type to another by using C or C++ casting syntax. For example, typing “`(float)intVar1/(float)intVar2`” in the watch window will display the ratio of two integer variables as a floating-point value.

You can even *call functions in your program* from within the watch window. Visual Studio reevaluates the expressions typed into the watch window(s) automatically, so if you invoke a function in the watch window, it will be called every time you hit a breakpoint or single-step your code. This allows you to leverage the functionality of your program in order to save yourself work when trying to interpret the data that you're inspecting in the debugger. For example, let's say that your game engine provides a function called `quatToAngleDeg()`, which converts a quaternion to an angle of rotation in degrees. You can call this function in the watch window in order to easily

检查调试器中任何四元数的旋转角度。
您还可以在监视窗口中的表达式上使用各种后缀，以更改 Visual Studio 显示数据的方式，如图 2.13 所示。

- “,d”后缀强制以十进制显示值。
- “,x”后缀强制以十六进制表示法显示值。
- “ , n ”后缀（其中 n 为任意正整数）强制 Visual Studio 将值视为包含 n 个元素的数组。这允许您扩展通过指针引用的数组数据。
- 您还可以在方括号中编写简单的表达式，计算“,n”后缀中n的值。例如，您可以输入如下内容：

```
my_array,[my_array_count]
```

要求调试器显示名为 my_array 的数组的 my_array_count 个元素。

在监视窗口中扩展非常大的数据结构时要小心，因为它有时会使调试器的速度变慢到无法使用的程度。

2.2.5.6 数据断点

当 CPU 的程序计数器到达特定的机器指令或代码行时，常规断点就会触发。然而，现代调试器的另一个非常有用的功能是能够设置断点，当特定内存地址被写入（即更改）时，断点就会触发。这些断点被称为 数据断点 ，因为它们是由数据更改触发的；有时也被称为 硬件断点 ，因为它们是通过 CPU 硬件的一项特殊功能实现的——即在预定义的内存地址被写入时触发中断的能力。

图 2.13。Visual Studio 监视窗口中的逗号后缀。

inspect the rotation angle of any quaternion within the debugger.

You can also use various suffixes on the expressions in the watch window in order to change the way Visual Studio displays the data, as shown in Figure 2.13.

- The “ , d” suffix forces values to be displayed in decimal notation.
- The “ , x” suffix forces values to be displayed in hexadecimal notation.
- The “ , *n*” suffix (where *n* is any positive integer) forces Visual Studio to treat the value as an array with *n* elements. This allows you to expand array data that is referenced through a pointer.
- You can also write simple expressions in square brackets that *calculate* the value of *n* in a “ , *n*” suffix. For example, you can type something like this:

```
my_array,[my_array_count]
```

to ask the debugger to show `my_array_count` elements of the array named `my_array`.

Be careful when expanding very large data structures in the watch window, because it can sometimes slow the debugger down to the point of being unusable.

2.2.5.6 Data Breakpoints

Regular breakpoints trip when the CPU’s program counter hits a particular machine instruction or line of code. However, another incredibly useful feature of modern debuggers is the ability to set a breakpoint that trips whenever a specific memory address is *written to* (i.e., changed). These are called *data breakpoints*, because they are triggered by changes to data, or sometimes *hardware breakpoints*, because they are implemented via a special feature of the CPU’s hardware—namely, the ability to raise an interrupt when a predefined memory address is written to.

Figure 2.13. Comma suffixes in the Visual Studio watch window.

图 2.14. Visual Studio 断点窗口。

图 2.15. 定义数据断点。

数据断点通常如下使用。假设您正在追踪一个错误，该错误表现为一个零（0.0f）值神秘地出现在特定对象 m_angle 的成员变量中，而该对象应始终包含非零角度。您不知道哪个函数可能将那个零写入了您的变量。但是，您知道该变量的地址。（您只需在监视窗口中输入“&object.m_angle”即可找到其地址。）要追踪罪魁祸首，您可以在 object.m_angle 的地址上设置一个数据断点，然后让程序运行。当值发生变化时，调试器将自动停止。然后，您可以检查调用堆栈以当场捕获有问题的函数。

要在 Visual Studio 中设置数据断点，请执行以下步骤。

- 打开“Windows”下“Debug”菜单中的“Breakpoints”窗口，然后选择“Breakpoints”（图 2.14）。
- 选择窗口左上角的“新建”下拉按钮。
- 选择“新建数据断点”。
- 输入原始地址或地址值表达式，例如“&myVariable”（图 2.15）。

2.2.5.7 条件断点

您还会注意到，在“断点”窗口中，您可以设置任何类型的断点（数据断点或常规代码行断点）的条件和命中次数。

条件断点会导致调试器每次遇到断点时都会执行您提供的 C/C++ 表达式。如果表达式为真，调试器会停止程序，让您有机会查看发生了什么。如果表达式为假，则断点会被忽略，程序会继续运行。这对于设置仅在某个类的特定实例上调用函数时触发的断点非常有用。例如：

Figure 2.14. The Visual Studio breakpoints window.

Figure 2.15. Defining a data breakpoint.

Here's how data breakpoints are typically used. Let's say you are tracking down a bug that manifests itself as a zero (`0.0f`) value mysteriously appearing inside a member variable of a particular object called `m_angle` that *should* always contain a nonzero angle. You have no idea which function might be writing that zero into your variable. However, you do know the address of the variable. (You can just type "`&object.m_angle`" into the watch window to find its address.) To track down the culprit, you can set a data breakpoint on the address of `object.m_angle`, and then simply let the program run. When the value changes, the debugger will stop automatically. You can then inspect the call stack to catch the offending function red-handed.

To set a data breakpoint in Visual Studio, take the following steps.

- Bring up the "Breakpoints" window found on the "Debug" menu under "Windows" and then "Breakpoints" (Figure 2.14).
- Select the "New" drop-down button in the upper-left corner of the window.
- Select "New Data Breakpoint."
- Type in the raw address or an address-valued expression, such as "`&myVariable`" (Figure 2.15).

2.2.5.7 Conditional Breakpoints

You'll also notice in the "Breakpoints" window that you can set conditions and hit counts on any type breakpoint—data breakpoints or regular line-of-code breakpoints.

A *conditional breakpoint* causes the debugger to evaluate the C/C++ expression you provide every time the breakpoint is hit. If the expression is true, the debugger stops your program and gives you a chance to see what's going on. If the expression is false, the breakpoint is ignored and the program continues. This is very useful for setting breakpoints that only trip when a function is called on a particular instance of a class. For example,

假设你的游戏级别上有 20 辆坦克，当第三辆坦克到达时，你想停止你的程序，因为你知道它的内存地址是
0x12345678，正在运行。通过将断点的条件表达式设置为类似“(uintptr_t)this == 0x12345678”的形式，可以将断点限制在内存地址（this 指针）为 0x12345678 的类实例上。

指定断点的命中次数会导致调试器在每次断点命中时递减计数器，并且只有当计数器达到零时才会真正停止程序。这在断点位于循环内部，并且需要检查循环第 376 次迭代（例如，数组中的第 376 个元素）期间发生的情况时非常有用。您不可能坐在那里按 375 次 F5 键！但是，您可以让 Visual Studio 的命中次数功能为您完成这项工作。

需要注意的是：条件断点会导致调试器在每次遇到断点时评估条件表达式，因此它们会降低调试器和游戏的性能。

2.2.5.8 调试优化构建

我上面提到过，使用开发或交付版本调试问题可能非常棘手，这主要是由于编译器优化代码的方式。理想情况下，每个程序员都希望在调试版本中完成所有调试工作。然而，这通常是不可能的。有时，一个错误很少发生，你会抓住任何机会调试它，即使它发生在别人机器上的非调试版本中。其他错误只发生在你的非调试版本中，但只要你运行调试版本，它们就会神奇地消失。这些可怕的非调试错误有时是由未初始化的变量引起的，因为变量和动态分配的内存块在调试模式下通常设置为零，但在非调试版本中却包含垃圾。非调试错误的其他常见原因包括非调试版本中意外遗漏的代码（例如，当重要代码被错误地放置在断言语句中时）、调试和开发/发布版本之间大小或数据成员打包发生变化的数据结构、仅由内联或编译器引入的优化触发的错误，以及（在极少数情况下）编译器优化器本身的错误，导致它在完全优化的版本中发出不正确的代码。

显然，每个程序员都应该具备在非调试版本中调试问题的能力，尽管这看起来可能不太愉快。减轻调试优化代码痛苦的最佳方法是多加练习，并抓住机会拓展这方面的技能。以下是一些技巧。

let's say you have a game level with 20 tanks on-screen, and you want to stop your program when the third tank, whose memory address you know to be 0x12345678, is running. By setting the breakpoint's condition expression to something like "`(uintptr_t)this == 0x12345678`", you can restrict the breakpoint only to the class instance whose memory address (`this` pointer) is 0x12345678.

Specifying a *hit count* for a breakpoint causes the debugger to decrement a counter every time the breakpoint is hit, and only actually stop the program when that counter reaches zero. This is really useful for situations where your breakpoint is inside a loop, and you need to inspect what's happening during the 376th iteration of the loop (e.g., the 376th element in an array). You can't very well sit there and hit the F5 key 375 times! But you *can* let the hit count feature of Visual Studio do it for you.

One note of caution: conditional breakpoints cause the debugger to evaluate the conditional expression every time the breakpoint is hit, so they can bog down the performance of the debugger and your game.

2.2.5.8 Debugging Optimized Builds

I mentioned above that it can be very tricky to debug problems using a development or ship build, due primarily to the way the compiler optimizes the code. Ideally, every programmer would prefer to do all of his or her debugging in a debug build. However, this is often not possible. Sometimes a bug occurs so rarely that you'll jump at any chance to debug the problem, even if it occurs in a non-debug build on someone else's machine. Other bugs only occur in your non-debug builds, but they magically disappear whenever you run the debug build. These dreaded *non-debug-only bugs* are sometimes caused by uninitialized variables, because variables and dynamically allocated memory blocks are often set to zero in debug mode but are left containing garbage in a non-debug build. Other common causes of non-debug-only bugs include code that has been accidentally omitted from the non-debug build(s) (e.g., when important code is erroneously placed inside an assertion statement), data structures whose size or data member packing changes between debug and development/ship builds, bugs that are only triggered by inlining or compiler-introduced optimizations, and (in rare cases) bugs in the compiler's optimizer itself, causing it to emit incorrect code in a fully optimized build.

Clearly, it behooves every programmer to be capable of debugging problems in a non-debug build, unpleasant as it may seem. The best ways to reduce the pain of debugging optimized code is to practice doing it and to expand your skill set in this area whenever you have the opportunity. Here are a few tips.

• 学习在调试器中阅读和单步执行反汇编代码。在非调试版本中，调试器通常难以跟踪当前正在执行的源代码行。由于指令重新排序，在源代码模式下查看时，您经常会看到程序计数器在函数内不规则地跳动。但是，当您在反汇编模式下处理代码（即，逐行执行汇编语言指令）时，一切就会恢复正常。每个 C/C++ 程序员都应该至少对其目标 CPU 的架构和汇编语言有所了解。这样，即使调试器让您感到困惑，您也不会感到困惑。（有关汇编语言的介绍，请参阅第 3.4.7.3 节。）

• 使用寄存器推断变量的值或地址。调试器有时无法在非调试版本中显示变量的值或对象的内容。但是，如果程序计数器与变量的初始使用位置相差不大，则很有可能它的地址或值仍存储在 CPU 的某个寄存器中。如果您可以通过反汇编追溯到变量首次加载到寄存器的位置，通常可以通过检查该寄存器来发现它的值或地址。使用寄存器窗口或在监视窗口中输入寄存器的名称来查看其内容。

• 通过地址检查变量和对象内容。给定变量或数据结构的地址，通常可以通过在监视窗口中将地址转换为适当的类型来查看其内容。例如，如果我们知道 Foo 类的实例位于地址 0x1378A0C0，我们可以在监视窗口中输入“ (Foo*)0x1378A0C0 ”，调试器会将该内存地址解释为指向 Foo 对象的指针。

• 利用静态变量和全局变量。即使在优化版本中，调试器通常也可以检查全局变量和静态变量。如果您无法推断出变量或对象的地址，请留意可能直接或间接包含其地址的静态变量或全局变量。例如，如果我们想查找物理系统内部对象的地址，我们可能会发现它实际上存储在全局 PhysicsWorld 对象的成员变量中。

• 修改代码。如果您可以相对轻松地重现非调试型错误，请考虑修改源代码以帮助您调试问题。添加打印语句，以便了解发生了什么。引入全局变量，以便更轻松地检查有问题的变量或

- *Learn to read and step through disassembly in the debugger*. In a non-debug build, the debugger often has trouble keeping track of which line of source code is currently being executed. Thanks to instruction reordering, you'll often see the program counter jump around erratically within the function when viewed in source code mode. However, things become sane again when you work with the code in disassembly mode (i.e., step through the assembly language instructions individually). Every C/C++ programmer should be at least a little bit familiar with the architecture and assembly language of their target CPU(s). That way, even if the debugger is confused, you won't be. (For an introduction to assembly language, see Section 3.4.7.3.)

- *Use registers to deduce variables' values or addresses*. The debugger will sometimes be unable to display the value of a variable or the contents of an object in a non-debug build. However, if the program counter is not too far away from the initial use of the variable, there's a good chance its address or value is still stored in one of the CPU's registers. If you can trace back through the disassembly to where the variable is first loaded into a register, you can often discover its value or its address by inspecting that register. Use the register window, or type the name of the register into a watch window, to see its contents.

- *Inspect variables and object contents by address*. Given the address of a variable or data structure, you can usually see its contents by casting the address to the appropriate type in a watch window. For example, if we know that an instance of the `Foo` class resides at address 0x1378A0C0, we can type "`(Foo*)0x1378A0C0`" in a watch window, and the debugger will interpret that memory address as if it were a pointer to a `Foo` object.

- *Leverage static and global variables*. Even in an optimized build, the debugger can usually inspect global and static variables. If you cannot deduce the address of a variable or object, keep your eye open for a static or global that might contain its address, either directly or indirectly. For example, if we want to find the address of an internal object within the physics system, we might discover that it is in fact stored in a member variable of the global `PhysicsWorld` object.

- *Modify the code*. If you can reproduce a non-debug-only bug relatively easily, consider modifying the source code to help you debug the problem. Add print statements so you can see what's going on. Introduce a global variable to make it easier to inspect a problematic variable or

调试器中的对象。添加代码来检测问题情况或隔离某个类的特定实例。

2.3 分析工具

游戏通常是高性能的实时程序。因此，游戏引擎程序员一直在寻找提升代码速度的方法。在本节中，我们将探讨一些可用于测量软件性能的工具。我们将在第四章详细介绍如何使用这些性能分析数据来优化软件。

有一条众所周知的、尽管不太科学的经验法则，称为帕累托原则（请参阅 http://en.wikipedia.org/wiki/Pareto_principle）。它也被称为 80/20 规则，因为它指出，在很多情况下，某个事件的 80% 影响仅来自 20% 的可能原因。在计算机科学中，此原则既适用于错误修复（只需修复 20% 的代码中的错误即可消除软件中 80% 的已知错误），也适用于软件优化，根据经验法则，运行任何软件所花费的 80%（或更多）的挂钟时间仅由 20%（或更少）的代码造成。换句话说，如果您优化 20% 的代码，您就有可能实现执行速度提升的 80%。

那么，如何知道代码中哪 20% 需要优化呢？为此，您需要一个分析器。分析器是一种测量代码执行时间的工具。它可以告诉您每个函数花费了多少时间。然后，您就可以将优化重点放在那些占用大部分执行时间的函数上。

一些性能分析器还会显示每个函数被调用的次数。这是一个需要理解的重要维度。一个函数耗费时间的原因有两个：(a) 它本身执行起来需要很长时间，或者 (b) 它被频繁调用。例如，一个运行 A* 算法来计算游戏世界中最佳路径的函数，可能每帧只会被调用几次。

如果您使用合适的分析器，甚至可以获得更多信息。一些分析器会报告调用图，这意味着对于任何给定的函数，您可以看到哪些函数调用了它（这些函数称为 父函数 ），以及它调用了哪些函数（这些函数称为 子函数 或 后代 ）。您甚至可以看到该函数调用其每个后代函数所花费的时间百分比，以及每个函数占总运行时间的百分比。

object in the debugger. Add code to detect a problem condition or to isolate a particular instance of a class.

2.3 Profiling Tools

Games are typically high-performance real-time programs. As such, game engine programmers are always looking for ways to speed up their code. In this section, we'll investigate some of the tools we can use to measure the performance of our software. We'll have more to say about how to use this profiling data in order to optimize our software in Chapter 4.

There is a well-known, albeit rather unscientific, rule of thumb known as the *Pareto principle* (see http://en.wikipedia.org/wiki/Pareto_principle). It is also known as the *80/20 rule*, because it states that in many situations, 80% of the effects of some event come from only 20% of the possible causes. In computer science, this principle has been applied both to bug-fixing (80% of the perceived bugs in a piece of software can be eliminated by fixing bugs in only 20% of the code), and to software optimization, where as a rule of thumb 80% (or more) of the wall clock time spent running any piece of software is accounted for by only 20% (or less) of the code. In other words, if you optimize 20% of your code, you can potentially realize 80% of all the gains in execution speed you'll ever realize.

So, how do you know *which* 20% of your code to optimize? For that, you need a *profiler*. A profiler is a tool that measures the execution time of your code. It can tell you how much time is spent in each function. You can then direct your optimizations toward only those functions that account for the lion's share of the execution time.

Some profilers also tell you how many *times* each function is called. This is an important dimension to understand. A function can eat up time for two reasons: (a) it takes a long time to execute on its own, or (b) it is called frequently. For example, a function that runs an A* algorithm to compute the optimal paths through the game world might only be called a few times each frame

Even more information can be obtained if you use the right profiler. Some profilers report the call graph, meaning that for any given function, you can see which functions called it (these are known as *parent functions*) and which functions it called (these are known as *child functions* or *descendants*). You can even see what *percentage* of the function's time was spent calling each of its descendants and the percentage of the overall running time accounted for by each individual function.

分析器分为两大类。

1. 统计分析器。这类分析器的设计理念是非侵入式的，这意味着无论是否启用分析功能，目标代码的运行速度几乎相同。这类分析器的工作原理是定期对 CPU 的程序计数器寄存器进行采样，并记录当前正在运行的函数。每个函数的采样次数可以得出该函数所占用的总运行时间的大致百分比。英特尔的 VTune 是采用英特尔奔腾处理器的 Windows 系统统计分析器的黄金标准，现在也适用于 Linux。详情请参阅 https://software.intel.com/en-us/intel-vtune-amplifier-xe。

2. 仪表分析器。这种分析器旨在提供尽可能准确、全面的时间数据，但代价是目标程序的实时执行——当打开分析功能时，目标程序通常会变得非常慢。这些分析器的工作原理是预处理可执行文件，并在每个函数中插入特殊的序言和尾声代码。序言和尾声代码调用分析库，分析库依次检查程序的调用堆栈并记录各种详细信息，包括哪个父函数调用了相关函数以及该父函数调用了子函数的次数。这种分析器甚至可以设置为监视源程序中的每一行代码，从而允许它报告每行代码的执行时间。结果非常准确和全面，但打开分析功能可能会使游戏几乎无法玩。IBM 的 Rational Quantify 是 Rational Purify Plus 工具套件的一部分，是一款出色的仪表分析器。有关使用 Quantify 进行分析的介绍，请参阅 http://www.ibm.com/developerworks/rational/library/957.html。

微软还发布了一款融合了这两种方法的分析器，名为 LOP，即低开销分析器 (low-overhead profiler)。LOP 采用统计方法，定期对处理器状态进行采样，这意味着它对程序执行速度的影响较小。然而，每次采样时，它都会分析调用堆栈，从而确定导致每个采样的父函数链。这使得 LOP 能够提供统计分析器通常无法提供的信息，例如父函数之间的调用分布。

在 PlayStation 4 上，SN Systems 的 Razor CPU 是测量在 PS4 CPU 上运行的游戏软件的首选分析器。它支持以下两种统计方式：

Profilers fall into two broad categories.

1. *Statistical profilers*. This kind of profiler is designed to be unobtrusive, meaning that the target code runs at almost the same speed, whether or not profiling is enabled. These profilers work by sampling the CPU's program counter register periodically and noting which function is currently running. The number of samples taken within each function yields an approximate percentage of the total running time that is eaten up by that function. Intel's *VTune* is the gold standard in statistical profilers for Windows machines employing Intel Pentium processors, and it is now also available for Linux. See https://software.intel.com/en-us/intel-vtune-amplifier-xe for details.

2. *Instrumenting profilers*. This kind of profiler is aimed at providing the most accurate and comprehensive timing data possible, but at the expense of real-time execution of the target program—when profiling is turned on, the target program usually slows to a crawl. These profilers work by preprocessing your executable and inserting special prologue and epilogue code into every function. The prologue and epilogue code calls into a profiling library, which in turn inspects the program's call stack and records all sorts of details, including which parent function called the function in question and how many times that parent has called the child. This kind of profiler can even be set up to monitor every line of code in your source program, allowing it to report how long each line is taking to execute. The results are stunningly accurate and comprehensive, but turning on profiling can make a game virtually unplayable. IBM's Rational Quantify, available as part of the Rational Purify Plus tool suite, is an excellent instrumenting profiler. See http://www.ibm.com/developerworks/rational/library/957.html for an introduction to profiling with Quantify.

Microsoft has also published a profiler that is a hybrid between the two approaches. It is called LOP, which stands for low-overhead profiler. LOP uses a statistical approach, sampling the state of the processor periodically, which means it has a low impact on the speed of the program's execution. However, with each sample, it analyzes the call stack, thereby determining the chain of parent functions that resulted in each sample. This allows LOP to provide information normally not available with a statistical profiler, such as the distribution of calls across parent functions.

On the PlayStation 4, SN Systems' *Razor CPU* is the profiler of choice for measuring game software running on the PS4's CPU. It supports both statis-

tical 和 instrumenting 分析方法。（有关更多详细信息，请参阅 https://www.snsystems. com/tech-blog/2014/02/14/function-level-profiling/。）其对应产品 Razor GPU 为在 PS4 GPU 上运行的着色器和计算作业提供分析和调试功能。

2.3.1 分析器列表

市面上有很多可用的性能分析工具。请参阅 http://en.wikipedia.org/wiki/List_of_performance_analysis_tool 获取一个相当全面的列表。

2.4 内存泄漏和损坏检测

困扰 C 和 C++ 程序员的另外两个问题是内存泄漏和内存损坏。内存泄漏是指分配了内存却从未释放。这会浪费内存，最终导致可能致命的内存不足。内存损坏是指程序无意中将数据写入错误的内存位置，覆盖了该位置的重要数据，同时又未能更新本应写入该数据的内存位置。这两个问题的罪魁祸首都在于语言特性“指针”。

指针是一个强大的工具。如果使用得当，它可以成为一种善的载体——但也可能轻易地变成一种恶的载体。如果指针指向已被释放的内存，或者被意外赋值了一个非零整数或浮点值，它就会成为破坏内存的危险工具，因为通过它写入的数据最终可能会出现在任何地方。同样，当使用指针来跟踪已分配的内存时，很容易忘记在不再需要时释放它。

这会导致内存泄漏。
显然，良好的编码习惯是避免指针相关内存问题的方法之一。而且，编写出几乎不会损坏或泄漏内存的可靠代码也是完全有可能的。尽管如此，拥有一个工具来帮助你检测潜在的内存损坏和泄漏问题肯定不会有什么坏处。值得庆幸的是，有很多这样的工具。

我个人最喜欢的是 IBM 的 Rational Purify，它是 Purify Plus 工具包的一部分。Purify 会在代码运行前对其进行检测，以便追踪所有指针引用以及代码进行的所有内存分配和释放。在 Purify 下运行代码时，您会收到一份实时报告，其中包含代码遇到的问题（包括实际问题和潜在问题）。程序退出时，您会收到一份详细的内存泄漏报告。每个问题都直接链接到导致问题的源代码，从而……

tical and instrumenting methods of profiling. (See https://www.snsystems.com/tech-blog/2014/02/14/function-level-profiling/ for some more details.) Its counterpart, *Razor GPU*, provides profiling and debugging facilities for shaders and compute jobs running on the PS4's GPU.

2.3.1 List of Profilers

There are a great many profiling tools available. See http://en.wikipedia.org/wiki/List_of_performance_analysis_tool for a reasonably comprehensive list.

2.4 Memory Leak and Corruption Detection

Two other problems that plague C and C++ programmers are memory leaks and memory corruption. A memory leak occurs when memory is allocated but never freed. This wastes memory and eventually leads to a potentially fatal out-of-memory condition. Memory corruption occurs when the program inadvertently writes data to the wrong memory location, overwriting the important data that was there—while simultaneously failing to update the memory location where that data *should* have been written. Blame for both of these problems falls squarely on the language feature known as the *pointer*.

A pointer is a powerful tool. It can be an agent of good when used properly—but it can also be all-too-easily transformed into an agent of evil. If a pointer points to memory that has been freed, or if it is accidentally assigned a nonzero integer or floating-point value, it becomes a dangerous tool for corrupting memory, because data written through it can quite literally end up anywhere. Likewise, when pointers are used to keep track of allocated memory, it is all too easy to forget to free the memory when it is no longer needed. This leads to memory leaks.

Clearly, good coding practices are one approach to avoiding pointer-related memory problems. And it is certainly possible to write solid code that essentially never corrupts or leaks memory. Nonetheless, having a tool to help you detect potential memory corruption and leak problems certainly can't hurt. Thankfully, many such tools exist.

My personal favorite is IBM's Rational Purify, which comes as part of the Purify Plus toolkit. Purify instruments your code prior to running it, in order to hook into all pointer dereferences and all memory allocations and deallocations made by your code. When you run your code under Purify, you get a live report of the problems—real and potential—encountered by your code. And when the program exits, you get a detailed memory leak report. Each problem is linked directly to the source code that caused the problem, mak-

追踪和修复这类问题相对容易。您可以在 http://www-306.ibm.com/software/awdtools/purify 上找到更多关于 Purify 的信息。

另外两个流行的工具是 Parasoft 的 Insure++ 和 Julian Seward 及其 Valgrind 开发团队开发的 Valgrind。这些工具都提供内存调试和分析功能。

2.5 其他工具

游戏程序员的工具箱里还有许多其他常用工具。我们不会在这里深入介绍它们，但以下列表可以让你了解它们的存在，并在你想了解更多信息时为你指明正确的方向。

• 差异工具。差异工具或 diff 工具是一种程序，用于比较文本文件的两个版本并确定它们之间发生了哪些变化。（有关 diff 工具的讨论，请参阅 http://en.wikipedia.org/wiki/Diff。）差异通常是逐行计算的，尽管现代 diff 工具也可以显示更改行中已修改的字符范围。大多数版本控制系统都带有 diff 工具。一些程序员喜欢特定的 diff 工具，并配置他们的版本控制软件以使用他们选择的工具。流行的工具包括 ExamDiff（http://www.prestosoft.com/edp_examdiff.asp）、AraxisMerge（http://www.araxis.com）、WinDiff（在大多数 Windows 版本的选项包中提供，也可从许多独立网站获得）和 GNU diff 工具包（http://www.

gnu.org/software/diffutils/diffutils.html）。

• 三向合并工具。当两个人编辑同一个文件时，会生成两组独立的差异。能够将两组差异合并为包含两个人更改的最终文件版本的工具称为三向合并工具。“三向”一词指的是涉及文件的三个版本：原始版本、用户 A 的版本和用户 B 的版本。（请参阅 http://en.wikipedia.org/wiki/3-way_merge）

#Three-way_merge 讨论了双向和三向合并技术。许多合并工具都附带相应的 diff 工具。一些流行的合并工具包括 AraxisMerge (http://www.araxis.com) 和 WinMerge (http://winmerge.org)。Perforce 也附带一个优秀的三向合并工具 (http://www.perforce.com/perforce/products/merge.html)。

ing tracking down and fixing these kinds of problems relatively easy. You can find more information on Purify at http://www-306.ibm.com/software/awdtools/purify.

Two other popular tools are Insure++ by Parasoft, and Valgrind by Julian Seward and the Valgrind development team. These tools provide both memory debugging and profiling facilities.

2.5 Other Tools

There are a number of other commonly used tools in a game programmer's toolkit. We won't cover them in any depth here, but the following list will make you aware of their existence and point you in the right direction if you want to learn more.

- *Difference tools*. A difference tool, or *diff tool*, is a program that compares two versions of a text file and determines what has changed between them. (See http://en.wikipedia.org/wiki/Diff for a discussion of diff tools.) Diffs are usually calculated on a line-by-line basis, although modern diff tools can also show you a range of characters on a changed line that have been modified. Most version control systems come with a diff tool. Some programmers like a particular diff tool and configure their version control software to use the tool of their choice. Popular tools include ExamDiff (http://www.prestosoft.com/edp_examdiff.asp), AraxisMerge (http://www.araxis.com), WinDiff (available in the Options Packs for most Windows versions and available from many independent websites as well), and the GNU diff tools package (http://www.gnu.org/software/diffutils/diffutils.html).
- *Three-way merge tools*. When two people edit the same file, two independent sets of diffs are generated. A tool that can merge two sets of diffs into a final version of the file that contains both person's changes is called a three-way merge tool. The name "three-way" refers to the fact that three versions of the file are involved: the original, user A's version and user B's version. (See http://en.wikipedia.org/wiki/3-way_merge #Three-way_merge for a discussion of two-way and three-way merge technologies.) Many merge tools come with an associated diff tool. Some popular merge tools include AraxisMerge (http://www.araxis.com) and WinMerge (http://winmerge.org). Perforce also comes with an excellent three-way merge tool (http://www.perforce.com/perforce/products/merge.html).

• 十六进制编辑器。十六进制编辑器是一种用于检查和修改二进制文件内容的程序。数据通常以十六进制格式的整数显示，因此得名。大多数优秀的十六进制编辑器可以将数据显示为从 1 字节到 16 字节的整数、32 位和 64 位浮点格式以及 ASCII 文本。十六进制编辑器在查找二进制文件格式问题或对未知二进制格式进行逆向工程时特别有用——这两者在游戏引擎开发领域都是比较常见的工作。市面上有上百万种不同的十六进制编辑器；我使用过 Expert Commercial Software (http://www.expertcomsoft.com/index.html) 的 HexEdit，效果不错，但您的体验可能有所不同。

作为一名游戏引擎程序员，您无疑会遇到其他可以让您的生活更轻松的工具，但我希望本章涵盖了您日常使用的主要工具。

- *Hex editors*. A hex editor is a program used for inspecting and modifying the contents of binary files. The data are usually displayed as integers in hexadecimal format, hence the name. Most good hex editors can display data as integers from one byte to 16 bytes each, in 32- and 64-bit floating-point format and as ASCII text. Hex editors are particularly useful when tracking down problems with binary file formats or when reverse-engineering an unknown binary format—both of which are relatively common endeavors in game engine development circles. There are quite literally a million different hex editors out there; I've had good luck with HexEdit by Expert Commercial Software (http://www.expertcomsoft.com/index.html), but your mileage may vary.

As a game engine programmer you will undoubtedly come across other tools that make your life easier, but I hope this chapter has covered the main tools you'll use on a day-to-day basis.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

3
游戏软件工程基础

在本章中，我们将讨论任何专业游戏程序员所需的基础知识。我们将探索数基及其表示法、典型计算机及其CPU的组件和架构、机器语言和汇编语言，以及C++编程语言。我们将回顾面向对象编程（OOP）中的一些关键概念，然后深入探讨一些在任何软件工程工作（尤其是在游戏开发）中都极具价值的高级主题。与第二章一样，部分读者可能已经熟悉其中一些内容。然而，我强烈建议所有读者至少浏览一下本章，以便我们能够使用相同的工具和资源开始我们的旅程。

3.1 C++ 回顾和最佳实践

由于 C++ 可以说是游戏行业最常用的语言，本书将主要关注 C++。然而，我们将涵盖的大多数概念同样适用于任何面向对象编程语言。当然，游戏行业还使用许多其他语言——像 C 这样的命令式语言；像 C# 这样的面向对象语言，以及

3
Fundamentals of Software Engineering for Games

In this chapter, we'll discuss the foundational knowledge needed by any professional game programmer. We'll explore numeric bases and representations, the components and architecture of a typical computer and its CPU, machine and assembly language, and the C++ programming language. We'll review some key concepts in object-oriented programming (OOP), and then delve into some advanced topics that should prove invaluable in any software engineering endeavor (and especially when creating games). As with Chapter 2, some of this material may already be familiar to some readers. However, I highly recommend that all readers at least skim this chapter, so that we all embark on our journey with the same set of tools and supplies.

3.1 C++ Review and Best Practices

Because C++ is arguably the most commonly used language in the game industry, we will focus primarily on C++ in this book. However, most of the concepts we'll cover apply equally well to *any* object-oriented programming language. Certainly a great many other languages are used in the game industry—imperative languages like C; object-oriented languages like C# and

Java；Python、Lua 和 Perl 等脚本语言；Lisp、Scheme 和 F# 等函数式语言，等等。我强烈建议每位程序员至少学习两门高级语言（越多越好），并至少学习一些汇编语言编程（参见 3.4.7.3 节）。每学习一门新语言都会进一步拓展你的视野，让你能够更深入、更熟练地思考编程。话虽如此，现在让我们将注意力转向面向对象编程的总体概念，尤其是 C++。

3.1.1 面向对象编程简要回顾

本书的大部分内容都假设你对面向对象设计的原理有扎实的理解。如果你对它们有些生疏，接下来的部分应该可以作为一次愉快而快速的复习。如果你不知道本节的内容，我建议你在继续阅读之前先阅读一两本关于面向对象编程（例如 [7]）和 C++ 的书籍（例如 [46] 和 [36]）。

3.1.1.1 类和对象

类是属性（数据）和行为（代码）的集合，它们共同构成一个有用且有意义的整体。类是一种规范，描述了如何构造类的各个实例（称为对象）。例如，你的宠物“Rover”是“狗”类的一个实例。因此，类与其实例之间存在一对多的关系。

3.1.1.2 封装

封装意味着对象仅向外界提供有限的接口；对象的内部状态和实现细节被隐藏。封装简化了类的使用者的工作，因为他们只需了解类的有限接口，而无需了解其实现中可能错综复杂的细节。它还允许编写类的程序员确保其实例始终处于逻辑一致的状态。

3.1.1.3 继承

继承允许将新类定义为现有类的扩展。新类会修改或扩展现有类的数据、接口和/或行为。如果 Child 类扩展了 Parent 类，我们称 Child 继承自 Parent 或从 Parent 派生。在这种关系中， Parent 类被称为基类或超类，而 Child 类则是派生类。

Java; scripting languages like Python, Lua and Perl; functional languages like Lisp, Scheme and F#, and the list goes on. I highly recommend that every programmer learn at least two high-level languages (the more the merrier), as well as learning at least some *assembly language* programming (see Section 3.4.7.3). Every new language that you learn further expands your horizons and allows you to think in a more profound and proficient way about programming overall. That being said, let's turn our attention now to object-oriented programming concepts in general, and C++ in particular.

3.1.1 Brief Review of Object-Oriented Programming

Much of what we'll discuss in this book assumes you have a solid understanding of the principles of object-oriented design. If you're a bit rusty, the following section should serve as a pleasant and quick review. If you have no idea what I'm talking about in this section, I recommend you pick up a book or two on object-oriented programming (e.g., [7]) and C++ in particular (e.g., [46] and [36]) before continuing.

3.1.1.1 Classes and Objects

A *class* is a collection of attributes (data) and behaviors (code) that together form a useful, meaningful whole. A class is a *specification* describing how individual *instances* of the class, known as *objects*, should be constructed. For example, your pet Rover is an instance of the class "dog." Thus, there is a one-to-many relationship between a class and its instances.

3.1.1.2 Encapsulation

Encapsulation means that an object presents only a limited interface to the outside world; the object's internal state and implementation details are kept hidden. Encapsulation simplifies life for the user of the class, because he or she need only understand the class' limited interface, not the potentially intricate details of its implementation. It also allows the programmer who wrote the class to ensure that its instances are always in a logically consistent state.

3.1.1.3 Inheritance

Inheritance allows new classes to be defined as *extensions* to preexisting classes. The new class modifies or extends the data, interface and/or behavior of the existing class. If class `Child` extends class `Parent`, we say that `Child` *inherits from* or is *derived from* `Parent`. In this relationship, the class `Parent` is known as the *base class* or *superclass*, and the class `Child` is the *derived class*

图 3.1. UML 静态类图描绘了一个简单的类层次结构。

或子类。显然，继承会导致类之间出现层次结构（树状结构）关系。

继承在类之间创建了一种“is-a”关系。例如，圆是一种形状。因此，如果我们正在编写一个 2D 绘图应用程序，那么从名为 Shape 的基类派生出 Circle 类可能是合理的。

我们可以使用统一建模语言 (UML) 定义的约定来绘制类层次结构图。在 UML 中，矩形表示类，空心三角形箭头表示继承。继承箭头从子类指向父类。图 3.1 是一个用 UML 静态类图表示的简单类层次结构示例。

多重继承

某些语言支持多重继承 (MI)，这意味着一个类可以有多个父类。理论上，MI 非常优雅，但在实践中，这种设计通常会导致很多混乱和技术难题（参见 http://en.wikipedia.org/wiki/Multiple_inheritance）。这是因为多重继承会将简单的类树转换为潜在的复杂类图。类图可能会出现各种简单类树不会出现的问题，例如致命菱形（http://en.wikipedia.org/wiki/Diamond_problem），其中派生类最终包含祖父基类的两个副本（参见图 3.2）。（在 C++ 中，虚拟继承可以避免祖父数据重复。）多重继承还会使类型转换变得复杂，因为指针的实际地址可能会根据其转换到的基类而变化。这是因为对象中存在多个虚表指针。

大多数 C++ 软件开发者会完全避免多重继承，或者仅允许有限形式的多重继承。一个常见的经验法则是，只允许简单的、无父类被多重继承到原本严格的单继承层次结构中。这类类有时被称为混合类。

Figure 3.1. UML static class diagram depicting a simple class hierarchy.

or *subclass*. Clearly, inheritance leads to hierarchical (tree-structured) relationships between classes.

Inheritance creates an “is-a” relationship between classes. For example, a circle *is a* type of shape. So, if we were writing a 2D drawing application, it would probably make sense to derive our `Circle` class from a base class called `Shape`.

We can draw diagrams of class hierarchies using the conventions defined by the Unified Modeling Language (UML). In this notation, a rectangle represents a class, and an arrow with a hollow triangular head represents inheritance. The inheritance arrow points from child class to parent. See Figure 3.1 for an example of a simple class hierarchy represented as a UML *static class diagram*.

Multiple Inheritance

Some languages support *multiple inheritance* (MI), meaning that a class can have more than one parent class. In theory MI can be quite elegant, but in practice this kind of design usually gives rise to a lot of confusion and technical difficulties (see http://en.wikipedia.org/wiki/Multiple_inheritance). This is because multiple inheritance transforms a simple *tree* of classes into a potentially complex *graph*. A class graph can have all sorts of problems that never plague a simple tree—for example, the *deadly diamond* (http://en.wikipedia.org/wiki/Diamond_problem), in which a derived class ends up containing *two copies* of a grandparent base class (see Figure 3.2). (In C++, *virtual inheritance* allows one to avoid this doubling of the grandparent’s data.) Multiple inheritance also complicates casting, because the actual address of a pointer may change depending on which base class it is cast to. This happens because of the presence of multiple vtable pointers within the object.

Most C++ software developers avoid multiple inheritance completely or only permit it in a limited form. A common rule of thumb is to allow only simple, parentless classes to be multiply inherited into an otherwise strictly single-inheritance hierarchy. Such classes are sometimes called *mix-in classes*

图 3.2。多重继承层次结构中的“致命钻石”。

图 3.3。混合类的示例。

Figure 3.2. “Deadly diamond” in a multiple inheritance hierarchy.

Figure 3.3. Example of a mix-in class.

因为它们可以用来在类树的任意位置引入新功能。参见图 3.3，这是一个略显牵强的混合类示例。

3.1.1.4 多态性

多态性是一种语言特性，它允许通过一个通用接口来操作不同类型的对象集合。从使用接口的代码的角度来看，通用接口使得异构的对象集合看起来是同构的。

例如，一个 2D 绘画程序可能会收到一个包含各种形状的列表，需要在屏幕上绘制。绘制这些异构形状集合的一种方法是使用 switch 语句，针对每种不同类型的形状执行不同的绘制命令。

```
void drawShapes(std::list<Shape*>& shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      switch (pShape->mType)
      {
      case CIRCLE:
         // draw shape as a circle
         break;

      case RECTANGLE:
         // draw shape as a rectangle
         break;

      case TRIANGLE:
         // draw shape as a triangle
         break;

      //...
      }
   }
}
```

这种方法的问题在于 drawShapes() 函数需要“了解”所有可以绘制的形状。在一个简单的例子中，这没什么问题，但随着代码规模和复杂度的增长，向系统添加新形状类型可能会变得困难。每当一个新的形状

because they can be used to introduce new functionality at arbitrary points in a class tree. See Figure 3.3 for a somewhat contrived example of a mix-in class.

3.1.1.4 Polymorphism

Polymorphism is a language feature that allows a collection of objects of different types to be manipulated through a single *common interface*. The common interface makes a heterogeneous collection of objects *appear* to be homogeneous, from the point of view of the code using the interface.

For example, a 2D painting program might be given a list of various shapes to draw on-screen. One way to draw this heterogeneous collection of shapes is to use a switch statement to perform different drawing commands for each distinct type of shape.

```
void drawShapes(std::list<Shape*>& shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      switch (pShape->mType)
      {
      case CIRCLE:
         // draw shape as a circle
         break;

      case RECTANGLE:
         // draw shape as a rectangle
         break;

      case TRIANGLE:
         // draw shape as a triangle
         break;

      //...
      }
   }
}
```

The problem with this approach is that the `drawShapes()` function needs to “know” about all of the kinds of shapes that can be drawn. This is fine in a simple example, but as our code grows in size and complexity, it can become difficult to add new types of shapes to the system. Whenever a new shape

添加类型后，必须在代码库中找到嵌入形状类型集知识的每个位置（例如此 switch 语句），并添加一个案例来处理新类型。

解决方案是将我们大部分代码与可能处理的对象类型隔离开来。为了实现这一点，我们可以为每种想要支持的形状类型定义一个类。所有这些类都将从公共基类 Shape 继承。我们将定义一个名为 Draw() 的虚函数——C++ 语言的主要多态机制，每个不同的形状类都将以不同的方式实现此函数。绘图函数无需“知道”它被赋予了哪些具体的形状类型，现在只需依次调用每个形状的 Draw() 函数即可。

```
struct Shape
{
   virtual void Draw() = 0; // pure virtual function
   virtual ~Shape() { } // ensure derived dtors are virtual
};

struct Circle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a circle
   }
};

struct Rectangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a rectangle
   }
};

struct Triangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a triangle
   }
};
```


type is added, one must find every place in the code base where knowledge of the set of shape types is embedded—like this switch statement—and add a case to handle the new type.

The solution is to insulate the majority of our code from any knowledge of the types of objects with which it might be dealing. To accomplish this, we can define classes for each of the types of shapes we wish to support. All of these classes would inherit from the common base class `Shape`. A *virtual function*—the C++ language's primary polymorphism mechanism—would be defined called `Draw()`, and each distinct shape class would implement this function in a different way. Without "knowing" what specific types of shapes it has been given, the drawing function can now simply call each shape's `Draw()` function in turn.

```
struct Shape
{
   virtual void Draw() = 0; // pure virtual function
   virtual ~Shape() { } // ensure derived dtors are virtual
};

struct Circle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a circle
   }
};

struct Rectangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a rectangle
   }
};

struct Triangle : public Shape
{
   virtual void Draw()
   {
      // draw shape as a triangle
   }
};
```

```
void drawShapes(std::list<Shape*>& shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      pShape->Draw(); // call virtual function
   }
}
```

3.1.1.5 组合和聚合

组合是指使用一组相互作用的对象来完成一项高级任务的做法。组合在类之间创建了一种“包含”或“使用”的关系。（从技术角度来说，“包含”关系称为组合，而“使用”关系称为聚合。）例如，一艘宇宙飞船有一个发动机，而发动机又有一个燃料箱。组合/聚合通常会使各个类更简单、更专注。缺乏经验的面向对象程序员通常过度依赖继承，而往往没有充分利用聚合和组合。

举个例子，假设我们正在为游戏前端设计一个图形用户界面。我们有一个类 Window，它代表任意矩形 GUI 元素。我们还有一个名为 Rectangle 的类，它封装了矩形的数学概念。一个初级程序员可能会从 Rectangle 类派生出 Window 类（使用“is-a”关系）。但在更灵活、封装性更好的设计中，Window 类会引用或包含一个 Rectangle 类（使用“has-a”或“uses-a”关系）。这使得这两个类更简单、更专注，也更容易测试、调试和重用。

3.1.1.6 设计模式

当同一类型的问题反复出现，并且许多不同的程序员都采用了非常相似的解决方案时，我们就说出现了一种设计模式。在面向对象编程中，许多作者已经识别并描述了许多常见的设计模式。关于这个主题最著名的书可能是“四人帮”的著作[19]。

以下是一些常见的通用设计模式的示例。

- 单例模式。此模式确保特定类只有一个实例（单例实例），并提供对它的全局访问点。
- 迭代器。迭代器提供了一种访问集合中各个元素的有效方法，而无需暴露集合的底层


```
void drawShapes(std::list<Shape*>& shapes)
{
   std::list<Shape*>::iterator pShape = shapes.begin();
   std::list<Shape*>::iterator pEnd = shapes.end();

   for ( ; pShape != pEnd; pShape++)
   {
      pShape->Draw(); // call virtual function
   }
}
```

3.1.1.5 Composition and Aggregation

Composition is the practice of using a *group* of *interacting* objects to accomplish a high-level task. Composition creates a “has-a” or “uses-a” relationship between classes. (Technically speaking, the “has-a” relationship is called *composition*, while the “uses-a” relationship is called *aggregation*.) For example, a spaceship *has an* engine, which in turn *has a* fuel tank. Composition/aggregation usually results in the individual classes being simpler and more focused. Inexperienced object-oriented programmers often rely too heavily on inheritance and tend to underutilize aggregation and composition.

As an example, imagine that we are designing a graphical user interface for our game’s front end. We have a class `Window` that represents any rectangular GUI element. We also have a class called `Rectangle` that encapsulates the mathematical concept of a rectangle. A naïve programmer might derive the `Window` class from the `Rectangle` class (using an “is-a” relationship). But in a more flexible and well-encapsulated design, the `Window` class would *refer to* or *contain* a `Rectangle` (employing a “has-a” or “uses-a” relationship). This makes both classes simpler and more focused and allows the classes to be more easily tested, debugged and reused.

3.1.1.6 Design Patterns

When the same type of problem arises over and over, and many different programmers employ a very similar solution to that problem, we say that a *design pattern* has arisen. In object-oriented programming, a number of common design patterns have been identified and described by various authors. The most well-known book on this topic is probably the “Gang of Four” book [19].

Here are a few examples of common general-purpose design patterns.

- *Singleton*. This pattern ensures that a particular class has only one instance (the *singleton instance*) and provides a global point of access to it.
- *Iterator*. An iterator provides an efficient means of accessing the individual elements of a collection, without exposing the collection’s underlying

实现。迭代器“知道”集合的实现细节，因此其用户不必知道。

- 抽象工厂。抽象工厂提供了一个接口，用于创建相关或依赖类的系列，而无需指定它们的具体类。

游戏行业有一套自己的设计模式，用于解决从渲染、碰撞、动画到音频等各个领域的问题。从某种意义上说，这本书讲的是现代 3D 游戏引擎设计中流行的高级设计模式。

看门人和 RAII

作为一个非常有用的设计模式示例，我们来简要了解一下“资源获取即初始化”模式 (RAII)。在此模式中，资源（例如文件、动态分配的内存块或互斥锁）的获取和释放分别绑定到类的构造函数和析构函数。这可以防止程序员意外忘记释放资源——您只需构造一个类的本地实例来获取资源，然后让它超出作用域即可自动释放。在顽皮狗，我们将此类称为“清洁工”，因为它们会为您“清理”工作。

例如，每当我们需要从特定类型的分配器分配内存时，我们都会将该分配器推送到全局分配器栈中，并且分配完成后，我们必须始终记得将该分配器弹出栈。为了更方便、更不容易出错，我们使用了分配管理器。这个小类的构造函数会推送分配器，析构函数则会弹出分配器：

```
class AllocJanitor
{
public:
    explicit AllocJanitor(mem::Context context)
    {
        mem::PushAllocator(context);
    }
    ~AllocJanitor()
    {
        mem::PopAllocator();
    }
};
```

要使用 janitor 类，我们只需构造它的本地实例。当此实例超出范围时，分配器将自动弹出：

implementation. The iterator “knows” the implementation details of the collection so that its users don’t have to.

- *Abstract factory*. An abstract factory provides an interface for creating families of related or dependent classes without specifying their concrete classes.

The game industry has its own set of design patterns for addressing problems in every realm from rendering to collision to animation to audio. In a sense, this book is all about the high-level design patterns prevalent in modern 3D game engine design.

Janitors and RAII

As one very useful example of a design pattern, let’s have a brief look at the “resource acquisition is initialization” pattern (RAII). In this pattern, the acquisition and release of a resource (such as a file, a block of dynamically allocated memory, or a mutex lock) are bound to the constructor and destructor of a class, respectively. This prevents programmers from accidentally forgetting to release the resource—you simply construct a local instance of the class to acquire the resource, and let it fall out of scope to release it automatically. At Naughty Dog, we call such classes *janitors* because they “clean up” after you.

For example, whenever we need to allocate memory from a particular type of allocator, we *push* that allocator onto a global *allocator stack*, and when we’re done allocating we must always remember to *pop* the allocator off the stack. To make this more convenient and less error-prone, we use an *allocation janitor*. This tiny class’s constructor pushes the allocator, and its destructor pops the allocator:

```
class AllocJanitor
{
public:
    explicit AllocJanitor(mem::Context context)
    {
        mem::PushAllocator(context);
    }
    ~AllocJanitor()
    {
        mem::PopAllocator();
    }
};
```

To use the janitor class, we simply construct a local instance of it. When this instance drops out of scope, the allocator will be popped automatically:

```
void f()
{
    // do some work...

    // allocate temp buffers from single-frame allocator
    {
        AllocJanitor janitor(mem::Context::kSingleFrame);

        U8* pByteBuffer = new U8[SIZE];
        float* pFloatBuffer = new float[SIZE];

        // use buffers...

        // (NOTE: no need to free the memory because we
        // used a single-frame allocator)
    } // janitor pops allocator when it drops out of scope

    // do more work...
}
```

有关非常有用的 RAII 模式的更多信息，请参阅 http://en.cppreference.com/w/cpp/language/raii。

3.1.2 C++语言标准化

自 1979 年诞生以来，C++ 语言一直在不断发展。其发明者 Bjarne Stroustrup 最初将该语言命名为“带类的 C”，但在 1983 年更名为“C++”。国际标准化组织 (ISO，www.iso.org) 于 1998 年首次对该语言进行了标准化，该版本即今天的 C++98。此后，ISO 定期发布 C++ 语言的更新标准，旨在使该语言更强大、更易于使用、更少歧义。这些目标的实现方式包括：改进语言的语义和规则，添加新的、更强大的语言特性，以及弃用或完全删除语言中那些已被证明存在问题或不受欢迎的部分。

C++ 编程语言标准的最新版本称为 C++17，于 2017 年 7 月 31 日发布。该标准的下一个版本 C++2a 在本文发布时正在开发中。以下按时间顺序总结了 C++ 标准的各个版本。

- C++98 是第一个官方 C++ 标准，由 ISO 于 1998 年制定。
- C++03 于 2003 年推出，旨在解决 C++98 标准中发现的各种问题。


```
void f()
{
    // do some work...

    // allocate temp buffers from single-frame allocator
    {
        AllocJanitor janitor(mem::Context::kSingleFrame);

        U8* pByteBuffer = new U8[SIZE];
        float* pFloatBuffer = new float[SIZE];

        // use buffers...

        // (NOTE: no need to free the memory because we
        // used a single-frame allocator)
    } // janitor pops allocator when it drops out of scope

    // do more work...
}
```

See http://en.cppreference.com/w/cpp/language/raii for more information on the highly useful RAII pattern.

3.1.2 C++ Language Standardization

Since its inception in 1979, the C++ language has been continually evolving. Bjarne Stroustrup, its inventor, originally named the language “C with Classes”, but it was renamed “C++” in 1983. The International Organization for Standardization (ISO, www.iso.org) first standardized the language in 1998—this version is known today as C++98. Since then, the ISO has been periodically publishing updated standards for the C++ language, with the goals of making the language more powerful, easier to use, and less ambiguous. These goals are achieved by refining the semantics and rules of the language, by adding new, more-powerful language features, and by deprecating, or removing entirely, those aspects of the language which have proven problematic or unpopular.

The most-recent variant of the C++ programming language standard is called C++17, which was published on July 31, 2017. The next iteration of the standard, C++2a, was in development at the time of this publication. The various versions of the C++ standard are summarized in chronological order below.

- *C++98* was the first official C++ standard, established by the ISO in 1998.
- *C++03* was introduced in 2003, to address various problems that had been identified in the C++98 standard.

• C++11（在其大部分开发过程中也称为 C++0x）于 2011 年 8 月 12 日获得 ISO 批准。C++11 为该语言添加了大量强大的新功能，包括：

- 类型安全的 nullptr 文字，用于替换从 C 语言继承的容易出错的 NULL 宏；
- 用于类型推断的 auto 和 decltype 关键字；
- “尾随返回类型”语法，允许使用函数输入参数的 decltype 来描述该函数的返回类型；
- override 和 final 关键字用于在定义和覆盖虚函数时提高表达能力；
- 默认和删除函数（允许程序员明确请求使用编译器生成的默认实现，或者函数的实现应该未定义）；
- 委托构造函数——一个构造函数可以调用同一个类中的另一个构造函数；
- 强类型枚举；
- constexpr 关键字，通过在编译时评估表达式来定义编译时常量值；
- 统一的初始化语法，扩展原始基于大括号的 POD 初始化器以涵盖非 POD 类型；
- 支持 lambda 函数和变量捕获（闭包）；
- 引入右值引用和移动语义，以便更有效地处理临时对象；
- 标准化属性说明符，以替换编译器特定的说明符，例如 __attribute__((...)) 和 __declspec() 。

C++11 还引入了改进和扩展的标准库，包括对线程（并发编程）的支持、改进的智能指针设施和扩展的通用算法集。

• C++14 于 2014 年 8 月 18 日获得 ISO 批准，并于 2014 年 12 月 15 日发布。它对 C++11 的增加和改进包括：

- 返回类型推导，在许多情况下允许使用简单的 auto 关键字声明函数返回类型，而无需 C++11 所要求的冗长的尾随 decltype 表达式；

- *C++11* (also known as *C++0x* during much of its development) was approved by the ISO on August 12, 2011. C++11 added a large number of powerful new features to the language, including:
  - a type-safe `nullptr` literal, to replace the bug-prone `NULL` macro that had been inherited from the C language;
  - the `auto` and `decltype` keywords for type inference;
  - a “trailing return type” syntax, which allows a `decltype` of a function’s input arguments to be used to describe the return type of that function;
  - the `override` and `final` keywords for improved expressiveness when defining and overriding virtual functions;
  - defaulted and deleted functions (allowing the programmer to explicitly request that a compiler-generated default implementation be used, or that a function’s implementation should be undefined);
  - delegating constructors—the ability of one constructor to invoke another within the same class;
  - strongly-typed enumerations;
  - the `constexpr` keyword for defining compile-time constant values by evaluating expressions at compile time;
  - a uniform initialization syntax that extends the original braces-based POD initializers to cover non-POD types as well;
  - support for lambda functions and variable capture (closures);
  - the introduction of rvalue references and move semantics for more efficient handling of temporary objects; and
  - standardized attribute specifiers, to replace compiler-specific specifiers such as `__attribute__((...))` and `__declspec()`.

  C++11 also introduced an improved and expanded standard library, including support for threading (concurrent programming), improved smart pointer facilities and an expanded set of generic algorithms.
- *C++14* was approved by the ISO on August 18, 2014 and released on December 15, 2014. Its additions and improvements to C++11 include:
  - return type deduction, which in many situations allows function return types to be declared using a simple `auto` keyword, without the need for the verbose trailing `decltype` expression required by C++11;

- 通用 lambda，允许 lambda 通过使用 auto 声明其输入参数，像模板函数一样运行；
- 在 lambda 中初始化“捕获”变量的能力；
- 以 0b 开头的二进制文字（例如， 0b10110110 ）；
- 支持数字文字中的数字分隔符，以提高可读性（例如，用 1'000'000 代替 1000000 ）；
- 变量模板，允许在声明变量时使用模板语法；
- 放宽对 constexpr 的一些限制，包括在常量表达式中使用 if 、 switch 和循环的能力。

• C++17 于 2017 年 7 月 31 日由 ISO 发布。它在许多方面扩展和改进了 C++14，包括但不限于以下内容：

- 删除了一些过时和/或危险的语言特性，包括三字母组合、register 关键字和已经弃用的 auto_ptr 智能指针类；
- 保证复制省略，即省略不必要的对象复制；
- 异常规范现在是类型系统的一部分，这意味着 void f() noexcept(true); 和 void f() noexcept(false); 现在是不同的类型；
- 向语言中添加两个新的文字：UTF-8 字符文字（例如， u8'x' ）和具有十六进制基数和十进制指数的浮点文字（例如， 0xC.68p+2 ）；
- 引入结构化绑定到 C++，允许将集合数据类型中的值“解包”到单个变量中（例如，auto [a, b] = func_that_returns_a_pair();）——这种语法与通过 Python 中的元组从函数返回多个值的语法非常相似；
- 添加了一些有用的标准化属性，包括 [[fallthrough]]，它允许您明确记录 switch 中缺少 break 语句的事实是故意的，从而抑制原本会生成的警告。

3.1.2.1 进一步阅读

有大量优秀的在线资源和书籍详细描述了 C++11、C++14 和 C++17 的特性，因此我们不会在这里尝试介绍它们。
以下是一些有用的参考：

- generic lambdas, allowing a lambda to act like a templated function by using `auto` to declare its input arguments;
- the ability to initialize “captured” variables in lambdas;
- binary literals prefaced with `0b` (e.g., `0b10110110`);
- support for digit separators in numeric literals, for improved readability (e.g., `1'000'000` instead of `1000000`);
- variable templates, which allow template syntax to be used when declaring variables; and
- relaxation of some restrictions on `constexpr`, including the ability to use `if`, `switch` and loops within constant expressions.

• *C++17* was published by the ISO on July 31, 2017. It extends and improves C++14 in many ways, including but not limited to the following:

- removed a number of out-dated and/or dangerous language features, including trigraphs, the `register` keyword, and the already-deprecated `auto_ptr` smart pointer class;
- guarantees copy *elision*, the omission of unnecessary object copying;
- exception specifications are now part of the type system, meaning that `void f() noexcept(true);` and `void f() noexcept(false);` are now distinct types;
- adds two new literals to the language: UTF-8 character literals (e.g., `u8'x'`), and floating-point literals with a hexadecimal base and decimal exponent (e.g., `0xC.68p+2`);
- introduces *structured bindings* to C++, allowing the values in a collection data type to be “unpacked” into individual variables (e.g., `auto [a, b] = func_that_returns_a_pair();`)—a syntax that is strikingly similar to that of returning multiple values from a function via a tuple in Python;
- adds some useful standardized attributes, including `[[fallthrough]]` which allows you to explicitly document the fact that a missing `break` statement in a `switch` is intentional, thereby suppressing the warning that would otherwise be generated.

3.1.2.1 Further Reading

There are plenty of great online resources and books that describe the features of C++11, C++14 and C++17 in detail, so we won’t attempt to cover them here. Here are a few useful references:

• 网站 http://en.cppreference.com/w/cpp 提供了出色的 C++ 参考手册，其中包括“自 C++11 以来”或“直到 C++17”等标注，分别指示何时将某些语言功能添加到标准中或从标准中删除。

• 有关 ISO 标准化工作的信息可在 https://isocpp.org/std 上找到。

• 请参阅以下网站以获取有关 C++11 主要新功能的详细摘要：

  ◦ https://www.codeproject.com/Articles/570638/Ten-CplusplusFeatures-Every-Cplusplus-Developer，以及

  ◦ https://blog.smartbear.com/development/the-biggest-changesin-c11-and-why-you-should-care。

• 请参阅http://thbecker.net/articles/auto_and_decltype/section_01.html，了解有关 auto 和 decltype 的详细说明。

• 请参阅http://www.drdobbs.com/cpp/the-c14-standard-what-you-needto-know/240169034，以详细了解 C++14 对 C++11 标准的更改。

- **有关 C++17 标准与 C++14 的完整区别列表，请参阅 https://isocpp.org/files/papers/p0636r0.html。**

3.1.2.2 使用哪些语言特性?

当你读到 C++ 中新增的众多炫酷功能时，你可能会想在你的引擎或游戏中使用所有这些功能。然而，仅仅因为某个功能存在并不意味着你的团队需要立即开始使用它。

在顽皮狗，我们倾向于采取保守的方式将新的语言特性引入代码库。作为我们工作室编码标准的一部分，我们列出了允许在运行时代码中使用的 C++ 语言特性，以及另一个略微宽松的语言特性列表，这些特性允许在我们的离线工具代码中使用。我们采取这种谨慎态度的原因有很多，我将在以下章节中概述。

缺乏全功能支持

你的编译器可能不完全支持“前沿”特性。例如，LLVM/Clang，索尼 PlayStation 上使用的 C++ 编译器

- The site http://en.cppreference.com/w/cpp provides an excellent C++ reference manual, including call-outs such as “since C++11” or “until C++17“ to indicate when certain language features were added to or removed from the standard, respectively.

- Information about the ISO’s standardization efforts can be found at https://isocpp.org/std.

- See the following sites for good summaries of C++11’s major new features:

  ◦ https://www.codeproject.com/Articles/570638/Ten-Cplusplus-Features-Every-Cplusplus-Developer, and

  ◦ https://blog.smartbear.com/development/the-biggest-changes-in-c11-and-why-you-should-care.

- See http://thbecker.net/articles/auto_and_decltype/section_01.html for a great treatment of `auto` and `decltype`.

- See http://www.drdobbs.com/cpp/the-c14-standard-what-you-need-to-know/240169034 for a good treatment of C++14’s changes to the C++11 standard.

- See https://isocpp.org/files/papers/p0636r0.html for a complete list of how the C++17 standard differs from C++14.

3.1.2.2 Which Language Features to Use?

As you read about all the cool new features being added to C++, it’s tempting to think that you need to use *all* of these features in your engine or game. However, just because a feature exists doesn’t mean your team needs to immediately start making use of it.

At Naughty Dog, we tend to take a conservative approach to adopting new language features into our codebase. As part of our studio’s coding standards, we have a list of those C++ language features that are approved for use in our runtime code, and another somewhat more liberal list of language features that are allowed in our offline tools code. There are a number of reasons for this cautious approach, which I’ll outline in the following sections.

Lack of Full Feature Support

The “bleeding edge” features may not be fully supported by your compiler. For example, LLVM/Clang, the C++ compiler used on the Sony Playstation

4，目前在 3.3 及更高版本中支持完整的 C++11 标准，在 3.4 及更高版本中支持完整的 C++14 标准。但它对 C++17 的支持涵盖了 Clang 3.5 至 4.0 版本，并且目前不支持 C++2a 标准草案。此外，Clang 默认以 C++98 模式编译代码——对一些更高级标准的支持可以作为扩展接受，但为了启用完整支持，必须向编译器传递特定的命令行参数。详情请参阅 https://clang.llvm.org/cxx_status.html。

标准切换成本

将代码库从一种标准切换到另一种标准并非零成本。因此，游戏工作室务必选择支持最先进的 C++ 标准，并在一段合理的时间内（例如，一个项目的整个开发周期）坚持使用。顽皮狗工作室最近才开始采用 C++11 标准，并且只允许在开发《最后生还者 第二部》的代码分支中使用该标准。《神秘海域 4：盗贼末路》和《神秘海域：失落的遗产》的代码分支最初是使用 C++98 标准编写的，我们认为，在该代码库中采用 C++11 特性所带来的相对较小的收益无法抵消这样做的成本和风险。

风险与回报

并非所有 C++ 语言特性都生来平等。有些特性很有用，而且几乎被普遍接受，比如 nullptr 。其他特性可能有好处，但也有坏处。还有一些语言特性可能被认为根本不适合在运行时引擎代码中使用。

举一个既有优点又有缺点的语言特性的例子，我们来看看 C++11 对 auto 关键字的新解释。这个关键字确实使变量和函数的编写更加方便。但顽皮狗的程序员意识到过度使用 auto 会导致代码混乱：举一个极端的例子，想象一下尝试读取别人编写的 .cpp 文件，其中几乎所有变量、函数参数和返回值都被声明为 auto 。这就像读取 Python 或 Lisp 等无类型语言一样。像 C++ 这样的强类型语言的好处之一是程序员能够快速轻松地确定所有变量的类型。因此，我们决定采用一个简单的规则：auto 只能在声明迭代器时使用，在其他方法都行不通的情况下（例如在模板定义中），或者在使用 auto 可以显著提高代码清晰度、可读性和可维护性的特殊情况下使用。在所有其他情况下，我们都要求使用显式类型声明。

4, currently supports the entire C++11 standard in versions 3.3 and later, and all of C++14 in versions 3.4 and later. But its support for C++17 is spread across Clang versions 3.5 through 4.0, and it currently has no support for the draft C++2a standard. Also, Clang compiles code in C++98 mode by default—support for some of the more-advanced standards are accepted as extensions, but in order to enable full support one must pass specific command-line arguments to the compiler. See https://clang.llvm.org/cxx_status.html for details.

Cost of Switching between Standards

There's a non-zero cost to switching your codebase from one standard to another. Because of this, it's important for a game studio to decide on the most-advanced C++ standard to support, and then stick with it for a reasonable length of time (e.g., for the duration of one project). At Naughty Dog, we adopted the C++11 standard only relatively recently, and we only allowed its use in the code branch in which *The Last of Us Part II* is being developed. The code in the branch used for *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy* had been written originally using the C++98 standard, and we decided that the relatively minor benefits of adopting C++11 features in that codebase did not outweigh the cost and risks of doing so.

Risk versus Reward

Not every C++ language feature is created equal. Some features are useful and pretty much universally acceptable, like `nullptr`. Others may have benefits, but also associated negatives. Still other language features may be deemed inappropriate for use in runtime engine code altogether.

As an example of a language feature with both benefits and downsides, consider the new C++11 interpretation of the `auto` keyword. This keyword certainly makes variables and functions more convenient to write. But the Naughty Dog programmers recognized that over-use of `auto` can lead to obfuscated code: As an extreme example, imagine trying to read a .cpp file written by somebody else, in which virtually every variable, function argument and return value is declared `auto`. It would be like reading a typeless language such as Python or Lisp. One of the benefits of a strongly-typed language like C++ is the programmer's ability to quickly and easily determine the types of all variables. As such, we decided to adopt a simple rule: `auto` may only be used when declaring iterators, in situations where no other approach works (such as within template definitions), or in special cases when code clarity, readability and maintainability is significantly improved by its use. In all other cases, we require the use of explicit type declarations.

举个例子，模板元编程就是一种可能被认为不适合在游戏等商业产品中使用的语言特性。Andrei Alexandrescu 的 Loki 库 [3] 大量使用模板元编程来实现一些非常有趣和神奇的功能。然而，生成的代码难以阅读，有时不可移植，并且给程序员的理解带来了极高的障碍。顽皮狗的编程主管认为，任何程序员都应该能够在短时间内介入并调试问题，即使是在他们可能不太熟悉的代码中。因此，顽皮狗禁止在运行时引擎代码中使用复杂的模板元编程，只有在收益大于成本的情况下才会在个案基础上例外。

总而言之，请记住，当你手里拿着锤子时，所有东西看起来都像钉子。不要仅仅因为语言中存在某些特性（或者因为它们是新的）就贸然使用它们。明智而深思熟虑的方法将带来稳定的代码库，使其尽可能易于理解、推理、调试和维护。

3.1.3 编码标准：为什么以及多少？

工程师之间关于编码规范的讨论常常会引发激烈的“宗教式”争论。我并不想在这里引发任何这样的争论，但我还是建议，至少遵循一套最低限度的编码规范是个好主意。编码规范的存在主要有两个原因。

1.一些标准使得代码更具可读性、可理解性和可维护性。

2. 其他约定有助于防止程序员搬起石头砸自己的脚。例如，编码标准可能会鼓励程序员只使用整个语言中较小、更易于测试且更不容易出错的子集。C++ 语言很容易被滥用，因此这种编码标准在使用 C++ 时尤为重要。

在我看来，编码约定中最重要的事情如下。

- 接口为王。保持你的接口（.h 文件）干净、简洁、精简、易于理解且注释良好。
- 好的名字有助于理解，避免混淆。坚持使用直观的名称，这些名称应直接映射到相关类、函数或变量的用途。提前花时间确定一个好的名字。避免

As an example of a language feature that could be deemed inappropriate for use in a commercial product like a game, consider *template metaprogramming*. Andrei Alexandrescu's Loki library [3] makes heavy use of template metaprogramming to do some pretty interesting and amazing things. However, the resulting code is tough to read, is sometimes non-portable, and presents programmers with an extremely high barrier to understanding. The programming leads at Naughty Dog believe that any programmer should be able to jump in and debug a problem on short notice, even in code with which he or she may not be very familiar. As such, Naughty Dog prohibits complex template metaprogramming in runtime engine code, with exceptions made only on a case-by-case basis where the benefits are deemed to outweigh the costs.

In summary, remember that when you have a hammer, everything can tend to look like a nail. Don't be tempted to use features of your language just because they're there (or because they're new). A judicious and carefully considered approach will result in a stable codebase that's as easy as possible to understand, reason about, debug and maintain.

3.1.3 Coding Standards: Why and How Much?

Discussions of coding conventions among engineers can often lead to heated "religious" debates. I do not wish to spark any such debate here, but I will go so far as to suggest that following at least a minimal set of coding standards is a good idea. Coding standards exist for two primary reasons.

1. Some standards make the code more readable, understandable and maintainable.
2. Other conventions help to prevent programmers from shooting themselves in the foot. For example, a coding standard might encourage the programmer to use only a smaller, more testable and less error-prone subset of the whole language. The C++ language is rife with possibilities for abuse, so this kind of coding standard is particularly important when using C++.

In my opinion, the most important things to achieve in your coding conventions are the following.

- *Interfaces are king*. Keep your interfaces (.h files) clean, simple, minimal, easy to understand and well-commented.
- *Good names encourage understanding and avoid confusion*. Stick to intuitive names that map directly to the purpose of the class, function or variable in question. Spend time up-front identifying a good name. Avoid

一种命名方案，要求程序员使用查找表来解读代码的含义。请记住，像 C++ 这样的高级编程语言是供人类阅读的。（如果你不同意，那就问问自己，为什么不直接用机器语言编写所有软件。）

• 不要使全局命名空间混乱。使用 C++ 命名空间或通用命名前缀，以确保您的符号不会与其他库中的符号冲突。（但请注意不要过度使用命名空间，也不要将其嵌套得太深。）命名 #define 符号时要格外小心；请记住，C++ 预处理器宏实际上只是文本替换，因此它们会跨越所有 C/C++ 作用域和命名空间的边界。

• 遵循 C++ 最佳实践。Scott Meyers 的《Effective C++》系列 [36,37]、Meyers 的《Effective STL》[38] 以及 John Lakos 的《Large-Scale C++ Software Design》[31] 等书籍提供了出色的指导，可帮助你避免麻烦。

• 保持一致。我尝试遵循的规则如下：如果您从头开始编写代码，请随意发明任何您喜欢的约定，然后坚持下去。编辑现有代码时，请尝试遵循已建立的任何约定。

• 让错误显而易见。Joel Spolsky 写了一篇关于编码规范的优秀文章，可以在 http://www.joelonsoftware.com/articles/Wrong.html 找到。Joel 认为，“最干净”的代码不一定是表面上看起来整洁的代码，而是以一种让常见编程错误更容易被发现的方式编写的代码。Joel 的文章总是充满乐趣和教育意义，我强烈推荐这篇。

3.2 捕获和处理错误

游戏引擎中有很多方法可以捕获和处理错误情况。作为一名游戏程序员，了解这些不同的机制、它们的优缺点以及何时使用它们至关重要。

3.2.1 错误类型

任何软件项目中都存在两种基本的错误情况：用户错误和程序员错误。用户错误是指程序用户执行了不正确的操作，例如输入无效内容、尝试打开不存在的文件等。程序员错误则是由代码本身的错误导致的。虽然它可能是由用户的操作触发的，但

a naming scheme that requires programmers to use a look-up table in order to decipher the meaning of your code. Remember that high-level programming languages like C++ are intended for *humans* to read. (If you disagree, just ask yourself why you don't write all your software directly in machine language.)

- *Don't clutter the global namespace.* Use C++ namespaces or a common naming prefix to ensure that your symbols don't collide with symbols in other libraries. (But be careful not to overuse namespaces, or nest them too deeply.) Name `#defined` symbols with extra care; remember that C++ preprocessor macros are really just text substitutions, so they cut across all C/C++ scope and namespace boundaries.
- *Follow C++ best practices.* Books like the *Effective C++* series by Scott Meyers [36,37], Meyers' *Effective STL* [38] and *Large-Scale C++ Software Design* by John Lakos [31] provide excellent guidelines that will help keep you out of trouble.
- *Be consistent.* The rule I try to use is as follows: If you're writing a body of code from scratch, feel free to invent any convention you like—then stick to it. When editing preexisting code, try to follow whatever conventions have already been established.
- *Make errors stick out.* Joel Spolsky wrote an excellent article on coding conventions, which can be found at http://www.joelonsoftware.com/articles/Wrong.html. Joel suggests that the "cleanest" code is not necessarily code that looks neat and tidy on a superficial level, but rather the code that is written in a way that makes common programming errors *easier to see*. Joel's articles are always fun and educational, and I highly recommend this one.

3.2 Catching and Handling Errors

There are a number of ways to catch and handle error conditions in a game engine. As a game programmer, it's important to understand these different mechanisms, their pros and cons and when to use each one.

3.2.1 Types of Errors

In any software project there are two basic kinds of error conditions: *user errors* and *programmer errors*. A user error occurs when the user of the program does something incorrect, such as typing an invalid input, attempting to open a file that does not exist, etc. A programmer error is the result of a *bug* in the code itself. Although it may be triggered by something the user has done, the

程序员错误的本质是，如果程序员没有犯错，问题本来是可以避免的，并且用户有合理的期望，认为程序应该能够妥善处理这种情况。

当然，“用户”的定义会根据具体情况而变化。在游戏项目中，用户错误大致可以分为两类：游戏玩家造成的错误和游戏开发过程中开发者造成的错误。跟踪特定错误影响的用户类型并进行适当的处□□理至关重要。

实际上还有第三种用户——团队中的其他程序员。（如果您正在编写游戏中间件软件，例如 Havok 或 OpenGL，则第三类扩展到世界各地使用您的库的其他程序员。）这是用户错误和程序员错误之间的界限变得模糊的地方。假设程序员 A 编写了一个函数 f()，程序员 B 试图调用它。如果 B 使用无效参数（例如空指针或超出范围的数组索引）调用 f()，那么程序员 A 可能会将此视为用户错误，但从 B 的角度来看，这将是程序员错误。（当然，人们也可以争辩说，程序员 A 应该预料到传递无效参数，并且应该妥善处理它们，因此问题实际上是程序员 A 的错误。）这里要记住的关键是，用户和程序员之间的界限会根据上下文而变化 - 它很少是黑白分明的区别。

3.2.2 处理错误

在处理错误时，两种类型的要求差异很大。最好尽可能优雅地处理用户错误，向用户显示一些有用的信息，然后允许他或她继续工作——或者在游戏中，继续玩。另一方面，程序员错误不应该采用优雅的“告知并继续”策略来处理。相反，通常最好的方法是暂停程序并提供详细的底层调试信息，以便程序员能够快速识别和修复问题。理想情况下，所有程序员错误都应该在软件发布之前被发现并修复。

3.2.2.1 处理播放器错误

当“用户”指的是正在玩游戏的人时，错误处理显然应该在游戏玩法的背景下进行。例如，如果玩家在没有弹药的情况下尝试重新装填武器，音频提示和/或动画可以提示玩家这个问题，而无需让玩家“退出游戏”。

essence of a programmer error is that the problem could have been avoided if the programmer had not made a mistake, and the user has a reasonable expectation that the program *should* have handled the situation gracefully.

Of course, the definition of “user” changes depending on context. In the context of a game project, user errors can be roughly divided into two categories: errors caused by the person playing the game and errors caused by the people who are making the game during development. It is important to keep track of which type of user is affected by a particular error and handle the error appropriately.

There’s actually a third kind of user—the other programmers on your team. (And if you are writing a piece of game middleware software, like Havok or OpenGL, this third category extends to other programmers all over the world who are using your library.) This is where the line between *user errors* and *programmer errors* gets blurry. Let’s imagine that programmer A writes a function `f()`, and programmer B tries to call it. If B calls `f()` with invalid arguments (e.g., a null pointer, or an out-of-range array index), then this could be seen as a user error by programmer A, but it would be a programmer error from B’s point of view. (Of course, one can also argue that programmer A should have anticipated the passing of invalid arguments and should have handled them gracefully, so the problem really is a programmer error, on A’s part.) The key thing to remember here is that the line between user and programmer can shift depending on context—it is rarely a black-and-white distinction.

3.2.2 Handling Errors

When handling errors, the requirements differ significantly between the two types. It is best to handle *user errors* as gracefully as possible, displaying some helpful information to the user and then allowing him or her to continue working—or in the case of a game, to continue playing. Programmer errors, on the other hand, should *not* be handled with a graceful “inform and continue” policy. Instead, it is usually best to halt the program and provide detailed low-level debugging information, so that a programmer can quickly identify and fix the problem. In an ideal world, *all* programmer errors would be caught and fixed before the software ships to the public.

3.2.2.1 Handling Player Errors

When the “user” is the person playing your game, errors should obviously be handled within the context of gameplay. For example, if the player attempts to reload a weapon when no ammo is available, an audio cue and/or an animation can indicate this problem to the player without taking him or her “out of the game.”

3.2.2.2 处理开发人员错误

当“用户”是游戏制作者，例如美术师、动画师或游戏设计师时，错误可能是由某种无效资源引起的。例如，动画可能与错误的骨架关联，纹理大小可能错误，或者音频文件的采样率可能不受支持。对于这类开发者错误，存在两种相互竞争的观点。

一方面，防止不良游戏资源长期存在似乎至关重要。一款游戏通常包含数千个资源，而一个问题资源可能会“丢失”，在这种情况下，不良资源就有可能一直保留到最终发行的游戏。如果将这种观点推向极端，那么处理不良游戏资源的最佳方法就是，即使遇到一个问题资源，也要阻止整个游戏运行。这无疑会强烈激励创建无效资源的开发者立即删除或修复它。

另一方面，游戏开发是一个混乱且反复的过程，一次性生成“完美”的资源确实很少见。按照这种思路，游戏引擎应该对几乎所有可以想象到的问题都具有鲁棒性，这样即使面对完全无效的游戏资源数据，工作也能继续进行。但这也并非理想状态，因为游戏引擎会变得臃肿不堪，充斥着错误捕获和处理代码，而一旦开发节奏稳定下来，游戏发布，这些代码就不再需要了。而且，发布带有“劣质”资源的产品的可能性也变得非常高。

根据我的经验，最好的方法是在这两个极端之间找到一个折中点。当开发人员发生错误时，我喜欢让错误显而易见，然后允许团队在问题存在的情况下继续工作。仅仅因为一名开发人员试图向游戏中添加无效资源就阻止团队中所有其他开发人员工作，代价极其高昂。游戏工作室的薪酬很高，当多名团队成员遭遇宕机时，成本会随着无法工作的人数成倍增加。当然，我们只应在切实可行的情况下以这种方式处理错误，而不会花费过多的工程时间或使代码臃肿。

举个例子，假设某个网格无法加载。在我看来，最好在游戏世界中该网格应该出现的位置画一个大红框，并在每个位置上方悬停一个文本字符串，内容是“网格加载失败”。这比在错误日志中打印一条容易被忽略的消息要好得多。而且这比直接让游戏崩溃要好得多，因为那样的话，在那个网格加载完成之前，没人能继续工作。

3.2.2.2 Handling Developer Errors

When the “user” is someone who is making the game, such as an artist, animator or game designer, errors may be caused by an invalid asset of some sort. For example, an animation might be associated with the wrong skeleton, or a texture might be the wrong size, or an audio file might have been sampled at an unsupported sample rate. For these kinds of *developer errors*, there are two competing camps of thought.

On the one hand, it seems important to prevent bad game assets from persisting for too long. A game typically contains literally thousands of assets, and a problem asset might get “lost,” in which case one risks the possibility of the bad asset surviving all the way into the final shipping game. If one takes this point of view to an extreme, then the best way to handle bad game assets is to prevent the entire game from running whenever even a single problematic asset is encountered. This is certainly a strong incentive for the developer who created the invalid asset to remove or fix it immediately.

On the other hand, game development is a messy and iterative process, and generating “perfect” assets the first time around is rare indeed. By this line of thought, a game engine should be robust to almost any kind of problem imaginable, so that work can continue even in the face of totally invalid game asset data. But this too is not ideal, because the game engine would become bloated with error-catching and error-handling code that won’t be needed once the development pace settles down and the game ships. And the probability of shipping the product with “bad” assets becomes too high.

In my experience, the best approach is to find a middle ground between these two extremes. When a developer error occurs, I like to *make the error obvious* and then allow the team to continue to work in the presence of the problem. It is extremely costly to prevent all the other developers on the team from working, just because one developer tried to add an invalid asset to the game. A game studio pays its employees well, and when multiple team members experience downtime, the costs are multiplied by the number of people who are prevented from working. Of course, we should only handle errors in this way when it is practical to do so, without spending inordinate amounts of engineering time, or bloating the code.

As an example, let’s suppose that a particular mesh cannot be loaded. In my view, it’s best to draw a big red box in the game world at the places that mesh would have been located, perhaps with a text string hovering over each one that reads, “Mesh *blah-dee-blah* failed to load.” This is superior to printing an easy-to-miss message to an error log. And it’s *far* better than just crashing the game, because then no one will be able to work until that one mesh

参考已修复。当然，对于特别严重的问题，只需发出错误消息并崩溃即可。没有万能的灵丹妙药可以解决所有类型的问题，并且随着经验的积累，您对特定情况应采用哪种错误处理方法的判断会更加准确。

3.2.2.3 处理程序员错误

检测和处理程序员错误（又称 bug）的最佳方法通常是在源代码中嵌入错误检查代码，并在错误检查失败时暂停程序。这种机制称为断言系统；我们将在 3.2.3.3 节中详细探讨断言。当然，正如我们上面所说，一个程序员的用户错误可能是另一个程序员的 bug；因此，断言并非总是处理所有程序员错误的正确方法。在断言和更优雅的错误处理技术之间做出明智的选择是一项需要长期培养的技能。

3.2.3 错误检测和处理的实现

我们已经了解了一些处理错误的哲学方法。现在，让我们将注意力转向程序员在实现错误检测和处理代码时所面临的选择。

3.2.3.1 错误返回代码

处理错误的一种常见方法是从首次检测到问题的函数返回某种失败代码。这可以是一个指示成功或失败的布尔值，也可以是一个“不可能”的值，即超出正常返回结果范围的值。例如，返回正整数或浮点值的函数可以返回负值来指示发生了错误。比布尔值或“不可能”返回值更好的是，函数可以设计为返回一个枚举值来指示成功或失败。这将错误代码与函数的输出清晰地区分开来，并且可以在失败时指示问题的确切性质（例如，enum Error { kSuccess,

`kAssetNotFound, kInvalidRange, ... }`).

调用函数应该拦截错误返回码并采取适当的措施。它可能会立即处理错误。或者，它可能会绕过问题，完成自身的执行，然后将错误代码传递给调用它的函数。

reference has been repaired. Of course, for particularly egregious problems it's fine to just spew an error message and crash. There's no silver bullet for all kinds of problems, and your judgment about what type of error handling approach to apply to a given situation will improve with experience.

3.2.2.3 Handling Programmer Errors

The best way to detect and handle programmer errors (a.k.a. bugs) is often to embed error-checking code into your source code and arrange for failed error checks to halt the program. Such a mechanism is known as an *assertion system*; we'll investigate assertions in detail in Section 3.2.3.3. Of course, as we said above, one programmer's user error is another programmer's bug; hence, assertions are not always the right way to handle every programmer error. Making a judicious choice between an assertion and a more graceful error-handling technique is a skill that one develops over time.

3.2.3 Implementation of Error Detection and Handling

We've looked at some philosophical approaches to handling errors. Now let's turn our attention to the choices we have as programmers when it comes to implementing error detection and handling code.

3.2.3.1 Error Return Codes

A common approach to handling errors is to return some kind of failure code from the function in which the problem is first detected. This could be a Boolean value indicating success or failure, or it could be an "impossible" value, one that is outside the range of normally returned results. For example, a function that returns a positive integer or floating-point value could return a negative value to indicate that an error occurred. Even better than a Boolean or an "impossible" return value, the function could be designed to return an enumerated value to indicate success or failure. This clearly separates the error code from the output(s) of the function, and the exact nature of the problem can be indicated on failure (e.g., `enum Error { kSuccess, kAssetNotFound, kInvalidRange, ... }`).

The calling function should intercept error return codes and act appropriately. It might handle the error immediately. Or, it might work around the problem, complete its own execution and then pass the error code on to whatever function called *it*.

3.2.3.2 异常

错误返回码是一种简单可靠的错误信息传递和响应方式。然而，错误返回码也有其缺点。错误返回码最大的问题或许在于，检测到错误的函数可能与能够处理该问题的函数完全无关。在最坏的情况下，调用堆栈中深度为 40 次调用的函数可能会检测到一个只能由顶级游戏循环或 main() 处理的问题。在这种情况下，调用堆栈上的 40 个函数中的每一个都需要重新编写，以便能够将相应的错误代码一直传递回顶级错误处理函数。

解决这个问题的一种方法是抛出一个异常。异常处理是 C++ 的一个非常强大的特性。它允许检测到问题的函数将错误传达给其余代码，而无需知道哪个函数可能处理该错误。当抛出异常时，关于错误的相关信息被放入程序员选择的数据对象中，称为异常对象。然后，调用堆栈会自动展开，以搜索将其调用包装在 try catch 块中的调用函数。如果找到 try catch 块，则将异常对象与所有可能的 catch 子句进行匹配，如果找到匹配项，则执行相应的 catch 代码块。在堆栈展开过程中，将根据需要调用任何自动变量的析构函数。

能够以如此清晰的方式将错误检测与错误处理分离无疑非常有吸引力，而异常处理对于某些软件项目来说是一个绝佳的选择。然而，异常处理确实会给程序增加一些开销。任何包含 try catch 块的函数的堆栈框架都必须进行扩充，以包含堆栈展开过程所需的额外信息。此外，即使程序中只有一个函数（或程序链接的库）使用了异常处理，整个程序都必须使用异常处理——因为编译器无法知道抛出异常时调用堆栈上哪些函数可能位于您的上方。

也就是说，可以对使用异常处理的一个或多个库进行“沙盒化”，以避免整个游戏引擎在启用异常的情况下编写。为此，您需要将所有 API 调用包装到相关库的函数中，这些函数在启用异常处理的转换单元中实现。每个函数都会在 try/catch 块中捕获所有可能的异常，并将其转换为错误返回代码。因此，任何链接到您的包装器库的代码都可以安全地禁用异常处理。

可以说，比开销问题更重要的是，

3.2.3.2 Exceptions

Error return codes are a simple and reliable way to communicate and respond to error conditions. However, error return codes have their drawbacks. Perhaps the biggest problem with error return codes is that the function that detects an error may be totally unrelated to the function that is capable of handling the problem. In the worst-case scenario, a function that is 40 calls deep in the call stack might detect a problem that can only be handled by the top-level game loop, or by `main()`. In this scenario, every one of the 40 functions on the call stack would need to be written so that it can pass an appropriate error code all the way back up to the top-level error-handling function.

One way to solve this problem is to throw an exception. *Exception handling* is a very powerful feature of C++. It allows the function that detects a problem to communicate the error to the rest of the code without knowing anything about which function might handle the error. When an exception is thrown, relevant information about the error is placed into a data object of the programmer's choice known as an *exception object*. The call stack is then automatically unwound, in search of a calling function that has wrapped its call in a `try-catch` block. If a `try-catch` block is found, the exception object is matched against all possible `catch` clauses, and if a match is found, the corresponding `catch`'s code block is executed. The destructors of any automatic variables are called as needed during the stack unwinding process.

The ability to separate error detection from error handling in such a clean way is certainly attractive, and exception handling is an excellent choice for some software projects. However, exception handling does add some overhead to the program. The stack frame of any function that contains a `try-catch` block must be augmented to contain additional information required by the stack unwinding process. Also, if even one function in your program (or a library that your program links with) uses exception handling, your entire program must use exception handling—the compiler can't know which functions might be above you on the call stack when you throw an exception.

That said, it is possible to "sandbox" a library or libraries that make use of exception handling, in order to avoid your entire game engine having to be written with exceptions enabled. To do this, you would wrap all the API calls into the libraries in question in functions that are implemented in a translation unit that has exception handling enabled. Each of these functions would catch all possible exceptions in a try/catch block and convert them into error return codes. Any code that links with your wrapper library can therefore safely disable exception handling.

Arguably more important than the overhead issue is the fact that excep-

tions 在某些方面并不比 goto 语句好。微软和 Fog Creek Software 的著名人物 Joel Spolsky 认为异常实际上比 goto 更糟糕，因为它们在源代码中不容易看到。如果一个既不抛出也不捕获异常的函数发现自己夹在调用堆栈中的此类函数之间，它就可能参与堆栈展开过程。展开过程本身并不完美：除非程序员考虑到可能抛出异常的每一种可能方式并进行适当的处□□理，否则您的软件很容易处于无效状态。这会使编写健壮的软件变得困难。当存在抛出异常的可能性时，代码库中的几乎每个函数都需要具有健壮性，以应对每次进行函数调用时从其下方抽出的地毯和销毁其所有本地对象的情况。

异常处理的另一个问题是其成本。虽然理论上现代异常处理框架在无错误情况下不会引入额外的运行时开销，但实际情况并非如此。例如，编译器在函数中添加的用于在发生异常时展开调用堆栈的代码往往会导致代码大小整体增加。这可能会降低指令缓存的性能，或者导致编译器决定不内联原本应该内联的函数。

显然，有一些非常有力的理由支持完全关闭游戏引擎中的异常处理。顽皮狗采用了这种方法，我在 Electronic Arts 和 Midway 参与的大多数项目也采用了这种方法。作为 Insomniac Games 的引擎总监，Mike Acton 曾多次明确表示反对在运行时游戏代码中使用异常处理。JPL 和 NASA 也不允许在其关键任务嵌入式软件中使用异常处理，大概是出于与我们在游戏行业中倾向于避免使用异常处理相同的原因。话虽如此，您的情况可能会有所不同。没有完美的工具，也没有唯一正确的方法。如果使用得当，异常可以使您的代码更易于编写和使用；只是要小心！

网上有很多关于这个话题的有趣文章。这里有一篇很好的帖子，涵盖了辩论双方的大部分关键问题：

- http://www.joelonsoftware.com/items/2003/10/13.html
- http://www.nedbatchelder.com/text/exceptions-vs-status.html
- http://www.joelonsoftware.com/items/2003/10/15.html

异常和 RAII

“资源获取即初始化”模式（RAII，参见第 3.1.1.6 节）通常与异常处理结合使用：构造函数尝试

tions are in some ways no better than `goto` statements. Joel Spolsky of Microsoft and Fog Creek Software fame argues that exceptions are in fact *worse* than `gotos` because they aren't easily seen in the source code. A function that neither throws nor catches exceptions may nevertheless become involved in the stack-unwinding process, if it finds itself sandwiched between such functions in the call stack. And the unwinding process is itself imperfect: Your software can easily be left in an invalid state unless the programmer considers every possible way that an exception can be thrown, and handles it appropriately. This can make writing robust software difficult. When the possibility for exception throwing exists, pretty much every function in your codebase needs to be robust to the carpet being pulled out from under it and all its local objects destroyed whenever it makes a function call.

Another issue with exception handling is its cost. Although in theory modern exception handling frameworks don't introduce additional runtime overhead in the error-free case, this is not necessarily true in practice. For example, the code that the compiler adds to your functions for unwinding the call stack when an exception occurs tends to produce an overall increase in code size. This might degrade I-cache performance, or cause the compiler to decide not to inline a function that it otherwise would have.

Clearly there are some pretty strong arguments for turning *off* exception handling in your game engine altogether. This is the approach employed at Naughty Dog and also on most of the projects I've worked on at Electronic Arts and Midway. In his capacity as Engine Director at Insomniac Games, Mike Acton has clearly stated his objection to the use of exception handling in runtime game code on numerous occasions. JPL and NASA also disallow exception handling in their mission-critical embedded software, presumably for the same reasons we tend to avoid it in the game industry. That said, your mileage certainly may vary. There is no perfect tool and no one right way to do anything. When used judiciously, exceptions *can* make your code easier to write and work with; just be careful out there!

There are many interesting articles on this topic on the web. Here's one good thread that covers most of the key issues on both sides of the debate:

- http://www.joelonsoftware.com/items/2003/10/13.html
- http://www.nedbatchelder.com/text/exceptions-vs-status.html
- http://www.joelonsoftware.com/items/2003/10/15.html

Exceptions and RAII

The "resource acquisition is initialization" pattern (RAII, see Section 3.1.1.6) is often used in conjuction with *exception handling:* The constructor attempts to

获取所需资源，如果失败则抛出异常。这样做是为了避免在对象创建后进行 if 检查来测试其状态——如果构造函数返回且没有抛出异常，我们就可以肯定资源已成功获取。

然而，即使没有异常，RAII 模式也可以使用。它只需要一点规则，在首次创建每个新资源对象时检查其状态。之后，就可以享受 RAII 的所有其他好处了。（也可以用断言失败来代替异常，以指示某些类型的资源获取失败。）

3.2.3.3 断言

断言是一行用于检查表达式的代码。如果表达式的值为真，则不执行任何操作。如果表达式的值为假，则程序停止运行，打印一条消息，并在可能的情况下调用调试器。

断言检查程序员的假设。它们就像漏洞的地雷。它们在代码首次编写时进行检查，以确保其正常运行。它们还能确保原始假设能够长期有效，即使周围的代码不断变化和发展。例如，如果程序员修改了曾经可以运行的代码，却意外地违反了原始假设，他们就等于触雷了。这会立即告知程序员问题所在，并允许他们以最小的麻烦纠正这种情况。如果没有断言，漏洞往往会“隐藏”起来，并在以后以难以追踪且耗时的方式显现出来。但是，如果代码中嵌入了断言，漏洞就会在引入的那一刻就暴露出来——这通常是修复问题的最佳时机，而导致问题的代码更改在程序员的脑海中仍然记忆犹新。Steve Maguire 在其必读著作《编写可靠的代码》[35] 中对断言进行了深入的讨论。

断言检查的成本通常在开发过程中可以承受，但在游戏发布前移除断言，如果有必要，可以挽回那一点点关键的性能损失。因此，断言的实现方式通常允许在非调试构建配置中将检查从可执行文件中移除。在 C 语言中，assert() 宏由标准库头文件 <assert.h> 提供；在 C++ 中，它由 <cassert> 头文件提供。

标准库对 assert() 的定义使其在调试版本（使用定义的 DEBUG 预处理器符号构建）中定义，并且

acquire the desired resource, and throws an exception if it fails to do so. This is done to avoid the need for an if check to test the status of the object after it has been created—if the constructor returns without throwing an exception, we know for certain that the resource was successfully acquired.

However, the RAII pattern can be used even without exceptions. All it requires is a little discipline to check the status of each new resource object when it is first created. After that, all of the other benefits of RAII can be reaped. (Exceptions can also be replaced by assertion failures to signal the failure of some kinds of resource acquisitions.)

3.2.3.3 Assertions

An *assertion* is a line of code that checks an expression. If the expression evaluates to true, nothing happens. But if the expression evaluates to false, the program is stopped, a message is printed and the debugger is invoked if possible.

Assertions check a programmer's assumptions. They act like *land mines* for bugs. They check the code when it is first written to ensure that it is functioning properly. They also ensure that the original assumptions continue to hold for long periods of time, even when the code around them is constantly changing and evolving. For example, if a programmer changes code that used to work, but accidentally violates its original assumptions, they'll hit the land mine. This immediately informs the programmer of the problem and permits him or her to rectify the situation with minimum fuss. Without assertions, bugs have a tendency to "hide out" and manifest themselves later in ways that are difficult and time-consuming to track down. But with assertions embedded in the code, bugs announce themselves the moment they are introduced—which is usually the best moment to fix the problem, while the code changes that caused the problem are fresh in the programmer's mind. Steve Maguire provides an in-depth discussion of assertions in his must-read book, *Writing Solid Code* [35].

The cost of the assertion checks can usually be tolerated during development, but stripping out the assertions prior to shipping the game can buy back that little bit of crucial performance if necessary. For this reason assertions are generally implemented in such a way as to allow the checks to be stripped out of the executable in non-debug build configurations. In C, an `assert()` macro is provided by the standard library header file `<assert.h>`; in C++, it's provided by the `<cassert>` header.

The standard library's definition of `assert()` causes it to be defined in debug builds (builds with the `DEBUG` preprocessor symbol defined) and

在非调试版本（定义了 NDEBUG 预处理器符号的版本）中剥离。在游戏引擎中，您可能需要更细粒度地控制哪些构建配置保留断言，以及哪些配置将其剥离。例如，您的游戏可能不仅支持调试和开发构建配置 - 您可能还拥有启用了全局优化的发布版本，甚至可能拥有供配置文件引导优化工具使用的 PGO 版本（参见第 2.2.4 节）。或者，您可能还想定义不同“风格”的断言 - 一些即使在游戏的发布版本中也始终保留，而其他一些则从非发布版本中剥离。出于这些原因，让我们看一下如何使用 C/C++ 预处理器实现您自己的 ASSERT() 宏。

断言实现

断言通常通过以下组合来实现：一个 #define 宏（用于判断 if / else 子句）、一个函数（用于在断言失败时调用，即表达式结果为 false ）以及一小段汇编代码（用于在附加调试器时暂停程序并中断程序）。以下是一个典型的实现：

```
#if ASSERTIONS_ENABLED

// define some inline assembly that causes a break
// into the debugger -- this will be different on each
// target CPU
#define debugBreak() asm { int 3 }

// check the expression and fail if it is false
#define ASSERT(expr) \
    if (expr) { } \
    else \
    { \
        reportAssertionFailure(#expr, \
                 __FILE__, __LINE__); \
        debugBreak(); \
    }

#else

#define ASSERT(expr)  // evaluates to nothing

#endif
```

让我们分解这个定义，以便了解它是如何工作的：

stripped in non-debug builds (builds with the `NDEBUG` preprocessor symbol defined). In a game engine, you may want finer-grained control over which build configurations retain assertions, and which configurations strip them out. For example, your game might support more than just a debug and development build configuration—you might also have a shipping build with global optimizations enabled, and perhaps even a PGO build for use by profile-guided optimization tools (see Section 2.2.4). Or you might also want to define different “flavors” of assertions—some that are always retained even in the shipping version of your game, and others that are stripped out of the non-shipping build. For these reasons, let’s take a look at how you can implement your own `ASSERT()` macro using the C/C++ preprocessor.

Assertion Implementation

Assertions are usually implemented via a combination of a `#defined` macro that evaluates to an `if/else` clause, a function that is called when the assertion fails (the expression evaluates to `false`), and a bit of assembly code that halts the program and breaks into the debugger when one is attached. Here’s a typical implementation:

```
#if ASSERTIONS_ENABLED

// define some inline assembly that causes a break
// into the debugger -- this will be different on each
// target CPU
#define debugBreak() asm { int 3 }

// check the expression and fail if it is false
#define ASSERT(expr) \
    if (expr) { } \
    else \
    { \
        reportAssertionFailure(#expr, \
                 __FILE__, __LINE__); \
        debugBreak(); \
    }

#else

#define ASSERT(expr)  // evaluates to nothing

#endif
```

Let’s break down this definition so we can see how it works:

• 外部 #if / #else / #endif 用于从代码库中剥离断言。当 ASSERTIONS_ENABLED 非零时，ASSERT() 宏将完整定义，代码中的所有断言检查都将包含在程序中。但是，当断言被关闭时，ASSERT(expr) 的计算结果为空，代码中所有与该宏相关的实例都会被有效移除。

• debugBreak() 宏会计算出所需的汇编语言指令，以使程序暂停并让调试器接管（如果连接了调试器）。不同 CPU 的执行方式有所不同，但通常都是一条汇编指令。

• ASSERT() 宏本身使用完整的 if / else 语句定义（而不是单独的 if ）。这样做是为了让宏可以在任何上下文中使用，即使在其他未加括号的 if / else 语句中也可以使用。

下面是使用单独的 if 定义 ASSERT() 时会发生什么情况的示例：

```
// WARNING: NOT A GOOD IDEA!
#define ASSERT(expr)  if (!(expr)) debugBreak()

void f()
{
    if (a < 5)
        ASSERT(a >= 0);
    else
        doSomething(a);
}
```

这扩展为以下错误代码：

```
void f()
{
    if (a < 5)
        if (!(a >= 0))
            debugBreak();
        else // oops! bound to the wrong if()!
            doSomething(a);
}
```

• ASSERT() 宏的 else 子句有两个作用。首先，它向程序员显示某种消息，指出哪里出了问题；然后，它中断并进入调试器。请注意，#expr 是消息显示函数的第一个参数。# 预处理运算符将表达式 expr 转换为字符串，从而允许将其作为断言失败消息的一部分打印出来。

- The outer `#if/#else/#endif` is used to strip assertions from the code base. When `ASSERTIONS_ENABLED` is nonzero, the `ASSERT()` macro is defined in its full glory, and all assertion checks in the code will be included in the program. But when assertions are turned off, `ASSERT(expr)` evaluates to nothing, and all instances of it in the code are effectively removed.
- The `debugBreak()` macro evaluates to whatever assembly-language instructions are required in order to cause the program to halt and the debugger to take charge (if one is connected). This differs from CPU to CPU, but it is usually a single assembly instruction.
- The `ASSERT()` macro itself is defined using a full `if/else` statement (as opposed to a lone `if`). This is done so that the macro can be used in any context, even within *other* unbracketed `if/else` statements.

Here's an example of what would happen if `ASSERT()` were defined using a solitary `if`:

```
// WARNING: NOT A GOOD IDEA!
#define ASSERT(expr)  if (!(expr)) debugBreak()

void f()
{
    if (a < 5)
        ASSERT(a >= 0);
    else
        doSomething(a);
}
```

This expands to the following *incorrect* code:

```
void f()
{
    if (a < 5)
        if (!(a >= 0))
            debugBreak();
        else // oops! bound to the wrong if()!
            doSomething(a);
}
```

- The `else` clause of an `ASSERT()` macro does two things. It displays some kind of message to the programmer indicating what went wrong, and then it breaks into the debugger. Notice the use of `#expr` as the first argument to the message display function. The pound (#) preprocessor operator causes the expression `expr` to be turned into a string, thereby allowing it to be printed out as part of the assertion failure message.

• 还要注意 __FILE__ 和 __LINE__ 的使用。这些编译器定义的宏神奇地包含了 .cpp 文件名以及它们所在代码行的行号。通过将它们传入消息显示函数，我们可以打印出问题的确切位置。

我强烈建议在代码中使用断言。但是，务必注意它们的性能成本。您可能需要考虑定义两种断言宏。常规的 ASSERT() 宏可以在所有版本中保持活动状态，这样即使不在调试模式下运行，也可以轻松捕获错误。第二个断言宏，例如 SLOW_ASSERT()，可以仅在调试版本中激活。然后，可以在断言检查成本过高而无法包含在发布版本中的地方使用此宏。显然，SLOW_ASSERT() 的实用性较低，因为它已从测试人员每天玩的游戏版本中删除。但至少这些断言在程序员调试代码时会生效。

正确使用断言也至关重要。它们应该用于捕获程序本身的错误，而不是用于捕获用户错误。此外，断言失败时应始终导致整个游戏停止。允许测试人员、美术师、设计师和其他非工程师跳过断言通常是一个坏主意。（这有点像狼来了：如果断言可以被跳过，那么它们就不再具有任何意义，变得无效。）换句话说，断言应该只用于捕获致命错误。如果可以跳过断言继续运行，那么最好以其他方式通知用户错误，例如通过屏幕消息或一些难看的亮橙色 3D 图形。

编译时断言

正如我们目前所讨论的，断言的一个弱点是，它所编码的条件只能在运行时检查。为了检查断言的条件，我们必须运行程序，并且相关的代码路径必须实际执行。

有时，我们在断言中检查的条件涉及在编译时完全已知的信息。例如，假设我们正在定义一个结构体，由于某种原因，它的大小需要恰好为 128 字节。我们希望添加一个断言，这样如果其他程序员（或未来的你自己）决定更改该结构体的大小，编译器就会给我们一个错误信息。换句话说，我们想写这样的东西：

- Notice also the use of `__FILE__` and `__LINE__`. These compiler-defined macros magically contain the .cpp file name and line number of the line of code on which they appear. By passing them into our message display function, we can print the exact location of the problem.

I highly recommend the use of assertions in your code. However, it's important to be aware of their performance cost. You may want to consider defining two kinds of assertion macros. The regular `ASSERT()` macro can be left active in *all* builds, so that errors are easily caught even when not running in debug mode. A second assertion macro, perhaps called `SLOW_ASSERT()`, could be activated only in debug builds. This macro could then be used in places where the cost of assertion checking is too high to permit inclusion in release builds. Obviously `SLOW_ASSERT()` is of lower utility, because it is stripped out of the version of the game that your testers play every day. But at least these assertions become active when programmers are debugging their code.

It's also extremely important to use assertions properly. They should be used to catch bugs in the program itself—*never* to catch user errors. Also, assertions should always cause the entire game to halt when they fail. It's usually a bad idea to allow assertions to be skipped by testers, artists, designers and other non-engineers. (This is a bit like the boy who cried wolf: if assertions can be skipped, then they cease to have any significance, rendering them ineffective.) In other words, assertions should only be used to catch fatal errors. If it's OK to continue past an assertion, then it's probably better to notify the user of the error in some other way, such as with an on-screen message, or some ugly bright-orange 3D graphics.

Compile-Time Assertions

One weakness of assertions, as we've discussed them thus far, is that the conditions encoded within them are only checked at runtime. We have to run the program, *and* the code path in question must actually *execute*, in order for an assertion's condition to be checked.

Sometimes the condition we're checking within an assertion involves information that is entirely known at compile time. For example, let's say we're defining a struct that for some reason needs to be exactly 128 bytes in size. We want to add an assertion so that if another programmer (or a future version of yourself) decides to change the size of the struct, the compiler will give us an error message. In other words, we'd like to write something like this:

```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // etc.
};

// sadly this doesn't work...
ASSERT(sizeof(NeedsToBe128Bytes) == 128);
```

问题当然在于 ASSERT() （或 assert() ）宏需要在运行时可执行，而我们甚至不能将可执行代码放在 .cpp 文件中函数定义之外的全局作用域中。解决这个问题的方法是使用编译时断言，也称为静态断言。

从 C++11 开始，标准库为我们定义了一个名为 static_assert() 的宏。因此，我们可以将上面的例子重写如下：

```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // etc.
};

static_assert(sizeof(NeedsToBe128Bytes) == 128,
    "wrong size");
```

如果您不使用 C++11，您可以随时创建自己的 STATIC_ ASSERT() 宏。它可以通过多种不同的方式实现，但基本思想始终相同：该宏在代码中声明：(a) 在文件范围内合法；(b) 在编译时而不是运行时计算所需表达式的值；(c) 当且仅当表达式为 false 时产生编译错误。一些定义 STATIC_ASSERT() 的方法依赖于编译器特定的细节，但这里有一种相当可移植的定义方法：

```
#define _ASSERT_GLUE(a, b)  a ## b
#define ASSERT_GLUE(a, b)   _ASSERT_GLUE(a, b)

#define STATIC_ASSERT(expr) \
    enum \
    { \
        ASSERT_GLUE(g_assert_fail_, __LINE__) \
            = 1 / (int)(!!(expr)) \
    }

STATIC_ASSERT(sizeof(int) == 4);   // should pass
STATIC_ASSERT(sizeof(float) == 1); // should fail
```



```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // etc.
};

// sadly this doesn't work...
ASSERT(sizeof(NeedsToBe128Bytes) == 128);
```

The problem of course is that the `ASSERT()` (or `assert()`) macro needs to be executable at runtime, and one can't even put executable code at global scope in a .cpp file outside of a function definition. The solution to this problem is a *compile-time assertion*, also known as a *static assertion*.

Starting with C++11, the standard library defines a macro named `static_assert()` for us. So we can re-write the example above as follows:

```
struct NeedsToBe128Bytes
{
    U32    m_a;
    F32    m_b;
    // etc.
};

static_assert(sizeof(NeedsToBe128Bytes) == 128,
    "wrong size");
```

If you're not using C++11, you can always roll your own `STATIC_ASSERT()` macro. It can be implemented in a number of different ways, but the basic idea is always the same: The macro places a declaration into your code that (a) is legal at file scope, (b) evaluates the desired expression at compile time rather than runtime, and (c) produces a compile error if and only if the expression is false. Some methods of defining `STATIC_ASSERT()` rely on compiler-specific details, but here's one reasonably portable way to define it:

```
#define _ASSERT_GLUE(a, b)  a ## b
#define ASSERT_GLUE(a, b)   _ASSERT_GLUE(a, b)

#define STATIC_ASSERT(expr) \
    enum \
    { \
        ASSERT_GLUE(g_assert_fail_, __LINE__) \
            = 1 / (int)(!!(expr)) \
    }

STATIC_ASSERT(sizeof(int) == 4);   // should pass
STATIC_ASSERT(sizeof(float) == 1); // should fail
```

其工作原理是定义一个包含单个枚举器的匿名枚举。通过将固定前缀（例如 g_assert_fail_）“粘合”到唯一后缀（在本例中为调用 STATIC_ASSERT() 宏的行号）上，使枚举器的名称在翻译单元内是唯一的。枚举器的值设置为 1 / (!!(expr))。双重否定 !! 确保 expr 具有布尔值。然后，该值被转换为 int，根据表达式是 true 还是 false 分别产生 1 或 0。如果表达式为真，则枚举器将被设置为值 1/1，即 1。但是，如果表达式为假，我们将要求编译器将枚举器设置为值 1/0，这是非法的，并将触发编译错误。

当我们上面定义的 STATIC_ASSERT() 宏失败时，Visual Studio 2015 会产生如下编译时错误消息：

```
1>test.cpp(48): error C2131: expression did not evaluate to
                             a constant
1>  test.cpp(48): note: failure was caused by an undefined
                        arithmetic operation
```

这是使用模板特化定义 STATIC_ASSERT() 的另一种方法。在本例中，我们首先检查是否使用了 C++11 或更高版本。如果是，我们将使用标准库的 static_assert() 实现，以实现最大程度的可移植性。否则，我们将回退到自定义实现。

```
#ifdef __cplusplus
    #if __cplusplus >= 201103L
        #define STATIC_ASSERT(expr) \
            static_assert(expr, \
                          "static assert failed:" \
                          #expr)
    #else
        // declare a template but only define the
        // true case (via specialization)
        template<bool> class TStaticAssert;
        template<> class TStaticAssert<true> {};

        #define STATIC_ASSERT(expr) \
            enum \
            { \
                ASSERT_GLUE(g_assert_fail_, __LINE__) \
                    = sizeof(TStaticAssert<!!(expr)>) \
            }
    #endif
#endif
```


This works by defining an anonymous enumeration containing a single enumerator. The name of the enumerator is made unique (within the translation unit) by “gluing” a fixed prefix such as `g_assert_fail_` to a unique suffix—in this case, the line number on which the `STATIC_ASSERT()` macro is invoked. The value of the enumerator is set to `1 / (!!(expr))`. The double negation `!!` ensures that `expr` has a Boolean value. This value is then cast to an `int`, yielding either `1` or `0` depending on whether the expression is `true` or `false`, respectively. If the expression is true, the enumerator will be set to the value `1/1` which is one. But if the expression is false, we'll be asking the compiler to set the enumerator to the value `1/0` which is illegal, and will trigger a compile error.

When our `STATIC_ASSERT()` macro as defined above fails, Visual Studio 2015 produces a compile-time error message like this:

```
1>test.cpp(48): error C2131: expression did not evaluate to
                             a constant
1>  test.cpp(48): note: failure was caused by an undefined
                        arithmetic operation
```

Here's another way to define `STATIC_ASSERT()` using template specialization. In this example, we first check to see if we're using C++11 or beyond. If so, we use the standard library's implementation of `static_assert()` for maximum portability. Otherwise we fall back to our custom implementation.

```
#ifdef __cplusplus
    #if __cplusplus >= 201103L
        #define STATIC_ASSERT(expr) \
            static_assert(expr, \
                          "static assert failed:" \
                          #expr)
    #else
        // declare a template but only define the
        // true case (via specialization)
        template<bool> class TStaticAssert;
        template<> class TStaticAssert<true> {};

        #define STATIC_ASSERT(expr) \
            enum \
            { \
                ASSERT_GLUE(g_assert_fail_, __LINE__) \
                    = sizeof(TStaticAssert<!!(expr)>) \
            }
    #endif
#endif
```

这种使用模板特化的实现可能比以前使用除以零的实现更可取，因为它在 Visual Studio 2015 中产生了稍微好一点的错误消息：

```
1>test.cpp(48): error C2027: use of undefined type
                            'TStaticAssert<false>'
1>test.cpp(48): note: see declaration of
                        'TStaticAssert<false>'
```

但是，每个编译器处理错误报告的方式不同，因此您的情况可能会有所不同。有关编译时断言的更多实现思路，一个很好的参考是 http://www.pixelbeat.org/programming/gcc/static_assert.html。

3.3 数据、代码和内存布局

3.3.1 数字表示

数字是游戏引擎开发（以及一般的软件开发）的核心。每位软件工程师都应该了解计算机是如何表示和存储数字的。本节将为您提供本书其余部分所需的基础知识。

3.3.1.1 数字基数

人们最自然地使用十进制（也称为十进制表示法）来思考。这种表示法使用十个不同的数字（0 到 9），从右到左的每个数字代表 10 的下一个最高次方。例如，数字 $7803 = (7 \times 10^3) + (8 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) = 7000 + 800 + 0 + 3$。

在计算机科学中，诸如整数和实值数之类的数学量需要存储在计算机内存中。众所周知，计算机以二进制格式存储数字，这意味着只有 0 和 1 这两个数字可用。我们称之为以二进制为基数的表示法，因为从右到左的每个数字代表 2 的下一个最高幂。计算机科学家有时会使用前缀“0b”来表示二进制数。例如，二进制数 0b1101 相当于十进制 13，因为 $0b1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$。

计算机界流行的另一种常见表示法是十六进制，即以 16 为基数。这种表示法使用 0 到 9 的 10 个数字和 A 到 F 的六个字母；字母 A 到 F 分别替换十进制值 10 到 15。在 C 和 C++ 语言中，使用前缀“0x”表示十六进制数。

This implementation, using template specialization, may be preferable to the previous one using division by zero, because it produces a slightly better error message in Visual Studio 2015:

```
1>test.cpp(48): error C2027: use of undefined type
                            'TStaticAssert<false>'
1>test.cpp(48): note: see declaration of
                        'TStaticAssert<false>'
```

However, each compiler handles error reporting differently, so your mileage may vary. For more implementation ideas for compile-time assertions, one good reference is http://www.pixelbeat.org/programming/gcc/static_assert.html.

3.3 Data, Code and Memory Layout

3.3.1 Numeric Representations

Numbers are at the heart of everything that we do in game engine development (and software development in general). Every software engineer should understand how numbers are represented and stored by a computer. This section will provide you with the basics you'll need throughout the rest of the book.

3.3.1.1 Numeric Bases

People think most naturally in *base ten*, also known as *decimal notation*. In this notation, ten distinct digits are used (0 through 9), and each digit from right to left represents the next highest power of 10. For example, the number $7803 = (7 \times 10^3) + (8 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) = 7000 + 800 + 0 + 3$.

In computer science, mathematical quantities such as integers and real-valued numbers need to be stored in the computer's memory. And as we know, computers store numbers in *binary* format, meaning that only the two digits 0 and 1 are available. We call this a *base-two* representation, because each digit from right to left represents the next highest power of 2. Computer scientists sometimes use a prefix of "0b" to represent binary numbers. For example, the binary number 0b1101 is equivalent to decimal 13, because $0b1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$.

Another common notation popular in computing circles is *hexadecimal*, or *base 16*. In this notation, the 10 digits 0 through 9 and the six letters A through F are used; the letters A through F replace the decimal values 10 through 15, respectively. A prefix of "0x" is used to denote hex numbers in the C and C++

编程语言。这种表示法很流行，因为计算机通常将数据存储在 8 位一组（称为字节）中，而一个十六进制数字正好代表 4 位，所以一对十六进制数字代表一个字节。例如，值 0xFF = 0b11111111 = 255 是 8 位（1 字节）可以存储的最大数字。十六进制数中的每个数字（从右到左）代表 16 的下一个幂。例如，0xB052 = (11 × 16 3) + (0 × 16 2) + (5 × 16 1) + (2 × 16 0) = (11 × 4096) + (0 × 256) + (5 × 16) + (2 × 1) = 45,138。

3.3.1.2 有符号整数和无符号整数

在计算机科学中，我们同时使用有符号整数和无符号整数。当然，“无符号整数”这个术语实际上有点用词不当——在数学中，整数或自然数的范围是从0（或1）到正无穷大，而整数的范围是从负无穷大到正无穷大。尽管如此，本书仍将使用计算机科学术语，并坚持使用“有符号整数”和“无符号整数”这两个术语。

大多数现代个人电脑和游戏机都能轻松处理 32 位或 64 位宽度的整数（尽管 8 位和 16 位整数在游戏编程中也大量使用）。为了表示 32 位无符号整数，我们只需使用二进制表示法对其进行编码即可（参见上文）。32 位无符号整数的可能值范围是 0x00000000（0）到

0xFFFFFFFF （4,294,967,295）。

为了用 32 位表示有符号整数，我们需要一种区分正值和负值的方法。一种简单的方法称为符号和数值编码，它将最高有效位保留为符号位。当该位为零时，值为正；当该位为一时，值为负。这样就只剩下 31 位来表示数值的数值，有效地将数值的可能范围减半（但允许每个不同数值的正负形式，包括零）。

大多数微处理器使用一种稍微高效的技术来编码负整数，称为二进制补码表示法。这种表示法只有一种表示零的方法，而简单的符号位则有两种表示方法（正零和负零）。在 32 位二进制补码表示法中，值 0xFFFFFFFF 被解释为 − 1，负值从 − 1 开始递减。任何设置了最高有效位的值都被视为负数。因此，从 0x00000000（0）到 0x7FFFFFFF（2,147,483,647）的值表示正整数，而 0x80000000（−2,147,483,648）到

0xFFFFFFFF（−1）表示负整数。

programming languages. This notation is popular because computers generally store data in groups of 8 bits known as *bytes*, and since a single hexadecimal digit represents 4 bits exactly, a *pair* of hex digits represents a byte. For example, the value 0xFF = 0b11111111 = 255 is the largest number that can be stored in 8 bits (1 byte). Each digit in a hexadecimal number, from right to left, represents the next power of 16. So, for example, 0xB052 $= (11 \times 16^3) + (0 \times 16^2) + (5 \times 16^1) + (2 \times 16^0) = (11 \times 4096) + (0 \times 256) + (5 \times 16) + (2 \times 1) = 45{,}138$.

3.3.1.2 Signed and Unsigned Integers

In computer science, we use both signed and unsigned integers. Of course, the term “unsigned integer” is actually a bit of a misnomer—in mathematics, the *whole numbers* or *natural numbers* range from 0 (or 1) up to positive infinity, while the *integers* range from negative infinity to positive infinity. Nevertheless, we'll use computer science lingo in this book and stick with the terms “signed integer” and “unsigned integer.”

Most modern personal computers and game consoles work most easily with integers that are 32 bits or 64 bits wide (although 8- and 16-bit integers are also used a great deal in game programming as well). To represent a 32-bit unsigned integer, we simply encode the value using binary notation (see above). The range of possible values for a 32-bit unsigned integer is 0x00000000 (0) to 0xFFFFFFFF (4,294,967,295).

To represent a *signed* integer in 32 bits, we need a way to differentiate between positive and negative vales. One simple approach called the *sign and magnitude* encoding reserves the most significant bit as a *sign bit*. When this bit is zero, the value is positive, and when it is one, the value is negative. This leaves us 31 bits to represent the magnitude of the value, effectively cutting the range of possible magnitudes in half (but allowing both positive and negative forms of every distinct magnitude, including zero).

Most microprocessors use a slightly more efficient technique for encoding negative integers, called *two's complement* notation. This notation has only one representation for the value zero, as opposed to the two representations possible with simple sign bit (positive zero and negative zero). In 32-bit two's complement notation, the value 0xFFFFFFFF is interpreted to mean −1, and negative values count down from there. Any value with the most significant bit set is considered negative. So values from 0x00000000 (0) to 0x7FFFFFFF (2,147,483,647) represent positive integers, and 0x80000000 (−2,147,483,648) to 0xFFFFFFFF (−1) represent negative integers.

图 3.4. 具有 16 位幅度和 16 位小数的定点表示法。

3.3.1.3 定点表示法

整数非常适合表示整数，但要表示分数和无理数，我们需要一种不同的格式来表达小数点的概念。

计算机科学家早期采用的一种方法是使用定点表示法。在这种表示法中，可以任意选择用多少位来表示数字的整数部分，其余位则用于表示小数部分。从左到右（即从最高有效位到最低有效位），数值位表示 2 的幂次方（...，16，8，4，2，1），而小数位表示 2 的逆幂（1 2 ，1 4 ，1 8 ，1 16 ，...）。例如，要将数字 -173.25 存储为 32 位定点表示法，其中 1 位符号位、16 位数值位和 15 位小数位，我们首先将符号位、整数部分和小数部分分别转换为相应的二进制数（负数 = 0b1，173 = 0b0000000010101101，0.25 = 14 = 0b010000000000000）。然后，我们将这些值打包成一个 32 位整数。最终结果是 0x8056A000。

图 3.4 对此进行了说明。
定点表示法的问题在于，它既限制了可表示的幅度范围，也限制了小数部分所能达到的精度。假设一个 32 位定点值，其中 16 位表示幅度，15 位表示小数，还有一个符号位。这种格式只能表示最大 ± 65,535 的幅度，这个范围并不算大。为了解决这个问题，我们采用浮点表示法。

3.3.1.4 浮点表示法

在浮点表示法中，小数点的位置是任意的，并借助指数来指定。浮点数分为三部分：尾数（包含小数点两边对应的数字）、指数（指示小数点在数字串中的位置）以及符号位（指示数值是正数还是负数）。有各种不同的方法

Figure 3.4. Fixed-point notation with 16-bit magnitude and 16-bit fraction.

3.3.1.3 Fixed-Point Notation

Integers are great for representing whole numbers, but to represent fractions and irrational numbers we need a different format that expresses the concept of a decimal point.

One early approach taken by computer scientists was to use *fixed-point* notation. In this notation, one arbitrarily chooses how many bits will be used to represent the whole part of the number, and the rest of the bits are used to represent the fractional part. As we move from left to right (i.e., from the most significant bit to the least significant bit), the magnitude bits represent decreasing powers of two (…, 16, 8, 4, 2, 1), while the fractional bits represent decreasing *inverse* powers of two ($\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$). For example, to store the number -173.25 in 32-bit fixed-point notation with one sign bit, 16 bits for the magnitude and 15 bits for the fraction, we first convert the sign, the whole part and the fractional part into their binary equivalents individually (negative = 0b1, 173 = 0b0000000010101101 and 0.25 = $\frac{1}{4}$ = 0b010000000000000). Then we pack those values together into a 32-bit integer. The final result is 0x8056A000. This is illustrated in Figure 3.4.

The problem with fixed-point notation is that it constrains both the range of magnitudes that can be represented and the amount of precision we can achieve in the fractional part. Consider a 32-bit fixed-point value with 16 bits for the magnitude, 15 bits for the fraction and a sign bit. This format can only represent magnitudes up to ±65,535, which isn't particularly large. To overcome this problem, we employ a *floating-point* representation.

3.3.1.4 Floating-Point Notation

In floating-point notation, the position of the decimal place is arbitrary and is specified with the help of an exponent. A floating-point number is broken into three parts: the *mantissa*, which contains the relevant digits of the number on both sides of the decimal point, the *exponent*, which indicates where in that string of digits the decimal point lies, and a *sign bit*, which of course indicates whether the value is positive or negative. There are all sorts of different ways

图 3.5。IEEE-754 32 位浮点格式。

在内存中如何布局这三个部分，但最常见的标准是 IEEE-754。它规定，32 位浮点数的最高有效位表示符号位，然后是 8 位指数，最后是 23 位尾数。

由符号位 s 、指数 e 和尾数 m 表示的值 v 是
$v = s \times 2^{(e-127)} \times (1+m)$.
符号位 s 的值为 + 1 或 − 1 。指数 e 偏移 127，以便可以轻松表示负指数。尾数以隐式 1 开头，该 1 实际上并未存储在内存中，其余位被解释为 2 的逆幂。因此，表示的值实际上是 1 + m ，其中 m 是存储在尾数中的小数值。

例如，图 3-5 所示的位模式表示值 0.15625，因为 s = 0（表示正数），e = 0b01 111100 = 124，m = 0b0100... = 0 × 2 − 1 + 1 × 2 − 2 = 1 4。因此，

$$\begin{aligned} v &= s \times 2^{(e-127)} \times (1+m) \\ &= (+1) \times 2^{(124-127)} \times (1+\tfrac{1}{4}) \\ &= 2^{-3} \times \tfrac{5}{4} \\ &= \tfrac{1}{8} \times \tfrac{5}{4} \\ &= 0.125 \times 1.25 = 0.15625. \end{aligned}$$

数量级与精度之间的权衡

浮点数的精度随着幅度的减小而增加，反之亦然。这是因为尾数的位数是固定的，并且这些位必须在整数部分和小数部分之间共享。如果大量的位用于表示较大的幅度，那么只有一小部分位可用于提供小数精度。在物理学中，术语“有效数字”通常用于描述这一概念（http://en.wikipedia.org/wiki/Significant_digits）。

为了理解幅度和精度之间的权衡，让我们看一下最大可能的浮点值 FLT_MAX ≈ 3.403 × 10 38 ，其

Figure 3.5. IEEE-754 32-bit floating-point format.

to lay out these three components in memory, but the most common standard is IEEE-754. It states that a 32-bit floating-point number will be represented with the sign in the most significant bit, followed by 8 bits of exponent and finally 23 bits of mantissa.

The value v represented by a sign bit s, an exponent e and a mantissa m is $v = s \times 2^{(e-127)} \times (1+m)$.

The sign bit s has the value $+1$ or -1. The exponent e is biased by 127 so that negative exponents can be easily represented. The mantissa begins with an implicit 1 that is not actually stored in memory, and the rest of the bits are interpreted as inverse powers of two. Hence the value represented is really $1+m$, where m is the fractional value stored in the mantissa.

For example, the bit pattern shown in Figure 3.5 represents the value 0.15625, because $s = 0$ (indicating a positive number), $e = 0b01111100 = 124$ and $m = 0b0100\ldots = 0 \times 2^{-1} + 1 \times 2^{-2} = \frac{1}{4}$. Therefore,

$$\begin{aligned} v &= s \times 2^{(e-127)} \times (1+m) \\ &= (+1) \times 2^{(124-127)} \times (1+\tfrac{1}{4}) \\ &= 2^{-3} \times \tfrac{5}{4} \\ &= \tfrac{1}{8} \times \tfrac{5}{4} \\ &= 0.125 \times 1.25 = 0.15625. \end{aligned}$$

The Trade-Off between Magnitude and Precision

The *precision* of a floating-point number increases as the *magnitude* decreases, and vice versa. This is because there are a fixed number of bits in the mantissa, and these bits must be shared between the whole part and the fractional part of the number. If a large percentage of the bits are spent representing a large magnitude, then a small percentage of bits are available to provide fractional precision. In physics the term *significant digits* is typically used to describe this concept (http://en.wikipedia.org/wiki/Significant_digits).

To understand the trade-off between magnitude and precision, let's look at the largest possible floating-point value, `FLT_MAX` $\approx 3.403 \times 10^{38}$, whose

以 32 位 IEEE 浮点格式表示为 0x7F7FFFFF。让我们分解一下：

- 我们可以用 23 位尾数表示的最大绝对值是十六进制的 0x00FFFFFF，或 24 个连续的二进制一 - 即尾数中的 23 个一，加上隐含的前导一。
- 指数 255 在 IEEE-754 格式中具有特殊含义——它用于表示非数字 (NaN) 和无穷大之类的值——因此不能用于常规数字。因此，最大八位指数实际上是 254，减去 127 的隐式偏差后，转换为 127。

因此 FLT_MAX 为 0x00FFFFFF × 2 127 = 0xFFFFFF00000000000000000000000000 。换句话说，我们的 24 个二进制 1 上移了 127 位，在尾数的最低有效数字后面留下了 127 − 23 = 104 个二进制零（或 104/4 = 26 个十六进制零）。这些尾随零并不对应于我们的 32 位浮点值中的任何实际位 - 它们只是由于指数而凭空出现的。如果我们从 FLT_MAX 中减去一个小数（“小”是指任何由少于 26 个十六进制数字组成的数字），结果仍然是 FLT_MAX ，因为这 26 个最低有效十六进制数字实际上并不存在！

对于幅值远小于 1 的浮点值，则会出现相反的效果。在这种情况下，指数很大但为负，有效数字会向相反方向移动。我们牺牲了表示大幅值的能力，换取了高精度。总而言之，浮点数中的有效数字（或真正有效位）的数量始终相同，并且可以使用指数将这些有效数字移动到更高或更低的幅值范围。

低于正常值

另一个需要注意的微妙之处是，零和我们能用浮点数表示的最小非零值之间存在有限的差距（正如迄今为止所描述的）。我们能表示的最小非零量级是 FLT_MIN = 2 − 126 ≈ 1.175 × 10 − 38 ，它的二进制表示为

0x00800000（即指数为 0x01，即减去偏差后为 − 126，尾数除隐式前导 1 外全为零）。下一个最小有效值为零，因此 -FLT_MIN 和 +FLT_MIN 之间存在有限的差距。这强调了在使用浮点表示时实数轴是量化的。（注意，C++ 标准库将 FLT_MIN 公开为更冗长的 std::numeric_limits

representation in 32-bit IEEE floating-point format is 0x7F7FFFFF. Let's break this down:

- The largest absolute value that we can represent with a 23-bit mantissa is 0x00FFFFFF in hexadecimal, or 24 consecutive binary ones—that's 23 ones in the mantissa, plus the implicit leading one.
- An exponent of 255 has a special meaning in the IEEE-754 format—it is used for values like not-a-number (NaN) and infinity—so it cannot be used for regular numbers. Hence the maximum eight-bit exponent is actually 254, which translates into 127 after subtracting the implicit bias of 127.

So `FLT_MAX` is $0x00FFFFFF \times 2^{127} = 0xFFFFFF00000000000000000000000000$. In other words, our 24 binary ones were shifted up by 127 bit positions, leaving $127 - 23 = 104$ binary zeros (or $104/4 = 26$ hexadecimal zeros) after the least significant digit of the mantissa. Those trailing zeros don't correspond to any actual bits in our 32-bit floating-point value—they just appear out of thin air because of the exponent. If we were to subtract a small number (where "small" means any number composed of fewer than 26 hexadecimal digits) from `FLT_MAX`, the result would still be `FLT_MAX`, because those 26 least significant hexadecimal digits don't really exist!

The opposite effect occurs for floating-point values whose magnitudes are much less than one. In this case, the exponent is large but negative, and the significant digits are shifted in the opposite direction. We trade the ability to represent large magnitudes for high precision. In summary, we always have the same number of *significant digits* (or really *significant bits*) in our floating-point numbers, and the exponent can be used to shift those significant bits into higher or lower ranges of magnitude.

Subnormal Values

Another subtlety to notice is that there is a finite gap between zero and the smallest nonzero value we can represent with floating-point notation (as it has been described thus far). The smallest nonzero magnitude we can represent is `FLT_MIN` $= 2^{-126} \approx 1.175 \times 10^{-38}$, which has a binary representation of 0x00800000 (i.e., the exponent is 0x01, or -126 after subtracting the bias, and the mantissa is all zeros except for the implicit leading one). The next smallest valid value is zero, so there is a finite gap between the values `-FLT_MIN` and `+FLT_MIN`. This underscores the fact that the real number line is *quantized* when using a floating-point representation. (Note that the C++ standard library exposes `FLT_MIN` as the rather more verbose `std::numeric_limits`

<float>::min() 。本书为简洁起见，仍使用 FLT_MIN。）零附近的空隙可以通过对浮点表示进行扩展来填补，这种扩展称为非规格化值，也称为次规格化值。使用此扩展，任何带有偏置指数 0 的浮点值都将被解释为次规格化数。指数将被视为 1 而不是 0，并且通常位于尾数位前面的隐式前导 1 将更改为 0。这具有用均匀分布的次规格化值的线性序列填充 -FLT_MIN 和 +FLT_MIN 之间的空隙的效果。最接近零的正次规格化浮点数由常量 FLT_TRUE_MIN 表示。

使用次正规值的好处是，它在接近零时能提供更高的精度。例如，它确保以下两个表达式等价，即使 a 和 b 的值非常接近 FLT_MIN：

```
if (a == b) { ... }
if (a - b == 0.0f) { ... }
```

如果没有次正规值，即使 a != b ，表达式 ab 也可能计算为零。

机器 Epsilon

对于特定的浮点表示，机器 epsilon 定义为满足公式 1 + ε / = 1 的最小浮点值 ε。对于精度为 23 位的 IEEE-754 浮点数，ε 的值为 2 − 23 ，约为 1.192 × 10 − 7 。ε 的最高有效位刚好落在值 1.0 的有效位范围内，因此将任何小于 ε 的值添加到 1.0 都没有效果。换句话说，当我们尝试将和放入只有 23 位的尾数中时，任何因添加小于 ε 的值而贡献的新位都将被“截断”。

末位单位 (ULP)

考虑两个浮点数，除了尾数中最低有效位的值外，其余各方面均相同。这两个值在最后一个位置上相差一个单位（1 ULP）。1 ULP 的实际值根据指数而变化。例如，浮点值 1.0f 的无偏指数为零，尾数中所有位均为零（隐式前导 1 除外）。在此指数下，1 ULP 等于机器 epsilon（2 − 23）。如果我们将指数更改为 1，得到值 2.0f，则 1 ULP 的值等于机器 epsilon 的两倍。如果指数为 2，得到值 4.0f，则 1 ULP 的值等于四

`<float>::min()`. We'll stick to `FLT_MIN` in this book for brevity.)

The gap around zero can be filled by employing an extension to the floating-point representation known as *denormalized values*, also known as *subnormal values*. With this extension, any floating-point value with a biased exponent of 0 is interpreted as a subnormal number. The exponent is treated as if it had been a 1 instead of a 0, and the implicit leading 1 that normally sits in front of the bits of the mantissa is changed to a 0. This has the effect of filling the gap between `-FLT_MIN` and `+FLT_MIN` with a linear sequence of evenly-spaced subnormal values. The positive subnormal `float` that is closest to zero is represented by the constant `FLT_TRUE_MIN`.

The benefit of using subnormal values is that it provides greater precision near zero. For example, it ensures that the following two expressions are equivalent, even for values of `a` and `b` that are very close to `FLT_MIN`:

```
if (a == b) { ... }
if (a - b == 0.0f) { ... }
```

Without subnormal values, the expression `a - b` could evaluate to zero even when `a != b`.

Machine Epsilon

For a particular floating-point representation, the *machine epsilon* is defined to be the smallest floating-point value ε that satisfies the equation, $1 + \varepsilon \neq 1$. For an IEEE-754 floating-point number, with its 23 bits of precision, the value of ε is 2^{-23}, which is approximately 1.192×10^{-7}. The most significant digit of ε falls *just inside* the range of significant digits in the value 1.0, so adding any value smaller than ε to 1.0 has no effect. In other words, any new bits contributed adding a value smaller than ε will get "chopped off" when we try to fit the sum into a mantissa with only 23 bits.

Units in the Last Place (ULP)

Consider two floating-point numbers which are identical in all respects except for the value of the least-significant bit in their mantissas. These two values are said to differ by one *unit in the last place* (1 ULP). The actual value of 1 ULP changes depending on the exponent. For example, the floating-point value `1.0f` has an unbiased exponent of zero, and a mantissa in which all bits are zero (except for the implicit leading 1). At this exponent, 1 ULP is equal to the machine epsilon (2^{-23}). If we change the exponent to a 1, yielding the value `2.0f`, the value of 1 ULP becomes equal to two times the machine epsilon. And if the exponent is 2, yielding the value `4.0f`, the value of 1 ULP is four

乘以机器 epsilon。通常，如果浮点值的无偏指数为 x ，则 1 ULP = 2 x · ε 。

最后提到的单位概念说明了浮点数的精度取决于其指数，这对于量化任何浮点计算中固有的误差非常有用。它还可用于查找相对于已知值的下一个最大可表示值的浮点值，或者反过来，查找相对于该值的下一个最小可表示值的浮点值。这反过来又有助于将大于或等于比较转换为大于比较。从数学上讲，条件 a ⩾ b 等同于条件 a + 1 ULP > b 。我们在顽皮狗引擎中使用了这个小技巧来简化角色对话系统中的一些逻辑。在这个系统中，可以使用简单的比较来选择角色要说的不同台词。我们仅支持大于和小于检查，而不是支持所有可能的比较运算符，并且通过在被比较的值上加或减 1 ULP 来处理大于或等于和小于或等于的情况。

浮点精度对软件的影响

有限精度和机器精度的概念对游戏软件有着实际的影响。例如，假设我们使用浮点变量来跟踪以秒为单位的绝对游戏时间。在时钟变量的幅度变得过大，以至于在其上添加 1/30 秒不再改变其值之前，我们的游戏可以运行多长时间？答案是 12.14 天或 2^20 秒。这比大多数游戏的运行时间要长，因此我们可能可以在游戏中使用以秒为单位的 32 位浮点时钟。但显然，了解浮点格式的局限性很重要，这样我们才能预测潜在问题并在必要时采取措施避免它们。

IEEE 浮点位技巧

请参阅[9，第 2.1 节]，了解一些非常有用的 IEEE 浮点“位技巧”，这些技巧可以使某些浮点计算速度快如闪电。

3.3.2 原始数据类型

C 和 C++ 提供了许多原始数据类型。C 和 C++ 标准提供了关于这些数据类型的相对大小和符号的指导原则，但每个编译器都可以自由地对这些类型进行略微不同的定义，以便在目标硬件上提供最佳性能。

times the machine epsilon. In general, if a floating-point value's unbiased exponent is x, then 1 ULP $= 2^x \cdot \varepsilon$.

The concept of units in the last place illustrates the idea that the precision of a floating-point number depends on its exponent, and is useful for quantifying the error inherent in any floating-point calculation. It can also be useful for finding the floating-point value that is the next largest representable value relative to a known value, or conversely the next smallest representable value relative to that value. This in turn can be useful for converting a greater-than-or-equal comparison into a greater-than comparison. Mathematically, the condition $a \geq b$ is equivalent to the condition $a + 1$ ULP $> b$. We use this little "trick" in the Naughty Dog engine to simplify some logic in our character dialog system. In this system, simple comparisons can be used to select different lines of dialog for the characters to say. Rather than supporting all possible comparison operators, we only support greater-than and less-than checks, and we handle greater-than-or-equal-to and less-than-or-equal-to by adding or subtracting 1 ULP to or from the value being compared.

Impact of Floating-Point Precision on Software

The concepts of limited precision and the machine epsilon have real impacts on game software. For example, let's say we use a floating-point variable to track absolute game time in seconds. How long can we run our game before the magnitude of our clock variable gets so large that adding $1/30^{th}$ of a second to it no longer changes its value? The answer is 12.14 days or 2^{20} seconds. That's longer than most games will be left running, so we can probably get away with using a 32-bit floating-point clock measured in seconds in a game. But clearly it's important to understand the limitations of the floating-point format so that we can predict potential problems and take steps to avoid them when necessary.

IEEE Floating-Point Bit Tricks

See [9, Section 2.1] for a few really useful IEEE floating-point "bit tricks" that can make certain floating-point calculations lightning fast.

3.3.2 Primitive Data Types

C and C++ provide a number of primitive data types. The C and C++ standards provide guidelines on the relative sizes and signedness of these data types, but each compiler is free to define the types slightly differently in order to provide maximum performance on the target hardware.

• char 。char 通常为 8 位，通常足够容纳一个 ASCII 或 UTF-8 字符（参见 6.4.4.1 节）。有些编译器将 char 定义为有符号的，而有些编译器默认使用无符号的。

• int 、 short 、 long 。int 用来保存有符号整数值，该值对于目标平台来说是最有效的大小；在 32 位 CPU 架构（如 Pentium 4 或 Xeon）上，它通常定义为 32 位宽，在 64 位架构（如 Intel Core i7）上，它通常定义为 64 位宽，不过 int 的大小还取决于其他因素，如编译器选项和目标操作系统。short 比 int 小，在许多机器上是 16 位。long 与 int 一样大或更大，可能是 32 位或 64 位宽，甚至更宽，同样取决于 CPU 架构、编译器选项和目标操作系统。

• 浮点数。在大多数现代编译器中，浮点数是 32 位 IEEE-754 浮点值。

• double . double 是双精度（即 64 位）IEEE-754 浮点值。

• bool 。bool 表示真/假值。bool 的大小在不同的编译器和硬件架构中差异很大。它从不以单个位实现，但有些编译器将其定义为 8 位，而其他编译器则使用完整的 32 位。

便携尺寸类型

C 和 C++ 内置的原始数据类型被设计为可移植的，因此不具有特定性。然而，在许多软件工程工作中，包括游戏引擎编程，准确了解特定变量的宽度通常很重要。

在 C++11 之前，程序员必须依赖编译器提供的不可移植的大小类型。例如，Visual Studio C/C++ 编译器定义了以下扩展关键字，用于声明具有明确位数的变量：__int8、__int16、__int32 和 __int64。大多数其他编译器都有自己的“大小”数据类型，语义相似，但语法略有不同。

由于编译器之间的这些差异，大多数游戏引擎通过定义自己的自定义大小类型来实现源代码的可移植性。
例如，在顽皮狗我们使用以下尺寸类型：

• F32 是 32 位 IEEE-754 浮点值。

• U8 、 I8 、 U16 、 I16 、 U32 、 I32 、 U64 和 I64 分别是无符号和有符号的 8 位、16 位、32 位和 64 位整数。

- `char`. A `char` is usually 8 bits and is generally large enough to hold an ASCII or UTF-8 character (see Section 6.4.4.1). Some compilers define `char` to be signed, while others use unsigned `char`s by default.
- `int`, `short`, `long`. An `int` is supposed to hold a signed integer value that is the most efficient size for the target platform; it is usually defined to be 32 bits wide on a 32-bit CPU architecture, such as Pentium 4 or Xeon, and 64 bits wide on a 64-bit architecture, such as Intel Core i7, although the size of an `int` is also dependent upon other factors such as compiler options and the target operating system. A `short` is intended to be smaller than an `int` and is 16 bits on many machines. A `long` is as large as or larger than an `int` and may be 32 or 64 bits wide, or even wider, again depending on CPU architecture, compiler options and the target OS.
- `float`. On most modern compilers, a `float` is a 32-bit IEEE-754 floating-point value.
- `double`. A `double` is a double-precision (i.e., 64-bit) IEEE-754 floating-point value.
- `bool`. A `bool` is a true/false value. The size of a `bool` varies widely across different compilers and hardware architectures. It is never implemented as a single bit, but some compilers define it to be 8 bits while others use a full 32 bits.

Portable Sized Types

The built-in primitive data types in C and C++ were designed to be portable and therefore nonspecific. However, in many software engineering endeavors, including game engine programming, it is often important to know exactly how wide a particular variable is.

Before C++11, programmers had to rely on non-portable sized types provided by their compiler. For example, the Visual Studio C/C++ compiler defined the following extended keywords for declaring variables that are an explicit number of bits wide: `__int8`, `__int16`, `__int32` and `__int64`. Most other compilers have their own “sized” data types, with similar semantics but slightly different syntax.

Because of these differences between compilers, most game engines achieved source code portability by defining their own custom sized types. For example, at Naughty Dog we use the following sized types:

- `F32` is a 32-bit IEEE-754 floating-point value.
- `U8`, `I8`, `U16`, `I16`, `U32`, `I32`, `U64` and `I64` are unsigned and signed 8-, 16-, 32- and 64-bit integers, respectively.

• U32F 和 I32F 分别是“快速”的无符号和有符号 32 位值。这两种数据类型都包含一个至少 32 位宽的值，但如果能在目标 CPU 上实现更快的代码执行速度，也可以更宽。

<cstdint>

C++11 标准库引入了一组标准化大小的整数类型。它们在 <cstdint> 头文件中声明，包括有符号类型 std::int8_t 、 std::int16_t 、 std::int32_t 和 std::int64_t ，以及无符号类型 std::uint8_t 、 std::uint16_t 、 std::uint32_t 和 std::uint64_t ，以及一些“快速”变体（例如我们在顽皮狗中定义的 I32F 和 U32F 类型）。这些类型使程序员无需为了实现可移植性而“包装”编译器特定的类型。有关这些大小类型的完整列表，请参阅 http://en.cppreference。

com/w/cpp/types/integer。

OGRE 的原始数据类型

OGRE 定义了许多自己的大小类型。例如 Ogre::uint8 、Ogre::
uint16 和 Ogre::uint32 是基本的无符号整数类型。

Ogre::Real 定义了一个实数浮点值。它通常被定义为 32 位宽（相当于 float 类型），但可以通过将预处理器宏 OGRE_DOUBLE _PRECISION 定义为 1 来全局重新定义为 64 位宽（相当于 double 类型）。这种改变 Ogre::Real 含义的能力通常只在游戏对双精度运算有特殊要求时才会使用，但这种情况很少见。图形芯片（GPU）总是使用 32 位或 16 位浮点数进行运算，CPU/FPU 在单精度运算时通常速度更快，而 SIMD 矢量指令操作的是 128 位寄存器，每个寄存器包含四个 32 位浮点数。因此，大多数游戏倾向于使用单精度浮点运算。

数据类型 Ogre::uchar 、 Ogre::ushort 、 Ogre::uint 和 Ogre ::ulong 分别只是 C/C++ 的 unsigned char 、 unsigned short 和 unsigned long 的简写形式。因此，它们的实用性与 C/C++ 原生对应类型并无二致。

Ogre::Radian 和 Ogre::Degree 类型尤其有趣。这两个类是对简单 Ogre::Real 值的包装。它们的主要作用是允许记录硬编码文字常量的角度单位，并提供两种单位制之间的自动转换。此外，Ogre::Angle 类型表示当前“默认”角度单位的角度。程序员可以在 OGRE 应用程序首次启动时定义默认角度单位是弧度还是度。

- `U32F` and `I32F` are “fast” unsigned and signed 32-bit values, respectively. Each of these data types contains a value that is at least 32 bits wide, but may be wider if that would result in faster code on the target CPU.

<cstdint>

The C++11 standard library introduces a set of standardized sized integer types. They are declared in the `<cstdint>` header, and they include the signed types `std::int8_t`, `std::int16_t`, `std::int32_t` and `std::int64_t` and the unsigned types `std::uint8_t`, `std::uint16_t`, `std::uint32_t` and `std::uint64_t`, along with “fast” variants (like the `I32F` and `U32F` types we defined at Naughty Dog). These types free the programmer from having to “wrap” compiler-specific types in order to achieve portability. For a complete list of these sized types, see http://en.cppreference.com/w/cpp/types/integer.

OGRE's Primitive Data Types

OGRE defines a number of sized types of its own. `Ogre::uint8`, `Ogre::uint16` and `Ogre::uint32` are the basic unsigned sized integral types.

`Ogre::Real` defines a real floating-point value. It is usually defined to be 32 bits wide (equivalent to a `float`), but it can be redefined globally to be 64 bits wide (like a `double`) by defining the preprocessor macro `OGRE_DOUBLE_PRECISION` to `1`. This ability to change the meaning of `Ogre::Real` is generally only used if one's game has a particular requirement for double-precision math, which is rare. Graphics chips (GPUs) always perform their math with 32-bit or 16-bit floats, the CPU/FPU is also usually faster when working in single-precision, and SIMD vector instructions operate on 128-bit registers that contain four 32-bit floats each. Hence, most games tend to stick to single-precision floating-point math.

The data types `Ogre::uchar`, `Ogre::ushort`, `Ogre::uint` and `Ogre::ulong` are just shorthand notations for C/C++'s `unsigned char`, `unsigned short` and `unsigned long`, respectively. As such, they are no more or less useful than their native C/C++ counterparts.

The types `Ogre::Radian` and `Ogre::Degree` are particularly interesting. These classes are wrappers around a simple `Ogre::Real` value. The primary role of these types is to permit the angular units of hard-coded literal constants to be documented and to provide automatic conversion between the two unit systems. In addition, the type `Ogre::Angle` represents an angle in the current “default” angle unit. The programmer can define whether the default will be radians or degrees when the OGRE application first starts up.

或许令人惊讶的是，OGRE 并没有提供其他游戏引擎中常见的一些大小可变的原始数据类型。例如，它没有定义有符号的 8 位、16 位或 64 位整数类型。如果你正在基于 OGRE 编写游戏引擎，你可能会发现自己需要在某些时候手动定义这些类型。

3.3.2.1 多字节值和字节顺序

宽度大于八位（一个字节）的值称为多字节量。它们在任何使用 16 位或更宽的整数和浮点值的软件项目中都很常见。例如，整数值 4660 = 0x1234 由两个字节 0x12 和 0x34 表示。我们称之为

0x12 表示最高有效字节，0x34 表示最低有效字节。对于 32 位值，例如 0xABCD1234，最高有效字节为 0xAB，最低有效字节为 0x34。相同的概念也适用于 64 位整数以及 32 位和 64 位浮点值。

多字节整数可以通过两种方式存储到内存中，不同的微处理器在存储方法的选择上可能有所不同（见图 3.6）。

- 小端字节序。如果微处理器将多字节值的最低有效字节存储在比最高有效字节更低的内存地址中，我们就说该处理器是小端字节序的。在小端字节序的机器上，数字 0xABCD1234 将使用连续的字节 0x34、0x12、0xCD、0xAB 存储在内存中。

- 大端字节法。如果微处理器将多字节值的最高有效字节存储在比最低有效字节更低的内存地址中，

```
U32 value = 0xABCD1234;
U8* pBytes = (U8*)&value;
```

图 3.6。值 0xABCD1234 的大端和小端表示。

Perhaps surprisingly, OGRE does not provide a number of sized primitive data types that are commonplace in other game engines. For example, it defines no signed 8-, 16- or 64-bit integral types. If you are writing a game engine on top of OGRE, you will probably find yourself defining these types manually at some point.

3.3.2.1 Multibyte Values and Endianness

Values that are larger than eight bits (one byte) wide are called *multibyte quantities*. They're commonplace on any software project that makes use of integers and floating-point values that are 16 bits or wider. For example, the integer value 4660 = 0x1234 is represented by the two bytes 0x12 and 0x34. We call 0x12 the most significant byte and 0x34 the least significant byte. In a 32-bit value, such as 0xABCD1234, the most-significant byte is 0xAB and the least-significant is 0x34. The same concepts apply to 64-bit integers and to 32- and 64-bit floating-point values as well.

Multibyte integers can be stored into memory in one of two ways, and different microprocessors may differ in their choice of storage method (see Figure 3.6).

- *Little-endian*. If a microprocessor stores the least significant byte of a multibyte value at a lower memory address than the most significant byte, we say that the processor is *little-endian*. On a little-endian machine, the number 0xABCD1234 would be stored in memory using the consecutive bytes 0x34, 0x12, 0xCD, 0xAB.
- *Big-endian*. If a microprocessor stores the most significant byte of a multibyte value at a lower memory address than the least significant byte,

```
U32 value = 0xABCD1234;
U8* pBytes = (U8*)&value;
```

Figure 3.6. Big- and little-endian representations of the value 0xABCD1234.

我们称处理器是大端的。在大端的机器上，数字 0xABCD1234 将使用字节 0xAB 存储在内存中，
0xCD，0x12，0x34。

大多数程序员不需要过多考虑字节序。但是，当您是游戏程序员时，字节序可能会成为您的眼中钉。这是因为游戏通常是在运行 Intel Pentium 处理器（小端）的 Windows 或 Linux 机器上开发的，但在 Wii、Xbox 360 或 PlayStation 3 等游戏机上运行——这三款游戏都使用 PowerPC 处理器的变体（可以配置为使用任一字节序，但默认情况下为大端）。现在想象一下，当您在 Intel 处理器上生成供游戏引擎使用的数据文件，然后尝试将该数据文件加载到在 Power PC 处理器上运行的引擎中时会发生什么。您写入该数据文件的任何多字节值都将以小端格式存储。但当游戏引擎读取该文件时，它希望所有数据都采用大端格式。结果会怎样？您将写入 0xABCD1234，但读取到 0x3412CDAB，这显然不是您想要的！

这个问题至少有两种解决方案。

1. 你可以将所有数据文件都写成文本，并将所有多字节数字存储为十进制或十六进制数字序列，每个数字对应一个字符（一个字节）。虽然这会降低磁盘空间的利用效率，但确实可行。

2. 您可以让您的工具在将数据写入二进制数据文件之前进行字节序交换。实际上，您可以确保数据文件使用目标微处理器（游戏主机）的字节序，即使工具运行在使用相反字节序的机器上。

整数字节序交换

从概念上来说，对整数进行字节序交换并不困难。只需从值的最高有效字节开始，将其与最低有效字节进行交换即可；重复此过程，直到到达值的中间位置。
例如，0xA7891023 将变成 0x231089A7。
唯一棘手的部分是知道要交换哪些字节。假设你正在将一个 C 结构体或 C++ 类的内容从内存写入文件。为了正确地对这些数据进行字节序交换，你需要跟踪结构体中每个数据成员的位置和大小，并根据其大小进行适当的交换。例如，结构体

we say that the processor is *big-endian*. On a big-endian machine, the number 0xABCD1234 would be stored in memory using the bytes 0xAB, 0xCD, 0x12, 0x34.

Most programmers don't need to think much about endianness. However, when you're a game programmer, endianness can become a bit of a thorn in your side. This is because games are usually *developed* on a Windows or Linux machine running an Intel Pentium processor (which is little-endian), but *run* on a console such as the Wii, Xbox 360 or PlayStation 3—all three of which utilize a variant of the PowerPC processor (which can be configured to use either endianness, but is big-endian by default). Now imagine what happens when you generate a data file for consumption by your game engine on an Intel processor and then try to load that data file into your engine running on a PowerPC processor. Any multibyte value that you wrote out into that data file will be stored in little-endian format. But when the game engine reads the file, it expects all of its data to be in big-endian format. The result? You'll write 0xABCD1234, but you'll read 0x3412CDAB, and that's clearly not what you intended!

There are at least two solutions to this problem.

1. You could write all your data files as text and store all multibyte numbers as sequences of decimal or hexadecimal digits, one character (one byte) per digit. This would be an inefficient use of disk space, but it would work.

2. You can have your tools endian-swap the data prior to writing it into a binary data file. In effect, you make sure that the data file uses the endianness of the target microprocessor (the game console), even if the tools are running on a machine that uses the opposite endianness.

Integer Endian-Swapping

Endian-swapping an integer is not conceptually difficult. You simply start at the most significant byte of the value and swap it with the least significant byte; you continue this process until you reach the halfway point in the value. For example, 0xA7891023 would become 0x231089A7.

The only tricky part is knowing *which bytes* to swap. Let's say you're writing the contents of a C `struct` or C++ `class` from memory out to a file. To properly endian-swap this data, you need to keep track of the locations and sizes of each data member in the `struct` and swap each one appropriately based on its size. For example, the structure

```
struct Example
{
   U32   m_a;
   U16   m_b;
   U32   m_c;
};
```

可能会被写入数据文件如下：

```
void writeExampleStruct(Example& ex, Stream& stream)
{
   stream.writeU32(swapU32(ex.m_a));
   stream.writeU16(swapU16(ex.m_b));
   stream.writeU32(swapU32(ex.m_c));
}
```

交换函数可能定义如下：

```
inline U16 swapU16(U16 value)
{
   return ((value & 0x00FF) << 8)
        | ((value & 0xFF00) >> 8);
}

inline U32 swapU32(U32 value)
{
   return ((value & 0x000000FF) << 24)
        | ((value & 0x0000FF00) << 8)
        | ((value & 0x00FF0000) >> 8)
        | ((value & 0xFF000000) >> 24);
}
```

您不能简单地将 Example 对象转换为字节数组，然后使用单个通用函数盲目地交换字节。我们需要知道要交换哪些数据成员以及每个成员的宽度，并且必须单独交换每个数据成员。

一些编译器提供了内置的字节序交换宏，让您无需自己编写。例如，gcc 提供了一个名为 __builtin_bswapXX() 的宏系列，用于执行 16 位、32 位和 64 位字节序交换。然而，这些特定于编译器的功能当然是不可移植的。

浮点字节序交换

正如我们所见，IEEE-754 浮点值具有详细的内部结构，包括尾数、指数和符号位。但是，你可以像整数一样交换它的字节序，因为字节就是字节。你只需要重新解释浮点数的位模式即可。


```
struct Example
{
   U32   m_a;
   U16   m_b;
   U32   m_c;
};
```

might be written out to a data file as follows:

```
void writeExampleStruct(Example& ex, Stream& stream)
{
   stream.writeU32(swapU32(ex.m_a));
   stream.writeU16(swapU16(ex.m_b));
   stream.writeU32(swapU32(ex.m_c));
}
```

and the swap functions might be defined like this:

```
inline U16 swapU16(U16 value)
{
   return ((value & 0x00FF) << 8)
        | ((value & 0xFF00) >> 8);
}

inline U32 swapU32(U32 value)
{
   return ((value & 0x000000FF) << 24)
        | ((value & 0x0000FF00) << 8)
        | ((value & 0x00FF0000) >> 8)
        | ((value & 0xFF000000) >> 24);
}
```

You cannot simply cast the `Example` object into an array of bytes and blindly swap the bytes using a single general-purpose function. We need to know both *which data members* to swap and *how wide* each member is, and each data member must be swapped individually.

Some compilers provide built-in endian-swapping macros, freeing you from having to write your own. For example, gcc offers a family of macros named `__builtin_bswapXX()` for performing 16-, 32- and 64-bit endian swaps. However, such compiler-specific facilities are of course non-portable.

Floating-Point Endian-Swapping

As we've seen, an IEEE-754 floating-point value has a detailed internal structure involving some bits for the mantissa, some bits for the exponent and a sign bit. However, you can endian-swap it just as if it were an integer, because bytes are bytes. You merely reinterpret the bit pattern of your `float`

就像它是 std::int32_t 一样，执行字节序交换操作，然后再次将结果重新解释为浮点数。

您可以使用 C++ 的 reinterpret_cast 运算符将浮点数重新解释为整数，方法是对指向浮点数的指针执行 reinterpret_cast 操作，然后取消引用类型转换后的指针；这被称为类型双关。但是，当启用严格别名时，类型双关可能会导致优化错误。（有关此问题的详细描述，请参阅 http://www.cocoawithlove.com/2008/04/using-pointers-to-recast-in-c-is-bad.html。）一种保证可移植的替代方法是使用联合，如下所示：

```
union U32F32
{
   U32    m_asU32;
   F32    m_asF32;
};

inline F32 swapF32(F32 value)
{
   U32F32 u;
   u.m_asF32 = value;

   // endian-swap as integer
   u.m_asU32 = swapU32(u.m_asU32);

   return u.m_asF32;
}
```

3.3.3 千字节与千比字节

您可能使用过公制 (SI) 单位，例如千字节 (kB) 和兆字节 (MB) 来描述内存数量。然而，使用这些单位来描述以 2 的幂为单位的内存数量并不严格正确。当计算机程序员提到“千字节”时，通常指的是 1024 字节。但 SI 单位将前缀“千”定义为 10^3 或 1000，而不是 1024。

为了解决这种歧义，国际电工委员会 (IEC) 于 1998 年建立了一套新的类似国际单位制 (SI) 的前缀，用于计算机科学。这些前缀以 2 的幂而不是 10 的幂来定义，以便计算机工程师能够精确而方便地指定 2 的幂的数量。在这个新系统中，我们用千比字节 (1024 字节，缩写为 KiB) 代替千字节 (1000 字节)。我们用兆比字节 (1,000,000 字节) 代替兆字节 (1,000,000 字节)。表 3.1 总结了以下单位的大小、前缀和名称：

as if it were a `std::int32_t`, perform the endian swapping operation, and then reinterpret the result as a `float` again.

You can reinterpret floats as integers by using C++'s `reinterpret_cast` operator on a pointer to the float, and then dereferencing the type-cast pointer; this is known as *type punning*. But punning can lead to optimization bugs when strict aliasing is enabled. (See http://www.cocoawithlove.com/2008/04/using-pointers-to-recast-in-c-is-bad.html for an excellent description of this problem.) One alternative that's guaranteed to be portable is to use a union, as follows:

```
union U32F32
{
   U32    m_asU32;
   F32    m_asF32;
};

inline F32 swapF32(F32 value)
{
   U32F32 u;
   u.m_asF32 = value;

   // endian-swap as integer
   u.m_asU32 = swapU32(u.m_asU32);

   return u.m_asF32;
}
```

3.3.3 Kilobytes versus Kibibytes

You've probably used metric (SI) units like kilobytes (kB) and megabytes (MB) to describe quantities of memory. However, the use of these units to describe quantities of memory that are measured in powers of two isn't strictly correct. When a computer programmer speaks of a "kilobyte," she or he usually means 1024 bytes. But SI units define the prefix "kilo" to mean 10^3 or 1000, not 1024.

To resolve this ambiguity, the International Electrotechnical Commission (IEC) in 1998 established a new set of SI-like prefixes for use in computer science. These prefixes are defined in terms of powers of two rather than powers of ten, so that computer engineers can precisely and conveniently specify quantities that are powers of two. In this new system, instead of kilobyte (1000 bytes), we say kibibyte (1024 bytes, abbreviated KiB). And instead of megabyte (1,000,000 bytes), we say mebibyte (1024×1024 = 1,048,576 bytes, abbreviated MiB). Table 3.1 summarizes the sizes, prefixes and names of the

公制（SI）			IEC		
价值	单元	姓名	价值	单元	姓名
1000	kB	千字节	1024	基布	千字节
1000^2	MB	兆字节	1024^2	信息局	兆字节
1000^3	GB	技嘉	1024^3	吉布	技嘉
1000^4	TB	太字节	1024^4	硼化钛	太字节
1000^5	PB	PB 级	1024^5	派布	百亿字节
1000^6	EB	艾字节	1024^6	爱尔兰	艾比字节
1000^7	ZB	泽字节	1024^7	齐布	泽比字节
1000^8	YB	尧字节	1024^8	伊布	优比字节

表 3.1. 用于描述字节数量的公制 (SI) 单位和 IEC 单位的比较。

SI 和 IEC 系统中最常用的字节数量单位。本书将通篇使用 IEC 单位。

3.3.4 声明、定义和联系

3.3.4.1 重新审视翻译单元

正如我们在第二章中看到的，C 或 C++ 程序由翻译单元 (translation unit) 组成。编译器每次翻译一个 .cpp 文件，并为每个文件生成一个输出文件，称为目标文件（.o 或 .obj）。.cpp 文件是编译器操作的最小翻译单元，因此得名“翻译单元”。目标文件不仅包含 .cpp 文件中定义的所有函数的编译机器码，还包含其所有全局变量和静态变量。此外，目标文件可能包含对其他 .cpp 文件中定义的函数和全局变量的未解析引用。

编译器每次只对一个翻译单元进行操作，因此每当它遇到对外部全局变量或函数的引用时，它必须“凭信心”地假设该实体确实存在，如图 3.7 所示。链接器的工作是将所有目标文件组合成最终的可执行映像。在此过程中，链接器会读取所有目标文件，并尝试解析它们之间所有未解析的交叉引用。如果成功，则会生成一个包含所有函数、全局变量和静态变量的可执行映像，所有跨翻译单元的引用都已正确解析。如图 3.8 所示。

链接器的主要工作是解析外部引用，并且在此

Metric (SI)			IEC		
Value	Unit	Name	Value	Unit	Name
1000	kB	kilobyte	1024	KiB	kibibyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte
1000^5	PB	petabyte	1024^5	PiB	pebibyte
1000^6	EB	exabyte	1024^6	EiB	exbibyte
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte
1000^8	YB	yottabyte	1024^8	YiB	yobibyte

Table 3.1. Comparison of metric (SI) units and IEC units for describing quantities of bytes.

most commonly used byte quantity units in both the SI and IEC systems. We'll use IEC units throughout this book.

3.3.4 Declarations, Definitions and Linkage

3.3.4.1 Translation Units Revisited

As we saw in Chapter 2, a C or C++ program is comprised of *translation units*. The compiler translates one .cpp file at a time, and for each one it generates an output file called an object file (.o or .obj). A .cpp file is the smallest unit of translation operated on by the compiler; hence, the name "translation unit." An object file contains not only the compiled machine code for all of the functions defined in the .cpp file, but also all of its global and static variables. In addition, an object file may contain *unresolved references* to functions and global variables defined in *other* .cpp files.

The compiler only operates on one translation unit at a time, so whenever it encounters a reference to an external global variable or function, it must "go on faith" and assume that the entity in question really exists, as shown in Figure 3.7. It is the linker's job to combine all of the object files into a final executable image. In doing so, the linker reads all of the object files and attempts to resolve all of the unresolved cross-references between them. If it is successful, an executable image is generated containing all of the functions, global variables and static variables, with all cross-translation-unit references properly resolved. This is depicted in Figure 3.8.

The linker's primary job is to resolve external references, and in this

图 3.7. 两个翻译单元中未解析的外部引用。

图 3.8. 成功链接后完全解析的外部引用。

图 3.9. 两个最常见的链接器错误。

Figure 3.7. Unresolved external references in two translation units.

Figure 3.8. Fully resolved external references after successful linking.

Figure 3.9. The two most common linker errors.

容量它只能产生两种错误：

1. 可能找不到外部引用的目标，在这种情况下，链接器会生成“未解析符号”错误。

2. 链接器可能会发现多个具有相同名称的变量或函数，在这种情况下，它会生成“多重定义符号”错误。

这两种情况如图3.9所示。

3.3.4.2 声明与定义

在 C 和 C++ 语言中，变量和函数必须先声明和定义才能使用。理解 C 和 C++ 中声明和定义之间的区别非常重要。

- 声明是对数据对象或函数的描述。它向编译器提供实体的名称及其数据类型或函数签名（即返回类型和参数类型）。

- 定义则描述了程序中一块独特的内存区域。这块内存可能包含一个变量、一个结构体或类的实例，或者一个函数的机器码。

换句话说，声明是对实体的引用，而定义则是实体本身。定义始终是声明，但反之则不然——在 C 和 C++ 中，可以编写一个不是定义的纯声明。

函数的定义方式是，在签名之后立即写入函数主体，并用花括号括起来：

foo.cpp

```
// definition of the max() function
int max(int a, int b)
{
   return (a > b) ? a : b;
}

// definition of the min() function
int min(int a, int b)
{
   return (a <= b) ? a : b;
}
```


capacity it can generate only two kinds of errors:

1. The target of an `extern` reference might not be found, in which case the linker generates an “unresolved symbol” error.

2. The linker might find more than one variable or function with the same name, in which case it generates a “multiply defined symbol” error.

These two situations are shown in Figure 3.9.

3.3.4.2 Declaration versus Definition

In the C and C++ languages, variables and functions must be *declared* and *defined* before they can be used. It is important to understand the difference between a declaration and a definition in C and C++.

- A *declaration* is a description of a data object or function. It provides the compiler with the *name* of the entity and its *data type* or *function signature* (i.e., return type and argument type(s)).
- A *definition*, on the other hand, describes a unique region of memory in the program. This memory might contain a variable, an instance of a struct or class or the machine code of a function.

In other words, a declaration is a *reference to an entity*, while a definition is the *entity itself*. A definition is always a declaration, but the reverse is not always the case—it is possible to write a pure declaration in C and C++ that is not a definition.

Functions are *defined* by writing the body of the function immediately after the signature, enclosed in curly braces:

foo.cpp

```
// definition of the max() function
int max(int a, int b)
{
   return (a > b) ? a : b;
}

// definition of the min() function
int min(int a, int b)
{
   return (a <= b) ? a : b;
}
```

可以为函数提供纯声明，以便它可以在其他翻译单元（或稍后在同一个翻译单元）中使用。这可以通过编写函数签名后跟分号以及可选前缀来实现。

of `extern`:

foo.h

```
extern int max(int a, int b); // a function declaration

int min(int a, int b); // also a declaration (the extern
                       // is optional/assumed)
```

变量和类及结构的实例的定义方式为：先写数据类型，然后是变量或实例的名称，最后是方括号中的可选数组说明符：

foo.cpp

```
// All of these are variable definitions:
U32 gGlobalInteger = 5;
F32 gGlobalFloatArray[16];
MyClass gGlobalInstance;
```

可以选择使用 extern 关键字声明在一个翻译单元中定义的全局变量以供其他翻译单元使用：

foo.h

```
// These are all pure declarations:
extern U32 gGlobalInteger;
extern F32 gGlobalFloatArray[16];
extern MyClass gGlobalInstance;
```

声明和定义的多样性

毫不奇怪，C/C++ 程序中的任何特定数据对象或函数都可以有多个相同的声明，但每个声明只能有一个定义。如果单个翻译单元中存在两个或多个相同的定义，编译器会注意到多个实体具有相同的名称并标记错误。如果不同的翻译单元中存在两个或多个相同的定义，编译器将无法识别问题，因为它一次只对一个翻译单元进行操作。但在这种情况下，链接器在尝试解析交叉引用时会给出“符号多重定义”错误。

A pure declaration can be provided for a function so that it can be used in other translation units (or later in the same translation unit). This is done by writing a function signature followed by a semicolon, with an optional prefix of `extern`:

foo.h

```
extern int max(int a, int b); // a function declaration

int min(int a, int b); // also a declaration (the extern
                       // is optional/assumed)
```

Variables and instances of classes and `structs` are defined by writing the data type followed by the name of the variable or instance and an optional array specifier in square brackets:

foo.cpp

```
// All of these are variable definitions:
U32 gGlobalInteger = 5;
F32 gGlobalFloatArray[16];
MyClass gGlobalInstance;
```

A global variable defined in one translation unit can optionally be *declared* for use in other translation units by using the `extern` keyword:

foo.h

```
// These are all pure declarations:
extern U32 gGlobalInteger;
extern F32 gGlobalFloatArray[16];
extern MyClass gGlobalInstance;
```

Multiplicity of Declarations and Definitions

Not surprisingly, any particular data object or function in a C/C++ program can have multiple identical *declarations*, but each can have only one *definition*. If two or more identical definitions exist in a single translation unit, the compiler will notice that multiple entities have the same name and flag an error. If two or more identical definitions exist in *different* translation units, the compiler will not be able to identify the problem, because it operates on one translation unit at a time. But in this case, the linker will give us a “multiply defined symbol” error when it tries to resolve the cross-references.

头文件中的定义和内联

将定义放在头文件中通常很危险。原因显而易见：如果一个包含定义的头文件被 #include 到多个 .cpp 文件中，那么很可能会引发“多重定义符号”的链接错误。

内联函数定义是此规则的一个例外，因为每次调用内联函数都会生成一份全新的该函数机器码副本，并直接嵌入到调用函数中。事实上，如果要在多个翻译单元中使用内联函数定义，则必须将其放在头文件中。请注意，仅仅在 .h 文件中使用 inline 关键字标记函数声明，然后将该函数的主体放在 .cpp 文件中是不够的。编译器必须能够“看到”函数主体才能对其进行内联。例如：

foo.h

```
// This function definition will be inlined properly.
inline int max(int a, int b)
{
   return (a > b) ? a : b;
}

// This declaration cannot be inlined because the
// compiler cannot "see" the body of the function.
inline int min(int a, int b);
```

foo.cpp

```
// The body of min() is effectively "hidden" from the
// compiler, so it can ONLY be inlined within foo.cpp.
int min(int a, int b)
{
   return (a <= b) ? a : b;
}
```

inline 关键字实际上只是给编译器的一个提示。它会对每个内联函数进行成本/收益分析，权衡函数代码的大小与内联的潜在性能优势，最终由编译器决定是否真的内联该函数。有些编译器提供了类似 __forceinline 的语法，允许程序员绕过编译器的成本/收益分析，直接控制函数内联。

Definitions in Header Files and Inlining

It is usually dangerous to place *definitions* in header files. The reason for this should be pretty obvious: if a header file containing a *definition* is `#included` into more than one .cpp file, it's a sure-fire way of generating a "multiply defined symbol" linker error.

Inline function definitions are an exception to this rule, because each invocation of an inline function gives rise to a brand new copy of that function's machine code, embedded directly into the calling function. In fact, inline function definitions *must* be placed in header files if they are to be used in more than one translation unit. Note that it is *not* sufficient to tag a function *declaration* with the `inline` keyword in a .h file and then place the body of that function in a .cpp file. The compiler must be able to "see" the body of the function in order to inline it. For example:

foo.h

```
// This function definition will be inlined properly.
inline int max(int a, int b)
{
   return (a > b) ? a : b;
}

// This declaration cannot be inlined because the
// compiler cannot "see" the body of the function.
inline int min(int a, int b);
```

foo.cpp

```
// The body of min() is effectively "hidden" from the
// compiler, so it can ONLY be inlined within foo.cpp.
int min(int a, int b)
{
   return (a <= b) ? a : b;
}
```

The `inline` keyword is really just a hint to the compiler. It does a cost/benefit analysis of each inline function, weighing the size of the function's code versus the potential performance benefits of inling it, and the compiler gets the final say as to whether the function will really be inlined or not. Some compilers provide syntax like `__forceinline`, allowing the programmer to bypass the compiler's cost/benefit analysis and control function inlining directly.

模板和头文件

模板化类或函数的定义必须在所有使用它的翻译单元中对编译器可见。因此，如果您希望模板可在多个翻译单元中使用，则必须将其放入头文件中（就像内联函数定义一样）。因此，模板的声明和定义是不可分割的：您不能在头文件中声明模板化函数或类，而将其定义“隐藏”在 .cpp 文件中，因为这样做会导致这些定义在包含该头文件的任何其他 .cpp 文件中不可见。

3.3.4.3 连锁

C 和 C++ 中的每个定义都有一个称为链接的属性。具有外部链接的定义对于它所在的翻译单元以外的翻译单元可见并可由其引用。具有内部链接的定义只能在其所在的翻译单元内部“看到”，因此不能被其他翻译单元引用。我们将此属性称为链接，因为它决定了是否允许链接器交叉引用相关实体。因此，从某种意义上说，链接相当于翻译单元中的 public: 和 private: 关键字。

默认情况下，定义具有外部链接。static 关键字用于将定义的链接更改为内部链接。请注意，两个或多个不同的 .cpp 文件中的两个或多个相同的静态定义会被链接器视为不同的实体（就像它们被赋予了不同的名称一样），因此它们不会生成“多重定义符号”错误。以下是一些示例：

foo.cpp

```
// This variable can be used by other .cpp files
// (external linkage).
U32 gExternalVariable;

// This variable is only usable within foo.cpp (internal
// linkage).
static U32 gInternalVariable;

// This function can be called from other .cpp files
// (external linkage).
void externalFunction()
{
    // ...
}
```


Templates and Header Files

The definition of a templated class or function must be visible to the compiler across all translation units in which it is used. As such, if you want a template to be usable in more than one translation unit, the template must be placed into a header file (just as inline function definitions must be). The declaration and definition of a template are therefore inseparable: You cannot declare templated functions or classes in a header but “hide” their definitions inside a .cpp file, because doing so would render those definitions invisible within any other .cpp file that includes that header.

3.3.4.3 Linkage

Every definition in C and C++ has a property known as *linkage*. A definition with *external linkage* is visible to and can be referenced by translation units other than the one in which it appears. A definition with *internal linkage* can only be “seen” inside the translation unit in which it appears and thus cannot be referenced by other translation units. We call this property *linkage* because it dictates whether or not the linker is permitted to cross-reference the entity in question. So, in a sense, linkage is the translation unit’s equivalent of the `public:` and `private:` keywords in C++ class definitions.

By default, definitions have external linkage. The `static` keyword is used to change a definition’s linkage to internal. Note that two or more identical `static` definitions in two or more different .cpp files are considered to be *distinct entities* by the linker (just as if they had been given different names), so they will *not* generate a “multiply defined symbol” error. Here are some examples:

foo.cpp

```
// This variable can be used by other .cpp files
// (external linkage).
U32 gExternalVariable;

// This variable is only usable within foo.cpp (internal
// linkage).
static U32 gInternalVariable;

// This function can be called from other .cpp files
// (external linkage).
void externalFunction()
{
    // ...
}
```

```
// This function can only be called from within foo.cpp
// (internal linkage).
static void internalFunction()
{
    // ...
}
```

酒吧.cpp

```
// This declaration grants access to foo.cpp's variable.
extern U32 gExternalVariable;

// This 'gInternalVariable' is distinct from the one
// defined in foo.cpp -- no error. We could just as
// well have named it gInternalVariableForBarCpp -- the
// net effect is the same.
static U32 gInternalVariable;

// This function is distinct from foo.cpp's
// version -- no error. It acts as if we had named it
// internalFunctionForBarCpp().
static void internalFunction()
{
    // ...
}

// ERROR -- multiply defined symbol!
void externalFunction()
{
    // ...
}
```

从技术上讲，声明根本不具有链接属性，因为它们不会在可执行映像中分配任何存储空间；因此，链接器是否应该被允许交叉引用该存储空间是毫无疑问的。声明仅仅是对在其他地方定义的实体的引用。然而，有时将声明称为具有内部链接会更方便，因为声明仅适用于其所在的翻译单元。如果我们以这种方式放宽术语，那么声明始终具有内部链接——无法在多个 .cpp 文件中交叉引用单个声明。（如果我们将声明放在头文件中，则多个 .cpp 文件可以“看到”该声明，但实际上它们各自获取的是该声明的不同副本，并且每个副本在该翻译单元内都有内部链接。）


```
// This function can only be called from within foo.cpp
// (internal linkage).
static void internalFunction()
{
    // ...
}
```

bar.cpp

```
// This declaration grants access to foo.cpp's variable.
extern U32 gExternalVariable;

// This 'gInternalVariable' is distinct from the one
// defined in foo.cpp -- no error. We could just as
// well have named it gInternalVariableForBarCpp -- the
// net effect is the same.
static U32 gInternalVariable;

// This function is distinct from foo.cpp's
// version -- no error. It acts as if we had named it
// internalFunctionForBarCpp().
static void internalFunction()
{
    // ...
}

// ERROR -- multiply defined symbol!
void externalFunction()
{
    // ...
}
```

Technically speaking, *declarations* don't have a linkage property at all, because they do not allocate any storage in the executable image; therefore, there is no question as to whether or not the linker should be permitted to cross-reference that storage. A declaration is merely a reference to an entity defined elsewhere. However, it is sometimes convenient to speak about declarations as having internal linkage, because a declaration only applies to the translation unit in which it appears. If we allow ourselves to loosen our terminology in this manner, then declarations *always* have internal linkage—there is no way to cross-reference a single declaration in multiple .cpp files. (If we put a declaration in a header file, then multiple .cpp files can "see" that declaration, but they are in effect each getting a distinct *copy* of the declaration, and each copy has internal linkage within that translation unit.)

这引出了头文件中允许内联函数定义的真正原因：因为内联函数默认具有内部链接，就像它们被声明为 static 一样。如果多个 .cpp 文件 #include 一个包含内联函数定义的头文件，则每个翻译单元都会获得该函数体的私有副本，并且不会生成“多重定义符号”错误。链接器将每个副本视为一个独立的实体。

3.3.5 C/C++程序的内存布局

用 C 或 C++ 编写的程序会将数据存储在内存中的多个不同位置。为了理解存储空间的分配方式以及各种 C/C++ 变量的工作原理，我们需要了解 C/C++ 程序的内存布局。

3.3.5.1 可执行映像

构建 C/C++ 程序时，链接器会创建一个可执行文件。大多数类 UNIX 操作系统（包括许多游戏机）都采用一种流行的可执行文件格式，称为可执行和链接格式 (ELF)。因此，这些系统上的可执行文件的扩展名是 .elf。Windows 可执行格式类似于 ELF 格式；Windows 下的可执行文件的扩展名是 .exe。无论其格式如何，可执行文件始终包含程序的部分映像，因为它在运行时会存在于内存中。我之所以说“部分”映像，是因为程序通常在运行时除了分配可执行映像中分配的内存之外，还会分配内存。

可执行映像被划分为多个连续的块，这些块被称为段或节。每个操作系统的布局略有不同，并且在同一操作系统上，不同可执行文件的布局也可能略有不同。但映像通常至少由以下四个段组成：

1. 文本段。有时也称为代码段，此块包含程序定义的所有函数的可执行机器代码。

2. 数据段。此段包含所有已初始化的全局变量和静态变量。每个全局变量所需的内存布局与程序运行时完全相同，并已填充适当的初始值。因此，当可执行文件加载到内存中时，已初始化的全局变量和静态变量即可使用。

3. BSS 段。“BSS”是一个过时的名称，代表“由符号启动的块”。此段包含所有未初始化的全局和

This leads us to the real reason why inline function definitions are permitted in header files: it is because inline functions have *internal linkage* by default, just as if they had been declared `static`. If multiple .cpp files `#include` a header containing an inline function definition, each translation unit gets a private copy of that function's body, and no "multiply defined symbol" errors are generated. The linker sees each copy as a distinct entity.

3.3.5 Memory Layout of a C/C++ Program

A program written in C or C++ stores its data in a number of different places in memory. In order to understand how storage is allocated and how the various types of C/C++ variables work, we need to understand the memory layout of a C/C++ program.

3.3.5.1 Executable Image

When a C/C++ program is built, the linker creates an *executable file*. Most UNIX-like operating systems, including many game consoles, employ a popular executable file format called the *executable and linking format* (ELF). Executable files on those systems therefore have a .elf extension. The Windows executable format is similar to the ELF format; executables under Windows have a .exe extension. Whatever its format, the executable file always contains a partial *image* of the program as it will exist in memory when it runs. I say a "partial" image because the program generally allocates memory at runtime in addition to the memory laid out in its executable image.

The executable image is divided into contiguous blocks called *segments* or *sections*. Every operating system lays things out a little differently, and the layout may also differ slightly from executable to executable on the same operating system. But the image is usually comprised of at least the following four segments:

1. *Text segment*. Sometimes called the *code segment*, this block contains executable machine code for all functions defined by the program.

2. *Data segment*. This segment contains all *initialized* global and static variables. The memory needed for each global variable is laid out exactly as it will appear when the program is run, and the proper initial values are all filled in. So when the executable file is loaded into memory, the initialized global and static variables are ready to go.

3. *BSS segment*. "BSS" is an outdated name which stands for "block started by symbol." This segment contains all of the *uninitialized* global and

程序定义的静态变量。C 和 C++ 语言明确将任何未初始化的全局变量或静态变量的初始值定义为零。但是，链接器不会在 BSS 段中存储可能非常大的零块，而是仅存储一个计数，该计数表示需要多少个零字节来容纳段中所有未初始化的全局变量和静态变量。当可执行文件加载到内存中时，操作系统会为 BSS 段预留请求的字节数，并在调用程序的入口点（例如 main() 或 WinMain() ）之前用零填充。

4. 只读数据段。此段有时也称为 rodata 段，包含程序定义的任何只读（常量）全局数据。例如，所有浮点常量（例如， const float kPi = 3.141592f; ）以及所有使用 const 关键字声明的全局对象实例（例如， const Foo gReadOnlyFoo; ）都位于此段中。需要注意的是，整型常量（例如， const int kMaxMonsters = 255; ）通常被编译器用作显式常量，这意味着它们会在使用时直接插入到机器码中。此类常量占用文本段中的存储空间，但不存在于只读数据段中。

全局变量（在文件作用域内，任何函数或类声明之外定义的变量）存储在数据段或 BSS 段中，具体取决于它们是否已初始化。以下全局变量将存储在数据段中，因为它已被初始化：

foo.cpp

```
F32 gInitializedGlobal = -2.0f;
```

并且以下全局变量将根据 BSS 段中给出的规范由操作系统分配并初始化为零，因为它尚未由程序员初始化：

foo.cpp

```
F32 gUninitializedGlobal;
```

我们已经看到，static 关键字可用于为全局变量或函数定义提供内部链接，这意味着它将对其他翻译单元“隐藏”。static 关键字也可用于在函数内声明全局变量。函数静态变量的词法作用域为声明它的函数（即，变量的名称只能在函数内部“看到”）。它在第一次调用函数时初始化（而不是像文件范围静态变量那样在调用 main() 之前初始化）。但就可执行映像中的内存布局而言，函数静态变量的行为相同

static variables defined by the program. The C and C++ languages explicitly define the initial value of any uninitialized global or static variable to be zero. But rather than storing a potentially very large block of zeros in the BSS section, the linker simply stores a *count* of how many zero bytes are required to account for all of the uninitialized globals and statics in the segment. When the executable is loaded into memory, the operating system reserves the requested number of bytes for the BSS section and fills it with zeros prior to calling the program's entry point (e.g., `main()` or `WinMain()`).

4. *Read-only data segment*. Sometimes called the *rodata* segment, this segment contains any read-only (constant) global data defined by the program. For example, all floating-point constants (e.g., `const float kPi = 3.141592f;`) and all global object instances that have been declared with the `const` keyword (e.g., `const Foo gReadOnlyFoo;`) reside in this segment. Note that integer constants (e.g., `const int kMaxMonsters = 255;`) are often used as *manifest constants* by the compiler, meaning that they are inserted directly into the machine code wherever they are used. Such constants occupy storage in the text segment, but they are not present in the read-only data segment.

Global variables (variables defined at *file scope* outside any function or class declaration) are stored in either the data or BSS segments, depending on whether or not they have been initialized. The following global will be stored in the data segment, because it has been initialized:

foo.cpp

```
F32 gInitializedGlobal = -2.0f;
```

and the following global will be allocated and initialized to zero by the operating system, based on the specifications given in the BSS segment, because it has not been initialized by the programmer:

foo.cpp

```
F32 gUninitializedGlobal;
```

We've seen that the `static` keyword can be used to give a global variable or function definition *internal linkage*, meaning that it will be "hidden" from other translation units. The `static` keyword can also be used to declare a global variable *within a function*. A function-static variable is *lexically scoped* to the function in which it is declared (i.e., the variable's name can only be "seen" inside the function). It is initialized the first time the function is called (rather than before `main()` is called, as with file-scope statics). But in terms of memory layout in the executable image, a function-static variable acts identically

到文件静态全局变量——根据它是否已初始化，它存储在数据或 BSS 段中。

```
void readHitchhikersGuide(U32 book)
{
    static U32 sBooksInTheTrilogy = 5; // data segment
    static U32 sBooksRead;             // BSS segment
    // ...
}
```

3.3.5.2 程序堆栈

当可执行程序加载到内存并运行时，操作系统会为程序栈保留一块内存区域。每当调用一个函数时，一块连续的栈内存区域就会被压入栈中——我们称这块内存为栈帧。如果函数 a() 调用另一个函数 b()，则会在 a() 的栈帧之上压入一个 b() 的新栈帧。当 b() 返回时，它的栈帧会被弹出，然后继续执行 a() 中断的地方。

堆栈框架存储三种数据：

1. 它存储调用函数的返回地址，以便被调用函数返回时可以在调用函数中继续执行。

2. 所有相关CPU寄存器的内容都保存在堆栈帧中。
   这使得新函数可以以任何它认为合适的方式使用寄存器，而不必担心覆盖调用函数所需的数据。返回到调用函数时，寄存器的状态将恢复，以便调用函数可以继续执行。被调用函数的返回值（如果有）通常会保留在特定的寄存器中，以便调用函数可以检索它，但其他寄存器将恢复为其原始值。

3. 栈帧还包含函数声明的所有局部变量；这些变量也称为自动变量。这使得每次不同的函数调用都能维护每个局部变量的私有副本，即使函数以递归方式调用自身也是如此。（实际上，有些局部变量实际上是分配给 CPU 寄存器的，而不是存储在栈帧中，但在大多数情况下，这些变量的操作方式就像它们是在函数的栈帧中分配的一样。）

堆栈帧的压栈和弹出通常是通过调整 CPU 中一个寄存器（称为堆栈指针）的值来实现的。图 3.10 演示了执行下面函数时发生的情况。

to a file-static global variable—it is stored in either the data or BSS segment based on whether or not it has been initialized.

```
void readHitchhikersGuide(U32 book)
{
    static U32 sBooksInTheTrilogy = 5; // data segment
    static U32 sBooksRead;             // BSS segment
    // ...
}
```

3.3.5.2 Program Stack

When an executable program is loaded into memory and run, the operating system reserves an area of memory for the *program stack*. Whenever a function is called, a contiguous area of stack memory is pushed onto the stack—we call this block of memory a *stack frame*. If function `a()` calls another function `b()`, a new stack frame for `b()` is pushed on top of `a()`'s frame. When `b()` returns, its stack frame is popped, and execution continues wherever `a()` left off.

A stack frame stores three kinds of data:

1. It stores the *return address* of the calling function so that execution may continue in the calling function when the called function returns.

2. The contents of all relevant *CPU registers* are saved in the stack frame. This allows the new function to use the registers in any way it sees fit, without fear of overwriting data needed by the calling function. Upon return to the calling function, the state of the registers is restored so that execution of the calling function may resume. The return value of the called function, if any, is usually left in a specific register so that the calling function can retrieve it, but the other registers are restored to their original values.

3. The stack frame also contains all *local variables* declared by the function; these are also known as *automatic variables*. This allows each distinct function invocation to maintain its own private copy of every local variable, even when a function calls itself recursively. (In practice, some local variables are actually allocated to CPU registers rather than being stored in the stack frame, but for the most part such variables operate as if they were allocated within the function's stack frame.)

Pushing and popping stack frames is usually implemented by adjusting the value of a single register in the CPU, known as the stack pointer. Figure 3.10 illustrates what happens when the functions shown below are executed.

```
void c()
{
    U32 localC1;
    // ...
}

F32 b()
{
    F32 localB1;
    I32 localB2;

    // ...

    c();

    // ...

    return localB1;
}
```

图 3.10。堆栈框架。


```
void c()
{
    U32 localC1;
    // ...
}

F32 b()
{
    F32 localB1;
    I32 localB2;

    // ...

    c();

    // ...

    return localB1;
}
```

Figure 3.10. Stack frames.

```
void a()
{
    U32 aLocalsA1[5];

    // ...

    F32 localA2 = b();

    // ...
}
```

当包含自动变量的函数返回时，其堆栈帧将被放弃，并且函数中的所有自动变量都应被视为不再存在。从技术上讲，这些变量占用的内存仍然存在于被放弃的堆栈帧中，但一旦调用另一个函数，这些内存很可能会被覆盖。一个常见的错误是返回局部变量的地址，如下所示：

```
U32* getMeaningOfLife()
{
    U32 anInteger = 42;
    return &anInteger;
}
```

如果你立即使用返回的指针，并且在此期间不调用任何其他函数，那么或许可以避免这种情况。但通常情况下，这种代码会崩溃——有时崩溃的方式很难调试。

3.3.5.3 动态分配堆

到目前为止，我们已经了解了程序的数据可以存储为全局变量、静态变量或局部变量。全局变量和静态变量在可执行映像中分配，由可执行文件的数据段和 BSS 段定义。局部变量则分配在程序堆栈上。这两种存储方式都是静态定义的，这意味着在程序编译和链接时，内存的大小和布局是已知的。然而，程序的内存需求通常在编译时无法完全确定。程序通常需要动态分配额外的内存。

为了实现动态分配，操作系统为每个正在运行的进程维护一块内存，进程可以通过调用 malloc()（或 Windows 下类似 HeapAlloc() 的操作系统特定函数）来分配内存，之后可以通过调用 free()（或类似 HeapFree() 的操作系统特定函数）返回内存，供进程在未来某个时间重用。这块内存是


```
void a()
{
    U32 aLocalsA1[5];

    // ...

    F32 localA2 = b();

    // ...
}
```

When a function containing automatic variables returns, its stack frame is abandoned and all automatic variables in the function should be treated as if they no longer exist. Technically, the memory occupied by those variables is still there in the abandoned stack frame—but that memory will very likely be overwritten as soon as another function is called. A common error involves returning the address of a local variable, like this:

```
U32* getMeaningOfLife()
{
    U32 anInteger = 42;
    return &anInteger;
}
```

You *might* get away with this if you use the returned pointer immediately and don't call any other functions in the interim. But more often than not, this kind of code will crash—sometimes in ways that can be difficult to debug.

3.3.5.3 Dynamic Allocation Heap

Thus far, we've seen that a program's data can be stored as global or static variables or as local variables. The globals and statics are allocated within the executable image, as defined by the data and BSS segments of the executable file. The locals are allocated on the program stack. Both of these kinds of storage are *statically* defined, meaning that the size and layout of the memory is known when the program is compiled and linked. However, a program's memory requirements are often not fully known at compile time. A program usually needs to allocate additional memory *dynamically*.

To allow for dynamic allocation, the operating system maintains a block of memory for each running process from which memory can be allocated by calling `malloc()` (or an OS-specific function like `HeapAlloc()` under Windows) and later returned for reuse by the process at some future time by calling `free()` (or an OS-specific function like `HeapFree()`). This memory block is

也称为堆内存，或空闲内存。当我们动态分配内存时，我们有时会说这块内存位于堆上。

在 C++ 中，全局的 new 和 delete 运算符用于在空闲存储区中分配和释放内存。但需要注意的是，个别类可能会重载这些运算符以自定义方式分配内存，甚至全局的 new 和 delete 运算符也可能被重载，因此不能简单地假设 new 总是从全局堆中分配内存。

我们将在第 1 章中更深入地讨论动态内存分配

7. 有关更多信息，请参阅http://en.wikipedia.org/wiki/Dynamic_memory_allocation。

3.3.6 成员变量

C 结构体和 C++ 类允许将变量分组到逻辑单元中。需要注意的是，类或结构体的声明不会分配内存。它仅仅是对数据布局的描述——一个模板，以后可以用来创建该结构体或类的实例。

例如：

```
struct Foo  // struct declaration
{
    U32  mUnsignedValue;
    F32  mFloatValue;
    bool mBooleanValue;
};
```

一旦声明了结构或类，就可以按照分配原始数据类型的任何方式进行分配（定义）；例如，

- 作为自动变量，位于程序堆栈上；

```
void someFunction()
{
    Foo localFoo;
    // ...
}
```

- 作为全局、文件静态或函数静态；

```
Foo gFoo;
static Foo sFoo;

void someFunction()
{
    static Foo sLocalFoo;
    // ...
}
```


known as *heap memory*, or the *free store*. When we allocate memory dynamically, we sometimes say that this memory resides *on the heap*.

In C++, the global `new` and `delete` operators are used to allocate and free memory to and from the free store. Be wary, however—individual classes may overload these operators to allocate memory in custom ways, and even the *global* `new` and `delete` operators can be overloaded, so you cannot simply assume that `new` is always allocating from the global heap.

We will discuss dynamic memory allocation in more depth in Chapter 7. For additional information, see http://en.wikipedia.org/wiki/Dynamic_memory_allocation.

3.3.6 Member Variables

C `structs` and C++ `classes` allow variables to be grouped into logical units. It's important to remember that a `class` or `struct` *declaration* allocates no memory. It is merely a description of the layout of the data—a cookie cutter which can be used to stamp out *instances* of that `struct` or `class` later on. For example:

```
struct Foo  // struct declaration
{
    U32  mUnsignedValue;
    F32  mFloatValue;
    bool mBooleanValue;
};
```

Once a struct or class has been declared, it can be allocated (defined) in any of the ways that a primitive data type can be allocated; for example,

- as an automatic variable, on the program stack;

```
void someFunction()
{
    Foo localFoo;
    // ...
}
```

- as a global, file-static or function-static;

```
Foo gFoo;
static Foo sFoo;

void someFunction()
{
    static Foo sLocalFoo;
    // ...
}
```

- 从堆动态分配。在这种情况下，用于保存数据地址的指针或引用变量本身可以分配为自动、全局、静态甚至动态。

```
Foo* gpFoo = nullptr; // global pointer to a Foo

void someFunction()
{
    // allocate a Foo instance from the heap
    gpFoo = new Foo;

    // ...

    // allocate another Foo, assign to local pointer
    Foo* pAnotherFoo = new Foo;

    // ...

    // allocate a POINTER to a Foo from the heap
    Foo** ppFoo = new Foo*;
    (*ppFoo) = pAnotherFoo;
}
```

3.3.6.1 类静态成员

正如我们所见，static 关键字根据上下文具有许多不同的含义：

- 在文件范围内使用时，静态表示“限制此变量或函数的可见性，以便只能在此 .cpp 文件内看到它。”
- 在函数作用域中使用时，static 表示“这个变量是全局的，不是自动的，但它只能在这个函数内部看到。”
- 当在结构或类声明中使用时，静态意味着“这个变量不是常规成员变量，而是像全局变量一样起作用。”

请注意，当 static 在类声明中使用时，它不会控制变量的可见性（就像在文件作用域中使用时一样），而是区分常规的实例成员变量和充当全局变量的类变量。类静态变量的可见性由类声明中是否使用 public: 、 protected: 或 private: 关键字决定。类静态变量会自动包含在声明它们的类或结构的命名空间中。因此，当在类或结构体之外使用变量时，必须使用类或结构体的名称来消除歧义（例如， Foo::sVarName ）。

- dynamically allocated from the heap. In this case, the pointer or reference variable used to hold the address of the data can itself be allocated as an automatic, global, static or even dynamically.

```
Foo* gpFoo = nullptr; // global pointer to a Foo

void someFunction()
{
    // allocate a Foo instance from the heap
    gpFoo = new Foo;

    // ...

    // allocate another Foo, assign to local pointer
    Foo* pAnotherFoo = new Foo;

    // ...

    // allocate a POINTER to a Foo from the heap
    Foo** ppFoo = new Foo*;
    (*ppFoo) = pAnotherFoo;
}
```

3.3.6.1 Class-Static Members

As we've seen, the `static` keyword has many different meanings depending on context:

- When used at file scope, `static` means "restrict the visibility of this variable or function so it can only be seen inside this .cpp file."
- When used at function scope, `static` means "this variable is a global, not an automatic, but it can only be seen inside this function."
- When used inside a `struct` or `class` declaration, `static` means "this variable is not a regular member variable, but instead acts just like a global."

Notice that when `static` is used inside a class declaration, it does not control the *visibility* of the variable (as it does when used at file scope)—rather, it differentiates between regular per-instance member variables and per-class variables that act like globals. The *visibility* of a class-static variable is determined by the use of `public:`, `protected:` or `private:` keywords in the class declaration. Class-static variables are automatically included within the namespace of the `class` or `struct` in which they are declared. So the name of the `class` or `struct` must be used to disambiguate the variable whenever it is used outside that `class` or `struct` (e.g., `Foo::sVarName`).

与常规全局变量的 extern 声明类似，类中类静态变量的声明不会分配内存。类静态变量的内存必须在 .cpp 文件中定义。例如：

foo.h

```
class Foo
{
public:
    static F32 sClassStatic;  // allocates no
                              // memory!
};
```

foo.cpp

```
F32 Foo::sClassStatic = -1.0f; // define memory and
                               // initialize
```

3.3.7 对象在内存中的布局

图 3.11。简单结构的内存布局。

能够直观地展示类和结构体的内存布局非常有用。这通常非常简单——我们可以简单地为结构体或类画一个方框，并用水平线分隔数据成员。图 3.11 展示了下面列出的结构体 Foo 的内存布局示例。

```
struct Foo
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    I32   mSignedValue;
};
```

图 3.12 记忆布局使用宽度表明成员大小。

数据成员的大小很重要，应该在图表中表示出来。这很容易做到，用每个数据成员的宽度来表示其位大小——例如，32 位整数的宽度大约是 8 位整数的四倍（参见图 3.12）。

```
struct Bar
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    bool  mBooleanValue; // diagram assumes this is 8 bits
};
```


Like an `extern` declaration for a regular global variable, the declaration of a class-static variable within a class allocates no memory. The memory for the class-static variable must be defined in a .cpp file. For example:

foo.h

```
class Foo
{
public:
    static F32 sClassStatic;  // allocates no
                              // memory!
};
```

foo.cpp

```
F32 Foo::sClassStatic = -1.0f; // define memory and
                               // initialize
```

3.3.7 Object Layout in Memory

Figure 3.11. Memory layout of a simple struct.

It's useful to be able to visualize the memory layout of your classes and structs. This is usually pretty straightforward—we can simply draw a box for the struct or class, with horizontal lines separating data members. An example of such a diagram for the `struct Foo` listed below is shown in Figure 3.11.

```
struct Foo
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    I32   mSignedValue;
};
```

Figure 3.12. A memory layout using width to indicate member sizes.

The sizes of the data members are important and should be represented in your diagrams. This is easily done by using the width of each data member to indicate its size in bits—i.e., a 32-bit integer should be roughly four times the width of an eight-bit integer (see Figure 3.12).

```
struct Bar
{
    U32   mUnsignedValue;
    F32   mFloatValue;
    bool  mBooleanValue; // diagram assumes this is 8 bits
};
```

3.3.7.1 对齐和打包

当我们开始更仔细地思考结构体和类在内存中的布局时，我们可能会开始思考，当小数据成员与大数据成员交织在一起时会发生什么。例如：

```
struct InefficientPacking
{
    U32   mU1; // 32 bits
    F32   mF2; // 32 bits
    U8    mB3; // 8 bits
    I32   mI4; // 32 bits
    bool  mB5; // 8 bits
    char* mP6; // 32 bits
};
```

您可能以为编译器只是将数据成员尽可能紧密地打包到内存中。然而，通常情况并非如此。相反，编译器通常会在布局中留下“空洞”，如图 3.13 所示。（可以使用预处理器指令（如 #pragma pack ）或通过命令行选项要求某些编译器不要留下这些空洞；但默认行为是将成员分开，如图 3.13 所示。）编译器为什么会留下这些“空洞”？原因在于每个数据类型都有一个自然对齐，必须尊重这一点才能允许 CPU 有效地读写内存。数据对象的对齐是指它在内存中的地址是否是其大小的倍数（通常是 2 的幂）：

图 3.13. 由于数据成员大小混合导致结构打包效率低下。

- 具有 1 字节对齐的对象驻留在任意内存地址。
- 具有 2 字节对齐的对象仅驻留在偶数地址（即，最低有效半字节为 0x0、0x2、0x4、0x8、0xA、0xC 或 0xE）。
- 具有 4 字节对齐的对象仅驻留在四的倍数的地址上（即，最低有效半字节为 0x0、0x4、0x8 或 0xC 的地址）。
- 16 字节对齐的对象仅驻留在 16 的倍数地址处（即，最低有效半字节为 0x0 的地址）。

对齐非常重要，因为许多现代处理器实际上只能读取和写入正确对齐的数据块。例如，如果程序请求从地址 0x6A341174 读取一个 32 位（4 字节）整数，内存控制器会顺利加载数据，因为该地址是 4 字节对齐的（在本例中，其最低有效半字节为 0x4）。但是，如果请求从地址 0x6A34117 3 加载一个 32 位整数，内存控制器现在必须读取两个 4 字节块：位于 0x6A341170 的块和位于

3.3.7.1 Alignment and Packing

As we start to think more carefully about the layout of our structs and classes in memory, we may start to wonder what happens when small data members are interspersed with larger members. For example:

```
struct InefficientPacking
{
    U32   mU1; // 32 bits
    F32   mF2; // 32 bits
    U8    mB3; // 8 bits
    I32   mI4; // 32 bits
    bool  mB5; // 8 bits
    char* mP6; // 32 bits
};
```

You might imagine that the compiler simply packs the data members into memory as tightly as it can. However, this is not usually the case. Instead, the compiler will typically leave “holes” in the layout, as depicted in Figure 3.13. (Some compilers can be requested not to leave these holes by using a preprocessor directive like `#pragma pack`, or via command-line options; but the default behavior is to space out the members as shown in Figure 3.13.)

Figure 3.13. Inefficient struct packing due to mixed data member sizes.

Why does the compiler leave these “holes”? The reason lies in the fact that every data type has a natural *alignment*, which must be respected in order to permit the CPU to read and write memory effectively. The *alignment* of a data object refers to whether its *address in memory* is a multiple of its *size* (which is generally a power of two):

- An object with 1-byte alignment resides at any memory address.
- An object with 2-byte alignment resides only at even addresses (i.e., addresses whose least significant nibble is 0x0, 0x2, 0x4, 0x8, 0xA, 0xC or 0xE).
- An object with 4-byte alignment resides only at addresses that are a multiple of four (i.e., addresses whose least significant nibble is 0x0, 0x4, 0x8 or 0xC).
- A 16-byte aligned object resides only at addresses that are a multiple of 16 (i.e., addresses whose least significant nibble is 0x0).

Alignment is important because many modern processors can actually only read and write properly aligned blocks of data. For example, if a program requests that a 32-bit (4-byte) integer be read from address 0x6A341174, the memory controller will load the data happily because the address is 4-byte aligned (in this case, its least significant nibble is 0x4). However, if a request is made to load a 32-bit integer from address 0x6A341173, the memory controller now has to read *two* 4-byte blocks: the one at 0x6A341170 and the one

图 3.14。32 位整数的对齐和未对齐读取。

位于 0x6A341174。然后，它必须对 32 位整数的两部分进行掩码和移位，并进行逻辑或运算，将其存入 CPU 的目标寄存器。如图 3.14 所示。

有些微处理器甚至还没到这个地步。如果你请求读取或写入未对齐的数据，你可能会得到垃圾数据。或者你的程序可能会完全崩溃！（PlayStation 2 就是这种对未对齐数据不容忍的典型例子。）

不同的数据类型有不同的对齐要求。一个好的经验法则是，数据类型应该与等于其字节宽度的边界对齐。例如，32 位值通常有 4 字节对齐要求，16 位值应有 2 字节对齐要求，而 8 位值可以存储在任何地址（1 字节对齐）。在支持 SIMD 矢量运算的 CPU 上，每个矢量寄存器包含四个 32 位浮点数，总共 128 位或 16 字节。正如您所猜测的，一个四浮点 SIMD 矢量通常有 16 字节对齐要求。

图 3.15. 通过将小成员分组在一起，可以提高打包效率。

这让我们回到图 3.13 中 struct Inefficient Packing 布局中的那些“空洞”。当结构体或类中较小的数据类型（例如 8 位 bool 类型）与较大的类型（例如 32 位整数或浮点型）交织在一起时，编译器会引入填充（空洞），以确保所有内容都正确对齐。在声明数据结构时，最好考虑对齐和打包问题。只需重新排列上例中 struct InefficientPacking 的成员，我们就可以消除一些浪费的填充空间，如下所示和图 3.15 所示：

Figure 3.14. Aligned and unaligned reads of a 32-bit integer.

at 0x6A341174. It must then mask and shift the two parts of the 32-bit integer and logically OR them together into the destination register on the CPU. This is shown in Figure 3.14.

Some microprocessors don't even go this far. If you request a read or write of unaligned data, you might just get garbage. Or your program might just crash altogether! (The PlayStation 2 is a notable example of this kind of intolerance for unaligned data.)

Different data types have different alignment requirements. A good rule of thumb is that a data type should be aligned to a boundary equal to the width of the data type in bytes. For example, 32-bit values generally have a 4-byte alignment requirement, 16-bit values should be 2-byte aligned, and 8-bit values can be stored at any address (1-byte aligned). On CPUs that support SIMD vector math, the vector registers each contain four 32-bit floats, for a total of 128 bits or 16 bytes. And as you would guess, a four-float SIMD vector typically has a 16-byte alignment requirement.

Figure 3.15. More efficient packing by grouping small members together.

This brings us back to those "holes" in the layout of `struct Inefficient Packing` shown in Figure 3.13. When smaller data types like 8-bit `bools` are interspersed with larger types like 32-bit integers or `floats` in a structure or class, the compiler introduces padding (holes) in order to ensure that everything is properly aligned. It's a good idea to think about alignment and packing when declaring your data structures. By simply rearranging the members of `struct InefficientPacking` from the example above, we can eliminate some of the wasted padding space, as shown below and in Figure 3.15:

```
struct MoreEfficientPacking
{
    U32   mU1; // 32 bits (4-byte aligned)
    F32   mF2; // 32 bits (4-byte aligned)
    I32   mI4; // 32 bits (4-byte aligned)
    char* mP6; // 32 bits (4-byte aligned)
    U8    mB3; // 8 bits  (1-byte aligned)
    bool  mB5; // 8 bits  (1-byte aligned)
};
```

在图 3.15 中，您会注意到整个结构的大小现在是 20 字节，而不是我们预期的 18 字节，因为它在末尾填充了两个字节。此填充由编译器添加，以确保结构在数组上下文中正确对齐。也就是说，如果定义了这些结构的数组，并且数组的第一个元素已对齐，那么末尾的填充将保证所有后续元素也正确对齐。

结构体整体的对齐等于其成员中最大的对齐要求。在上面的例子中，成员的最大对齐要求是 4 字节，因此整个结构体也应该 4 字节对齐。我通常喜欢在结构体的末尾添加显式填充，以使浪费的空间清晰可见，如下所示：

```
struct BestPacking
{
    U32   mU1;     // 32 bits (4-byte aligned)
    F32   mF2;     // 32 bits (4-byte aligned)
    I32   mI4;     // 32 bits (4-byte aligned)
    char* mP6;     // 32 bits (4-byte aligned)
    U8    mB3;     // 8 bits (1-byte aligned)
    bool  mB5;     // 8 bits (1-byte aligned)
    U8    _pad[2]; // explicit padding
};
```

3.3.7.2 C++类的内存布局

有两件事使得 C++ 类在内存布局方面与 C 结构略有不同：继承和虚函数。

当类 B 继承自类 A 时，B 的数据成员在内存中会紧跟在类 A 之后，如图 3.16 所示。每个新的派生类都只是将其数据成员添加到末尾，尽管对齐要求可能会导致类之间出现填充。（多重继承会产生一些奇怪的效果，例如在派生类的内存布局中包含单个基类的多个副本。我们在此不讨论细节，因为游戏程序员通常更倾向于完全避免多重继承。）

图 3.16。继承对类布局的影响。


```
struct MoreEfficientPacking
{
    U32   mU1; // 32 bits (4-byte aligned)
    F32   mF2; // 32 bits (4-byte aligned)
    I32   mI4; // 32 bits (4-byte aligned)
    char* mP6; // 32 bits (4-byte aligned)
    U8    mB3; // 8 bits  (1-byte aligned)
    bool  mB5; // 8 bits  (1-byte aligned)
};
```

You'll notice in Figure 3.15 that the size of the structure as a whole is now 20 bytes, not 18 bytes as we might expect, because it has been padded by two bytes at the end. This padding is added by the compiler to ensure proper alignment of the structure in an *array context*. That is, if an array of these structs is defined and the first element of the array is aligned, then the padding at the end guarantees that *all subsequent elements* will also be aligned properly.

The alignment of a structure as a whole is equal to the largest alignment requirement among its members. In the example above, the largest member alignment is 4-byte, so the structure as a whole should be 4-byte aligned. I usually like to add explicit padding to the end of my structs to make the wasted space visible and explicit, like this:

```
struct BestPacking
{
    U32   mU1;     // 32 bits (4-byte aligned)
    F32   mF2;     // 32 bits (4-byte aligned)
    I32   mI4;     // 32 bits (4-byte aligned)
    char* mP6;     // 32 bits (4-byte aligned)
    U8    mB3;     // 8 bits (1-byte aligned)
    bool  mB5;     // 8 bits (1-byte aligned)
    U8    _pad[2]; // explicit padding
};
```

3.3.7.2 Memory Layout of C++ Classes

Two things make C++ classes a little different from C structures in terms of memory layout: *inheritance* and *virtual functions*.

When class B inherits from class A, B's data members simply appear immediately after A's in memory, as shown in Figure 3.16. Each new derived class simply tacks its data members on at the end, although alignment requirements may introduce padding between the classes. (Multiple inheritance does some whacky things, like including multiple copies of a single base class in the memory layout of a derived class. We won't cover the details here, because game programmers usually prefer to avoid multiple inheritance altogether anyway.)

Figure 3.16. Effect of inheritance on class layout.

如果某个类包含或继承了一个或多个虚函数，则会在类布局中添加四个额外的字节（如果目标硬件使用 64 位地址，则为八个字节），通常在类布局的最开始处。这四个或八个字节统称为虚拟表指针或 vpointer，因为它们包含指向称为虚函数表或 vtable 的数据结构的指针。特定类的 vtable 包含指向它声明或继承的所有虚函数的指针。每个具体类都有自己的虚拟表，并且该类的每个实例都有一个指向它的指针，存储在其 vpointer 中。

虚函数表是多态性的核心，因为它允许编写代码时无需考虑它所处理的具体类。回到那个随处可见的例子：Shape 基类有 Circle 、 Rectangle 和 Triangle 派生类。假设 Shape 定义了一个名为 Draw() 的虚函数。所有派生类都重写了这个函数，分别提供了名为 Circle::Draw() 、 Rectangle::Draw() 和 Triangle::Draw() 的不同实现。任何从 Shape 派生的类的虚函数表都会包含一个 Draw() 函数的条目，但该条目指向的函数实现取决于具体的类。Circle 的虚函数表包含一个指向 Circle::Draw() 的指针，而 Rectangle 的虚函数表指向 Rectangle::Draw() ，Triangle 的虚函数表指向 Triangle::Draw() 。给定一个指向 Shape 的任意指针（ Shape* pShape ），代码可以简单地取消引用虚表指针，在虚表中查找 Draw() 函数的入口，然后调用它。结果是，当 pShape 指向 Circle 实例时，调用 Circle ::Draw() ；当 pShape 指向 Rectangle 实例时，调用 Rectangle::Draw() ；当 pSha pe 指向

`Triangle`.

以下代码片段说明了这些想法。请注意，基类 Shape 定义了两个虚函数，SetId () 和 Draw()，后者被声明为纯虚函数。（这意味着 Shape 没有提供 Draw() 函数的默认实现，派生类如果想要实例化，必须重写它。）Circle 类派生自 Shap e，添加了一些数据成员和函数来管理其中心和半径，并重写了 Draw() 函数；如图 3.17 所示。Triangle 类也派生自 Shape。它添加了一个 Vector3 对象数组来存储其三个顶点，并添加了一些函数来获取和设置各个顶点。正如我们所料，Triangle 类重写了 Draw()，为了便于说明，它还重写了 SetId()。Triangle 类生成的内存映像如图 3.18 所示。

If a class contains or inherits one or more *virtual functions*, then four additional bytes (or eight bytes if the target hardware uses 64-bit addresses) are added to the class layout, typically at the very beginning of the class' layout. These four or eight bytes are collectively called the *virtual table pointer* or *vpointer*, because they contain a pointer to a data structure known as the *virtual function table* or *vtable*. The vtable for a particular class contains pointers to all the virtual functions that it declares or inherits. Each concrete class has its own virtual table, and every instance of that class has a pointer to it, stored in its vpointer.

The virtual function table is at the heart of polymorphism, because it allows code to be written that is ignorant of the specific concrete classes it is dealing with. Returning to the ubiquitous example of a `Shape` base class with derived classes for `Circle`, `Rectangle` and `Triangle`, let's imagine that `Shape` defines a virtual function called `Draw()`. The derived classes all override this function, providing distinct implementations named `Circle::Draw()`, `Rectangle::Draw()` and `Triangle::Draw()`. The virtual table for any class derived from `Shape` will contain an entry for the `Draw()` function, but that entry will point to different function implementations, depending on the concrete class. `Circle`'s vtable will contain a pointer to `Circle::Draw()`, while `Rectangle`'s virtual table will point to `Rectangle::Draw()`, and `Triangle`'s vtable will point to `Triangle::Draw()`. Given an arbitrary pointer to a `Shape` (`Shape* pShape`), the code can simply dereference the vtable pointer, look up the `Draw()` function's entry in the vtable, and call it. The result will be to call `Circle::Draw()` when `pShape` points to an instance of `Circle`, `Rectangle::Draw()` when `pShape` points to a `Rectangle`, and `Triangle::Draw()` when `pShape` points to a `Triangle`.

These ideas are illustrated by the following code excerpt. Notice that the base class `Shape` defines two virtual functions, `SetId()` and `Draw()`, the latter of which is declared to be *pure virtual*. (This means that `Shape` provides no default implementation of the `Draw()` function, and derived classes *must* override it if they want to be instantiable.) Class `Circle` derives from `Shape`, adds some data members and functions to manage its center and radius, and overrides the `Draw()` function; this is depicted in Figure 3.17. Class `Triangle` also derives from `Shape`. It adds an array of `Vector3` objects to store its three vertices and adds some functions to get and set the individual vertices. Class `Triangle` overrides `Draw()` as we'd expect, and for illustrative purposes it also overrides `SetId()`. The memory image generated by the `Triangle` class is shown in Figure 3.18.

图 3.17。pShape1 指向 Circle 类的一个实例。

```
class Shape
{
public:
    virtual void  SetId(int id) { m_id = id; }
    int           GetId() const { return m_id; }

    virtual void  Draw() = 0; // pure virtual -- no impl.

private:
    int  m_id;
};

class Circle : public Shape
{
public:
    void     SetCenter(const Vector3& c) { m_center=c; }
    Vector3  GetCenter() const { return m_center; }

    void     SetRadius(float r) { m_radius = r; }
    float    GetRadius() const { return m_radius; }

    virtual void Draw()
    {
        // code to draw a circle
    }

private:
    Vector3  m_center;
    float    m_radius;
};
```

Figure 3.17. `pShape1` points to an instance of class `Circle`.

```
class Shape
{
public:
    virtual void  SetId(int id) { m_id = id; }
    int           GetId() const { return m_id; }

    virtual void  Draw() = 0; // pure virtual -- no impl.

private:
    int  m_id;
};

class Circle : public Shape
{
public:
    void     SetCenter(const Vector3& c) { m_center=c; }
    Vector3  GetCenter() const { return m_center; }

    void     SetRadius(float r) { m_radius = r; }
    float    GetRadius() const { return m_radius; }

    virtual void Draw()
    {
        // code to draw a circle
    }

private:
    Vector3  m_center;
    float    m_radius;
};
```

```
class Triangle : public Shape
{
public:
    void    SetVertex(int i, const Vector3& v);
    Vector3 GetVertex(int i) const { return m_vtx[i]; }

    virtual void Draw()
    {
        // code to draw a triangle
    }

    virtual void SetId(int id)
    {
        // call base class' implementation
        Shape::SetId(id);

        // do additional work specific to Triangles...
    }

private:
    Vector3  m_vtx[3];
};


// ----------------------------

void main(int, char**)
{
    Shape* pShape1 = new Circle;
    Shape* pShape2 = new Triangle;

    pShape1->Draw();
    pShape2->Draw();

    // ...
}
```

3.4 计算机硬件基础

使用 C++、C# 或 Python 等高级语言进行编程是构建软件的有效方法。但是，语言级别越高，它就越能让你免受代码运行硬件底层细节的干扰。要成为一名真正精通的程序员，了解目标硬件的架构至关重要。这些知识可以帮助你更好地理解硬件。


```
class Triangle : public Shape
{
public:
    void    SetVertex(int i, const Vector3& v);
    Vector3 GetVertex(int i) const { return m_vtx[i]; }

    virtual void Draw()
    {
        // code to draw a triangle
    }

    virtual void SetId(int id)
    {
        // call base class' implementation
        Shape::SetId(id);

        // do additional work specific to Triangles...
    }

private:
    Vector3  m_vtx[3];
};


// ----------------------------

void main(int, char**)
{
    Shape* pShape1 = new Circle;
    Shape* pShape2 = new Triangle;

    pShape1->Draw();
    pShape2->Draw();

    // ...
}
```


3.4 Computer Hardware Fundamentals

Programming in a high-level language like C++, C# or Python is an efficient way to build software. But the higher-level your language is, the more it shields you from the underlying details of the hardware on which your code runs. To become a truly proficient programmer, it's important to understand the architecture of your target hardware. This knowledge can help you to op-

图 3.18。pShape2 指向 Triangle 类的一个实例。

优化代码。这对于并发编程也至关重要——所有程序员如果希望充分利用现代计算硬件中不断提升的并行度，就必须理解并发。

3.4.1 向过去的简单计算机学习

在接下来的篇幅中，我们将讨论一个简单的通用 CPU 的设计，而不是深入探讨任何特定处理器的细节。然而，有些读者可能会发现，阅读一些真实 CPU 的细节有助于我们进行一些理论性讨论。我自己在青少年时期就通过编程学习了计算机的工作原理，当时我的 Apple II 电脑和 Commodore 64 电脑都配备了一个简单的 CPU，称为 6502，由 MOS Technology Inc. 设计和制造。我还通过阅读和使用英特尔整个 x86 CPU 系列的共同祖先 8086（及其近亲 8088）获得了一些知识。由于其简单性，这两款处理器都非常适合学习。6502 尤其如此，它是我用过的最简单的 CPU。一旦你了解了 6502 和/或 8086 的工作原理，现代 CPU 就会更容易理解。

为此，这里有一些关于 6502 和 8086 架构和编程细节的优秀资源：

- Gary B. Little 所著《Inside the Apple IIe》[33] 一书的第 1 章和第 2 章对 6502 汇编语言编程进行了精彩的概述。该书可在线获取，网址为 http://www.apple2scans.net/files/InsidetheIIe.pdf。

Figure 3.18. `pShape2` points to an instance of class `Triangle`.

timize your code. It's also crucial for concurrent programming—and all programmers must understand concurrency if they hope to take full advantage of the ever-increasing degree of parallelism in modern computing hardware.

3.4.1 Learning from the Simpler Computers of Yesteryear

In the following pages, we'll discuss the design of a simple, generic CPU, rather than diving into the specifics of any one particular processor. However, some readers may find it helpful to ground our somewhat theoretical discussion by reading about the specifics of a real CPU. I myself learned how computers work when I was a teenager, by programming my Apple II computer and the Commodore 64. Both of these machines had a simple CPU known as the 6502, which was designed and manufactured by MOS Technology Inc. I also learned some of what I know by reading about and working with the common ancestor to Intel's entire x86 line of CPUs, the 8086 (and its cousin the 8088). Both of these processors are great for learning purposes due to their simplicity. This is especially true of the 6502, which is the simplest CPU I've ever worked with. Once you understand how a 6502 and/or an 8086 works, modern CPUs will be a lot easier to understand.

To that end, here are a few great resources on the details of 6502 and 8086 architecture and programming:

- Chapters 1 and 2 of the book "Inside the Apple IIe" by Gary B. Little [33] provides a great overview of 6502 assembly language programming. The book is available online at http://www.apple2scans.net/files/InsidetheIIe.pdf.

图 3.19 简单的串行冯·诺依曼计算机架构。

• http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html 对 x86 指令集架构进行了很好的概述。

• 您还应该阅读 Michael Abrash 的《图形编程黑皮书》[1]，其中有大量关于 8086 汇编编程的有用信息，以及游戏早期软件优化和图形编程的绝妙技巧。

3.4.2 计算机的解剖

最简单的计算机由一个中央处理器 (CPU) 和一个内存组组成，它们通过一条或多条总线在称为主板的电路板上相互连接，并通过一组 I/O 端口和/或扩展槽连接到外部外围设备。这种基本设计被称为冯·诺依曼架构，因为它是由数学家兼物理学家约翰·冯·诺依曼及其同事于 1945 年在机密的 ENIAC 项目上首次描述的。图 3.19 展示了一个简单的串行计算机架构。

3.4.3 中央处理器

CPU 是计算机的“大脑”。它由以下组件组成：

• 用于执行整数运算和位移的算术/逻辑单元（ALU），

• 用于执行浮点运算的浮点单元（FPU）（通常使用 IEEE 754 浮点标准表示），

• 几乎所有现代 CPU 都包含一个矢量处理单元 (VPU)，它能够并行对多个数据项执行浮点和整数运算，

Figure 3.19. A simple serial von Neumann computer architecture.

- http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html gives a nice overview of the x86 instruction set architecture.
- You should also definitely check out *Michael Abrash's Graphics Programming Black Book* [1] for loads of useful information about assembly programming on the 8086, plus incredible tips for software optimization and graphics programming back in the early days of gaming.

3.4.2 Anatomy of a Computer

The simplest computer consists of a *central processing unit* (CPU) and a bank of *memory*, connected to one another on a circuit board called the *motherboard* via one or more *buses*, and connected to external peripheral devices by means of a set of *I/O ports* and/or *expansion slots*. This basic design is referred to as the *von Neumann architecture* because it was first described in 1945 by the mathematician and physicist John von Neumann and his colleagues while working on the classified ENIAC project. A simple serial computer architecture is illustrated in Figure 3.19.

3.4.3 CPU

The CPU is the "brains" of the computer. It consists of the following components:

- an *arithmetic/logic unit* (ALU) for performing integer arithmetic and bit shifting,
- a *floating-point unit* (FPU) for doing floating-point arithmetic (typically using the IEEE 754 floating-point standard representation),
- virtually all modern CPUs also contain a *vector processing unit* (VPU) which is capable of performing floating-point and integer operations on multiple data items in parallel,

图 3.20 串行 CPU 的典型组件。

• 用于与片上和片外存储设备接口的存储控制器（MC）或存储管理单元（MMU），

• 一组寄存器，在计算过程中充当临时存储（以及其他用途），以及

• 控制单元 (CU)，用于解码和调度机器语言指令到芯片上的其他组件，并在它们之间路由数据。

所有这些组件都由一个周期性的方波信号（称为时钟）驱动。时钟频率决定了CPU执行指令或算术运算等操作的速率。

串行CPU的典型组件如图3.20所示。

3.4.3.1 ALU

ALU 执行一元和二进制算术运算，例如求反、加、减、乘和除，以及逻辑运算，例如与、或、异或（简称 XOR 或 EOR）、按位补码和位移位。在某些 CPU 设计中，ALU 在物理上被拆分为算术单元 (AU) 和逻辑单元 (LU)。

ALU 通常只执行整数运算。浮点计算需要截然不同的电路，通常由物理上独立的浮点单元 (FPU) 执行。早期的 CPU（例如英特尔 8088/8086）没有片上 FPU；如果需要浮点运算支持，这些 CPU 必须使用单独的 FPU 协处理器芯片（例如英特尔 8087）进行增强。在后期的 CPU 设计中，FPU 通常集成在主 CPU 芯片上。

Figure 3.20. Typical components of a serial CPU.

- a *memory controller* (MC) or *memory management unit* (MMU) for interfacing with on-chip and off-chip memory devices,
- a bank of *registers* which act as temporary storage during calculations (among other things), and
- a *control unit* (CU) for decoding and dispatching machine language instructions to the other components on the chip, and routing data between them.

All of these components are driven by a periodic square wave signal known as the *clock*. The frequency of the clock determines the rate at which the CPU performs operations such as executing instructions or performing arithmetic. The typical components of a serial CPU are shown in Figure 3.20.

3.4.3.1 ALU

The ALU performs unary and binary arithmetic operations such as negation, addition, subtraction, multiplication and division, and also performs logical operations such as AND, OR, exclusive OR (abbreviated XOR or EOR), bitwise complement and bit shifting. In some CPU designs, the ALU is physically split into an arithmetic unit (AU) and a logic unit (LU).

An ALU typically performs integer operations only. Floating-point calculations require very different circuitry, and are usually performed by a physically separate floating-point unit (FPU). Early CPUs like the Intel 8088/8086 had no on-chip FPU; if floating-point math support was desired, these CPUs had to be augmented with a separate FPU co-processor chip such as the Intel 8087. In later CPU designs, an FPU was typically included on the main CPU die.

3.4.3.2 视觉处理单元

矢量处理单元 (VPU) 的作用有点像 ALU/FPU 的组合，因为它通常可以执行整数和浮点运算。VPU 的独特之处在于它能够将算术运算符应用于输入数据向量（每个向量通常由 2 到 16 个浮点值或最多 64 个不同宽度的整数值组成），而不是标量输入。矢量处理也称为单指令多数据 (SIMD)，因为单个算术运算符（例如乘法）可同时应用于多对输入。有关更多详细信息，请参阅第 4.10 节。

如今的 CPU 实际上并不包含 FPU 本身。所有浮点计算，即使是涉及标量浮点值的计算，都由 VPU 执行。消除 FPU 可以减少 CPU 芯片上的晶体管数量，从而允许使用这些晶体管来实现更大的缓存、更复杂的乱序执行逻辑等等。正如我们将在 4.10.6 节中看到的那样，优化编译器通常会将对浮点变量执行的数学运算转换为使用 VPU 的矢量化代码。

3.4.3.3 寄存器

为了最大限度地提高性能，ALU 或 FPU 通常只能对存储在特殊高速存储单元（称为寄存器）中的数据进行计算。寄存器通常在物理上与计算机主存储器分离，位于芯片内部，并且靠近访问它们的组件。它们通常使用快速、高成本的多端口静态 RAM 或 SRAM 实现。（有关内存技术的更多信息，请参见第 3.4.5 节。）CPU 中的一组寄存器称为寄存器文件。

由于寄存器不属于主存，[1] 因此它们通常没有地址，但有名称。这些名称可能很简单，例如 R0、R1、R2 等，尽管早期的 CPU 倾向于使用字母或简短的助记符作为寄存器名称。例如，英特尔 8088/8086 有四个 16 位通用寄存器，分别名为 AX、BX、CX 和 DX。MOS Technology, Inc. 生产的 6502 使用称为累加器 2 (A) 的寄存器执行所有算术运算，并使用两个称为 X 和 Y 的辅助寄存器执行其他操作，例如对数组进行索引。

1 一些早期计算机确实使用主 RAM 来实现寄存器。例如，IBM 7030 Stretch（IBM 首台基于晶体管的超级计算机）中的 32 个寄存器“覆盖”在主 RAM 的前 32 个地址上。在一些早期的 ALU 设计中，它的一个输入来自寄存器，另一个输入来自主 RAM。这些设计很实用，因为当时 RAM 的访问延迟相对于 CPU 的整体性能来说很低。

2 “累加器”一词的出现是因为早期的 ALU 每次只处理一位，因此需要通过屏蔽并将各个位移入结果寄存器来累积答案。

3.4.3.2 VPU

A vector processing unit (VPU) acts a bit like a combination ALU/FPU, in that it can typically perform both integer and floating-point arithmetic. What differentiates a VPU is its ability to apply arithmetic operators to *vectors* of input data—each of which typically consists of between two and 16 floating-point values or up to 64 integer values of various widths—rather than to scalar inputs. Vector processing is also known as *single instruction multiple data* or SIMD, because a single arithmetic operator (e.g, multiplication) is applied to multiple pairs of inputs simultaneously. See Section 4.10 for more details.

Today's CPUs don't actually contain an FPU *per se*. Instead, all floating-point calculations, even those involving scalar `float` values, are performed by the VPU. Eliminating the FPU reduces the transistor count on the CPU die, allowing those transistors to be used to implement larger caches, more-complex out-of-order execution logic, and so on. And as we'll see in Section 4.10.6, optimizing compilers will typically convert math performed on `float` variables into *vectorized* code that uses the VPU anyway.

3.4.3.3 Registers

In order to maximize performance, an ALU or FPU can usually only perform calculations on data that exists in special high-speed memory cells called *registers*. Registers are typically physically separate from the computer's main memory, located on-chip and in close proximity to the components that access them. They're usually implemented using fast, high-cost multi-ported static RAM or SRAM. (See Section 3.4.5 for more information about memory technologies.) A bank of registers within a CPU is called a *register file*.

Because registers aren't part of main memory,[1] they typically don't have addresses but they do have names. These could be as simple as R0, R1, R2 etc., although early CPUs tended to use letters or short mnemonics as register names. For example, the Intel 8088/8086 had four 16-bit general-purpose registers named AX, BX, CX and DX. The 6502 by MOS Technology, Inc. performed all of its arithmetic operations using a register known as the *accumulator*[2] (A), and used two auxillary registers called X and Y for other operations such as indexing into an array.

[1] Some early computers did use main RAM to implement registers. For example, the 32 registers in the IBM 7030 Stretch (IBM's first transistor-based supercomputer) were "overlaid" on the first 32 addresses in main RAM. In some early ALU designs, one of its inputs would come from a register while the other came from main RAM. These designs were practical because back then, RAM access latencies were low relative to the overall performance of the CPU.

[2] The term "accumulator" arose because early ALUs used to work one bit at a time, and would therefore *accumulate* the answer by masking and shifting the individual bits into the result register.

CPU 中的一些寄存器被设计用于常规计算。它们被恰当地命名为通用寄存器（GPR)。每个 CPU 还包含一些专用寄存器 (SPR)。这些寄存器包括：

- 指令指针（IP），
- 堆栈指针（SP），
- 基指针（BP）和
- 状态寄存器。

指令指针

指令指针 (IP) 包含机器语言程序中当前正在执行的指令的地址（有关机器语言的更多信息，请参见第 3.4.7.2 节）。

堆栈指针

在 3.3.5.2 节中，我们了解了程序的调用栈如何既作为函数相互调用的主要机制，又作为为局部变量分配内存的手段。栈指针 (SP) 包含程序调用栈顶部的地址。栈的内存地址可以向上或向下增长，但为了便于讨论，我们假设它向下增长。在这种情况下，可以通过从栈指针的值中减去数据项的大小，然后将该项写入 SP 指向的新地址，从而将数据项压入栈中。同样，也可以通过从 SP 指向的地址读取数据项，然后将其大小添加到 SP 指向的新地址，从而将数据项从栈中弹出。

基指针

基指针 (BP) 包含当前函数在调用栈上的栈帧的基地址。函数的许多局部变量都分配在其栈帧内，但优化器可能会将其他局部变量在函数运行期间独占地分配给某个寄存器。栈分配的变量占用一段距离基指针唯一偏移量的内存地址。只需从 BP 中存储的地址中减去其唯一偏移量（假设堆栈向下增长），即可在内存中定位此类变量。

状态寄存器

一种称为状态寄存器、条件代码寄存器或标志的特殊寄存器

Some of the registers in a CPU are designed to be used for general calculations. They're appropriately named *general-purpose registers* (GPR). Every CPU also contains a number of *special-purpose registers* (SPR). These include:

- the *instruction pointer* (IP),
- the *stack pointer* (SP),
- the *base pointer* (BP) and
- the *status register*.

Instruction Pointer

The *instruction pointer* (IP) contains the address of the currently-executing instruction in a machine language program (more on machine language in Section 3.4.7.2).

Stack Pointer

In Section 3.3.5.2 we saw how a program's *call stack* serves both as the primary mechanism by which functions call one another, and as the means by which memory is allocated for local variables. The *stack pointer* (SP) contains the address of the top of the program's call stack. The stack can grow up or down in terms of memory addresses, but for the purpose of this discussion let's assume it grows downward. In this case, a data item may be *pushed* onto the stack by subtracting the size of the item from the value of the stack pointer, and then writing the item at the new address pointed to by SP. Likewise, an item can be *popped* off the stack by reading it from the address pointed to by SP, and then adding its size to SP.

Base Pointer

The *base pointer* (BP) contains the base address of the current function's *stack frame* on the call stack. Many of a function's local variables are allocated within its stack frame, although the optimizer may assign others exclusively to a register for the duration of the function. Stack-allocated variables occupy a range of memory addresses at a unique offset from the base pointer. Such a variable can be located in memory by simply subtracting its unique offset from the address stored in BP (assuming the stack grows down).

Status Register

A special register known as the *status register*, *condition code register* or *flags*

寄存器包含反映最新 ALU 运算结果的位。例如，如果减法运算的结果为零，则状态寄存器中的零位（通常称为“Z”）将被置位，否则该位将被清除。同样，如果加法运算导致溢出，即二进制 1 必须“进位”到多字加法的下一个字，则进位位（通常称为“C”）将被置位，否则将被清除。

状态寄存器中的标志可用于通过条件分支控制程序流，或者可用于执行后续计算，例如在多字加法中将进位添加到下一个字。

寄存器格式

重要的是要理解，FPU 和 VPU 通常使用各自的专用寄存器组进行操作，而不是使用 ALU 的通用整数寄存器。原因之一是速度——寄存器离使用它们的计算单元越近，访问其数据所需的时间就越短。另一个原因是 FPU 和 VPU 的寄存器通常比 ALU 的 GPR 更宽。

例如，32 位 CPU 的每个 GPR 都是 32 位宽，但 FPU 可能操作 64 位双精度浮点数，甚至 80 位“扩展”双精度值，这意□□味着其寄存器必须分别有 64 位或 80 位宽。同样，VPU 使用的每个寄存器都需要包含一个输入数据向量，这意味着这些寄存器必须比典型的 GPR 宽得多。例如，英特尔的 SSE2（流式 SIMD 扩展）向量处理器可以配置为对包含四个单精度（32 位）浮点值或两个双精度（64 位）值的向量执行计算。因此，每个 SSE2 向量寄存器的宽度均为 128 位。

在 FPU 普及的时代，ALU 和 FPU 之间寄存器的物理分离是导致 int 和 float 之间转换成本高昂的原因之一。不仅每个值的位模式必须在其二进制补码整数格式和 IEEE 754 浮点表示之间来回转换，而且还必须在通用整数寄存器和 FPU 寄存器之间进行物理传输。然而，如今的 CPU 本身已不再包含 FPU——所有浮点运算通常都由矢量处理单元 (VPU) 执行。VPU 可以同时处理整数和浮点运算，两者之间的转换成本要低得多，即使将数据从整数 GPU 移动到矢量寄存器或反之亦然。即便如此，避免在 int 和 float 格式之间转换数据仍然是一个好主意。

register contains bits that reflect the results of the most-recent ALU operation. For instance, if the result of a subtraction is zero, the *zero bit* (typically named “Z”) is set within the status register, otherwise the bit is cleared. Likewise, if an add operation resulted in an overflow, meaning that a binary 1 must be “carried” to the next word of a multi-word addition, the *carry bit* (often named “C”) is set, otherwise it is cleared.

The flags in the status register can be used to control program flow via *conditional branching*, or they can be used to perform subsequent calculations, such as adding the carry bit to the next word in a multi-word addition.

Register Formats

It’s important to understand that the FPU and VPU typically operate on their own private sets of registers, rather than making use of the ALU’s general-purpose integer registers. One reason for this is speed—the closer the registers are to the compute unit that uses them, the less time is needed to access the data they contain. Another reason is that the FPU’s and VPU’s registers are typically *wider* than the ALU’s GPRs.

For example, a 32-bit CPU has GPRs that are 32 bits wide each, but an FPU might operate on 64-bit double-precision floats, or even 80-bit “extended” double-precision values, meaning that its registers have to be 64 or 80 bits wide, respectively. Likewise, each register used by a VPU needs to contain a *vector* of input data, meaning that these registers must be much wider than a typical GPR. For example, Intel’s SSE2 (streaming SIMD extensions) vector processor can be configured to perform calculations on vectors containing either four single-precision (32-bit) floating-point values each, or two double-precision (64-bit) values each. Hence SSE2 vector registers are each 128 bits wide.

The physical separation of registers between ALU and FPU is one reason why conversions between `int` and `float` were very expensive, back in the days when FPUs were commonplace. Not only did the bit pattern of each value have to be converted back and forth between its two’s complement integer format and its IEEE 754 floating-point representation, but the data also had to be transferred physically between the general-purpose integer registers and the FPU’s registers. However, today’s CPUs no longer contain an FPU per se—all floating-point math is typically performed by a vector processing unit. A VPU can handle both integer and floating-point math, and conversions between the two are much less expensive, even when moving data from an integer GPU into a vector register or vice-versa. That said, it’s still a good idea to avoid converting data between `int` and `float` formats

尽可能地这样做，因为即使是低成本的转换也比没有转换要昂贵。

3.4.3.4 控制单元

如果说 CPU 是计算机的“大脑”，那么控制单元 (CU) 就是 CPU 的“大脑”。它的工作是管理 CPU 内部的数据流，并协调 CPU 所有其他组件的运行。

计算单元 (CU) 运行程序的方式是读取机器语言指令流，将每条指令分解为操作码和操作数进行解码，然后根据当前指令的操作码发出工作请求和/或将数据路由到 ALU、FPU、VPU、寄存器和/或内存控制器。在流水线 CPU 和超标量 CPU 中，计算单元 (CU) 还包含复杂的电路来处理分支预测和指令的乱序执行调度。我们将在 3.4.7 节中更详细地介绍计算单元 (CU) 的工作原理。

3.4.4 时钟

每个数字电子电路本质上都是一个状态机。为了改变状态，电路必须由数字信号驱动。这种信号可以通过将电路中某条线路的电压从 0 伏变为 3.3 伏，或反之亦然来提供。

CPU 内的状态变化通常由称为系统时钟的周期性方波信号驱动。此信号的每个上升沿或下降沿称为一个时钟周期，CPU 在每个周期可以执行至少一个基本操作。因此，对于 CPU 来说，时间似乎是量化的。3 CPU 执行其操作的速率由系统时钟的频率决定。在过去的几十年里，时钟速度显着提高。20 世纪 70 年代开发的早期 CPU，如 MOS 技术的 6502 和英特尔的 8086/8088 CPU，其时钟频率在 1-2 MHz 范围内（每秒数百万个周期）。今天的 CPU，如英特尔酷睿 i7，通常时钟频率在 2-4 GHz 范围内（每秒数十亿个周期）。

重要的是要意识到，一条 CPU 指令并不一定需要一个时钟周期来执行。并非所有指令都生来平等——有些指令非常简单，而有些则更为复杂。有些指令在底层实现为更简单的微操作（μ 操作）的组合，因此执行所需的周期比更简单的指令要多得多。

3 这与模拟电子电路形成对比，在模拟电子电路中，时间被视为连续的。例如，老式信号发生器可以产生真正的正弦波，其电压随时间在 -5 伏和 5 伏之间平滑变化。

where possible, because even a low-cost conversion is still more expensive than no conversion.

3.4.3.4 Control Unit

If the CPU is the “brains” of the computer, then the *control unit* (CU) is the “brains” of the CPU. Its job is to manage the flow of data within the CPU, and to orchestrate the operation of all of the CPU’s other components.

The CU runs a program by reading a stream of *machine language* instructions, decoding each instruction by breaking it into its opcode and operands, and then issuing work requests and/or routing data to the ALU, FPU, VPU, the registers and/or the memory controller as dictated by the current instruction’s opcode. In pipelined and superscalar CPUs, the CU also conains complex circuitry to handle branch prediction and the scheduling of instructions for out-of-order execution. We’ll look at the operation of the CU in more detail in Section 3.4.7.

3.4.4 Clock

Every digital electronic circuit is essentially a state machine. In order for it to change states, the circuit must be driven to do so by a digital signal. Such a signal might be provided by *changing the voltage* on a line in the circuit from 0 volts to 3.3 volts, or vice-versa.

State changes within a CPU are typically driven by a periodic square wave signal known as the *system clock*. Each rising or falling edge of this signal is known as a *clock cycle*, and the CPU can perform at least one primitive operation on every cycle. To a CPU, time therefore appears to be *quantized*.[3]

The rate at which a CPU can perform its operations is governed by the *frequency* of the system clock. Clock speeds have increased significantly over the past few decades. Early CPUs developed in the 1970s, like the MOS Technology’s 6502 and Intel’s 8086/8088 CPUs, had clocks that ran in the 1–2 MHz range (millions of cycles per second). Today’s CPUs, like the Intel Core i7, are typically clocked in the 2–4 GHz range (*billions* of cycles per second).

It’s important to realize that one CPU instruction doesn’t necessarily take one clock cycle to execute. Not all instructions are created equal—some instructions are very simple, while others are more complex. Some instructions are implemented under the hood as a combination of simpler micro-operations (μ-ops) and therefore take many more cycles to execute than their simpler counterparts.

[3] Contrast this to an *analog* electronic circuit, where time is treated as *continuous*. For example, an old-school signal generator can produce a true sine wave that smoothly varies between, say, −5 volts and 5 volts over time.

此外，虽然早期的 CPU 确实可以在一个时钟周期内执行一些指令，但如今的流水线 CPU 甚至将最简单的指令分解为多个阶段。流水线 CPU 中的每一阶段都需要一个时钟周期来执行，这意味着具有 N 级流水线的 CPU 的最小指令延迟为 N 个时钟周期。简单的流水线 CPU 可以每个时钟周期一条指令的速率退出指令，因为每个时钟滴答都会将一条新指令送入流水线。但是，如果要跟踪流水线中的某条特定指令，则需要 N 个周期才能从流水线的起点移动到终点。我们将在 4.2 节中更深入地讨论流水线 CPU。

3.4.4.1 时钟速度与处理能力

CPU 或计算机的“处理能力”有多种定义。一种常见的衡量标准是机器的吞吐量，即在给定时间间隔内可以执行的操作数。吞吐量以每秒百万条指令 (MIPS) 或每秒浮点运算次数 (FLOPS) 为单位表示。

由于指令或浮点运算通常不会在一个周期内完成，并且不同指令的运行所需的周期数也不同，因此 CPU 的 MIPS 或 FLOPS 指标只是平均值。因此，您不能简单地查看 CPU 的时钟频率并确定其以 MIPS 或 FLOPS 为单位的处理能力。例如，运行在 3 GHz 的串行 CPU，其中一次浮点乘法平均需要六个周期才能完成，理论上可以达到 0.5 GFLOPS。但是，包括流水线、超标量设计、矢量处理、多核 CPU 和其他形式的并行性在内的许多因素共同作用，混淆了时钟速度和处理能力之间的关系。因此，确定 CPU 或计算机真实处理能力的唯一方法是测量它——通常是通过运行标准化基准测试。

3.4.5 内存

计算机中的内存就像邮局里的邮箱，每个邮箱或“单元”通常包含一个字节的数据（八位值）。每个单字节存储单元都由其地址标识——一个简单的编号

4 实际上，早期计算机通常以大于 8 位的“字”为单位访问内存。例如，IBM 701（1952 年生产）以 36 位字为单位寻址内存，而 PDP-1（1959 年生产）可以访问多达 4096 个 18 位内存字。1972 年，英特尔 8008 处理器普及了八位字节。当时，编码大小写英文字母都需要七位。通过将其扩展至八位，还可以支持各种特殊字符。

Also, while early CPUs could truly execute some instructions in a single clock cycle, today's pipelined CPUs break even the simplest instruction down into multiple *stages*. Each stage in a pipelined CPU takes one clock cycle to execute, meaning that a CPU with an *N*-stage pipeline has a minimum *instruction latency* of *N* clock cycles. A simple pipelined CPU can retire instructions at a *rate* of one instruction per clock cycle, because a new instruction is fed into the pipeline each clock tick. But if you were to trace one particular instruction through the pipeline, it would take *N* cycles to move from start to finish through the pipeline. We'll discuss pipelined CPUs in more depth in Section 4.2.

3.4.4.1 Clock Speed versus Processing Power

The "processing power" of a CPU or computer can be defined in various ways. One common measure is the *throughput* of the machine—the number of operations it can perform during a given interval of time. Throughput is expressed either in units of millions of instructions per second (MIPS) or floating-point operations per second (FLOPS).

Because instructions or floating-point operations don't generally complete in exactly one cycle, and because different instructions take differing numbers of cycles to run, the MIPS or FLOPS metrics of a CPU are just averages. As such, you cannot simply look at the clock frequency of a CPU and determine its processing power in MIPS or FLOPS. For example, a serial CPU running at 3 GHz, in which one floating-point multiply takes six cycles to complete on average, could theoretically achieve 0.5 GFLOPS. But many factors including pipelining, superscalar designs, vector processing, multicore CPUs and other forms of parallelism conspire to obfuscate the relationship between clock speed and processing power. Thus the only way to determine the true processing power of a CPU or computer is to measure it—typically by running standardized benchmarks.

3.4.5 Memory

The *memory* in a computer acts like a bank of mailboxes at the post office, with each box or "cell" typically containing a single *byte* of data (an eight-bit value).[4] Each one-byte memory cell is identified by its *address*—a simple numbering

[4] Actually, early computers often accessed memory in units of "words" that were larger than 8 bits. For example, the IBM 701 (produced in 1952) addressed memory in units of 36-bit words, and the PDP-1 (made in 1959) could access up to 4096 18-bit memory words. Eight-bit bytes were popularized by the Intel 8008 in 1972. Seven bits are required to encode both lower- and uppercase English letters. By extending this to eight bits, a wide range of special characters could also be supported.

方案范围从 0 到 N − 1，其中 N 是可寻址内存的大小（以字节为单位）。

记忆有两种基本类型：

- 只读存储器（ROM），以及
- 读/写存储器，由于历史原因被称为随机存取存储器 (RAM)。

即使没有通电，ROM 模块也能保留其数据。某些类型的 ROM 只能编程一次。其他类型的 ROM，称为电可擦除可编程 ROM (EEPROM)，可以反复重新编程。（闪存驱动器就是 EEPROM 存储器的一个例子。）RAM 可以进一步分为静态 RAM (SRAM) 和动态 RAM (DRAM)。只要通电，静态 RAM 和动态 RAM 都会保留其数据。但与静态 RAM 不同的是，动态 RAM 还需要定期"刷新"（通过读取数据然后重新写入）以防止其内容消失。这是因为 DRAM 存储单元由 MOS 电容器构成，而 MOS 电容器会逐渐失去电荷，读取此类存储单元会破坏其中包含的数据。

RAM 还可以根据其他各种设计特征进行分类，例如：

- 它是否是多端口的，这意味着它可以被 CPU 内的多个组件同时访问；
- 它是通过与时钟（SDRAM）同步运行还是异步运行；
- 以及它是否支持双倍数据速率访问 (DDR)，这意味着可以在时钟的上升沿和下降沿读取或写入 RAM。

3.4.6 公交车

数据通过称为总线的连接在 CPU 和内存之间传输。总线只是一束称为线路的并行数字"线路"，每条线路可以代表一位数据。当线路传输电压信号 6 时，它表示二进制 1；当线路没有电压时，它表示二进制 1。

5 随机存取存储器 (RAM) 之所以得名，是因为早期的存储器技术使用延迟循环来存储数据，这意味着数据只能按照写入的顺序读取。RAM 技术改进了这种情况，允许数据随机访问，即以任意顺序访问。

6 早期的晶体管-晶体管逻辑 (TTL) 设备在 5 伏的电源电压下工作，因此 5 伏信号代表二进制 1。当今大多数数字电子设备都采用互补金属氧化物半导体逻辑 (CMOS)，它可以在较低的电源电压下工作，通常在 1.2 到 3.3 伏之间。

scheme ranging from 0 to $N - 1$, where N is the size of addressable memory in bytes.

Memory comes in two basic flavors:

- *read-only memory* (ROM), and
- read/write memory, known for historical reasons[5] as *random access memory* (RAM).

ROM modules retain their data even when power is not applied to them. Some types of ROM can be programmed only once. Others, known as *electronically erasable programmable ROM* or EEPROM, can be reprogrammed over and over again. (Flash drives are one example of EEPROM memory.)

RAM can be further divided into *static* RAM (SRAM) and *dynamic* RAM (DRAM). Both static and dynamic RAM retain their data as long as power is applied to them. But unlike static RAM, dynamic RAM also needs to be "refreshed" periodically (by reading the data and then re-writing it) in order to prevent its contents from disappearing. This is because DRAM memory cells are built from MOS capacitors that gradually lose their charge, and reading such memory cells is destructive to the data they contain.

RAM can also be categorized by various other design characteristics, such as:

- whether it is *multi-ported*, meaning that it be accessed simultaneously by more than one component within the CPU;
- whether it operates by being synchronized to a clock (SDRAM) or asynchronously;
- and whether or not it supports double data rate access (DDR), meaning the RAM can be read or written on both rising and falling clock edges.

3.4.6 Buses

Data is transferred between the CPU and memory over connections known as *buses*. A bus is just a bundle of parallel digital "wires" called *lines*, each of which can represent a single bit of data. When the line carries a voltage signal[6] it represents a binary one, and when the line has no voltage applied

[5] Random-access memory was so named because earlier memory technologies used delay loops to store data, meaning that it could only be read in the order it was written. RAM technology improved on this situation by permitting data to be accessed at random, i.e., in any order.

[6] Early *transistor-transistor* logic (TTL) devices operated at a supply voltage of 5 volts, so a 5-volt signal would represent a binary 1. Most of today's digital electronic devices utilize *complementary metal oxide semiconductor* logic (CMOS) which can operate at a lower supply voltage, typically between 1.2 and 3.3 volts.

（0伏）表示二进制零。一束n条并联的单位线可以传输n位数（即0到

$2^n - 1$).

典型的计算机包含两条总线：地址总线和数据总线。CPU 通过地址总线向内存控制器提供地址，将数据从内存单元加载到其某个寄存器中。内存控制器的响应方式是将存储在相关单元中的数据项的位呈现到数据总线上，CPU 可以在数据总线上“看到”这些数据项。同样，CPU 通过在地址总线上广播目标地址并将要写入的数据项的位模式放置在数据总线上来将数据项写入内存。内存控制器的响应方式是将给定的数据写入相应的内存单元。我们应该注意，地址和数据总线有时实现为两组物理上独立的线路，有时实现为一组线路，它们在内存访问周期的不同阶段在地址和数据总线功能之间复用。

3.4.6.1 总线宽度

地址总线的宽度（以位为单位）控制着 CPU 可以访问的地址范围（即机器中可寻址内存的大小）。例如，具有 16 位地址总线的计算机最多可以访问 64 Ki B 内存，地址范围为 0x0000 到 0xFFFF。具有 32 位地址总线的计算机可以访问 4 GiB 内存，地址范围为 0x00000000 到 0xFFFFFFFF。而具有 64 位地址总线的计算机可以访问惊人的 16 EiB（艾字节）内存。也就是 2 64 = 16 × 1024 6 ≈ 1.8 × 10 19 字节！

数据总线的宽度决定了 CPU 寄存器和内存之间一次可以传输多少数据。（数据总线的宽度通常与 CPU 中的通用寄存器相同，但并非总是如此。）8 位数据总线意味着数据可以一次传输一个字节——从内存加载 16 位值需要两个独立的内存周期，一个用于获取最低有效字节，另一个用于获取最高有效字节。另一方面，64 位数据总线可以将内存和 64 位寄存器之间的数据传输作为单个内存操作。

访问比机器数据总线宽度更窄的数据项是可能的，但这通常比访问与数据总线宽度匹配的数据项成本更高。例如，在 64 位机器上读取 16 位值时，仍然必须从内存中读取完整的 64 位数据。然后，需要屏蔽所需的 16 位字段，并可能将其移入目标寄存器中。这就是为什么 C 语言

(0 volts) it represents a binary zero. A bundle of n single-bit lines arranged in parallel can transmit an n-bit number (i.e., any number in the range 0 through $2^n - 1$).

A typical computer contains two buses: An *address bus* and a *data bus*. The CPU loads data from a memory cell into one of its registers by supplying an address to the memory controller via the address bus. The memory controller responds by presenting the bits of the data item stored in the cell(s) in question onto the data bus, where it can be “seen” by the CPU. Likewise, the CPU writes a data item to memory by broadcasting the destination address over the address bus and placing the bit pattern of the data item to write onto the data bus. The memory controller responds by writing the given data into the corresponding memory cell(s). We should note that the address and data buses are sometimes implemented as two physically separate sets of wires, and sometimes as a single set of wires which are multiplexed between address and data bus functions during different phases of the memory access cycle.

3.4.6.1 Bus Widths

The *width* of the *address bus*, measured in bits, controls the range of possible addresses that can be accessed by the CPU (i.e., the size of *addressable memory* in the machine). For example, a computer with a 16-bit address bus can access at most 64 KiB of memory, using addresses in the range 0x0000 through 0xFFFF. A computer with a 32-bit address bus can access a 4 GiB memory, using addresses in the range 0x00000000 through 0xFFFFFFFF. And a machine with a 64-bit address bus can access a staggering 16 EiB (exbibytes) of memory. That’s $2^{64} = 16 \times 1024^6 \approx 1.8 \times 10^{19}$ bytes!

The width of the *data bus* determines how much data can be transferred between CPU registers and memory at a time. (The data bus is typically the same width as the general-purpose registers in the CPU, although this isn’t always the case.) An 8-bit data bus means that data can be transferred one byte at a time—loading a 16-bit value from memory would require two separate memory cycles, one to fetch the least-significant byte and one to fetch the most-significant byte. At the other end of the spectrum, a 64-bit data bus can transfer data between memory and a 64-bit register as a single memory operation.

It’s possible to access data items that are narrower than the width of a machine’s data bus, but it’s typically more costly than accessing items whose widths match that of the data bus. For example, when reading a 16-bit value on a 64-bit machine, a full 64 bits worth of data must still be read from memory. The desired 16-bit field then has to be masked off and possibly shifted into place within the destination register. This is one reason why the C language

不会将 int 定义为特定的位数——它被特意定义为与目标机器上字的“自然”大小相匹配，目的是提高源代码的可移植性。（讽刺的是，由于对 int 宽度的隐式假设，这种策略实际上导致源代码的可移植性降低。）

3.4.6.2 单词

“字”这个术语通常用来描述多字节值。然而，构成一个字的字节数并没有统一的定义。它在一定程度上取决于上下文。

有时，“字”一词指的是最小的多字节值，即 16 位或两个字节。在这种情况下，双字为 32 位（四个字节），四字为 64 位（八个字节）。这就是 Windows API 中“字”一词的用法。

另一方面，“字”一词也用于指代特定机器上数据项的“自然”大小。例如，一台具有 32 位寄存器和 32 位数据总线的机器，最自然地以 32 位（四字节）值运行，程序员和硬件人员有时会说这样的机器的字长为 32 位。这里的要点是，当你听到“字”一词被用来指代数据项的大小时，要注意上下文。

3.4.6.3 n 位计算机

您可能遇到过“ n 位计算机”这个术语。它通常指具有 n 位数据总线和/或寄存器的计算机。但该术语有点模糊，因为它也可能指地址总线为 n 位宽的计算机。此外，在某些 CPU 上，数据总线和寄存器宽度并不匹配。例如，8088 具有 16 位寄存器和 16 位地址总线，但它只有 8 位数据总线。因此，它在内部运行类似于 16 位机器，但其 8 位数据总线使其在内存访问方面表现得像 8 位机器。

再次强调，在谈论 n 位机器时要注意上下文。

3.4.7 机器语言和汇编语言

就 CPU 而言，“程序”只不过是相对简单的指令的连续流。每条指令都会指示控制单元 (CU)，并最终指示 CPU 内的其他组件（例如内存控制器、ALU、FPU 或 VPU）执行操作。指令可能会在计算机内部或 CPU 内部移动数据，也可能以某种方式转换数据（例如，通过执行算术运算）。

does not define an `int` to be a specific number of bits wide—it was purposefully defined to match the “natural” size of a word on the target machine, in an attempt to make source code more portable. (Ironically this policy has actually resulted in source code often being *less* portable due to implicit assumptions about the width of an `int`.)

3.4.6.2 Words

The term “word” is often used to describe a multi-byte value. However, the number of bytes comprising a word is not universally defined. It depends to some degree on context.

Sometimes the term “word” refers to the smallest multi-byte value, namely 16 bits or two bytes. In that context, a double word would be 32 bits (four bytes) and a quad word would be 64 bits (eight bytes). This is the way the term “word” is used in the Windows API.

On the other hand, the term “word” is also used to refer to the “natural” size of data items on a particular machine. For example, a machine with 32-bit registers and a 32-bit data bus operates most naturally with 32-bit (four byte) values, and programmers and hardware folks will sometimes say that such a machine as a *word size* of 32 bits. The takeaway here is to be aware of context whenever you hear the term “word” being used to refer to the size of a data item.

3.4.6.3 *n*-Bit Computers

You may have encountered the term “*n*-bit computer.” This usually means a machine with an *n*-bit data bus and/or registers. But the term is a bit ambiguous, because it might also refer to a computer whose address bus is *n* bits wide. Also, on some CPUs the data bus and register widths don’t match. For example, the 8088 had 16-bit registers and a 16-bit address bus, but it only had an 8-bit data bus. Hence it acted like a 16-bit machine internally, but its 8-bit data bus caused it to behave like an 8-bit machine in terms of memory accesses. Again, be aware of context when talking about an *n*-bit machine.

3.4.7 Machine and Assembly Language

As far as the CPU is concerned, a “program” is nothing more than a sequential stream of relatively simple *instructions*. Each instruction tells the control unit (CU), and ultimately the other components within the CPU such as the memory controller, ALU, FPU or VPU, to perform an operation. An instruction might move data around within the computer or within the CPU itself, or it might transform that data in some manner (e.g., by performing an arith-

指令集（对数据进行统计或逻辑运算）。通常，程序中的指令是按顺序执行的，但有些指令可以通过“跳转”到程序整体指令流中的新位置来改变这种顺序控制流。

3.4.7.1 指令集架构（ISA）

不同制造商的 CPU 设计差异很大。特定 CPU 支持的所有指令集，以及 CPU 设计的其他各种细节，例如寻址模式和内存指令格式，被称为其指令集架构 (ISA)。（请勿将其与编程语言的应用程序二进制接口 (ABI) 混淆，后者定义了更高级的协议，例如调用约定。）我们不会在这里尝试介绍任何 CPU ISA 的细节，但以下几类指令类型几乎适用于所有 ISA：

- 移动指令。这些指令在寄存器之间或内存和寄存器之间移动数据。某些 ISA 将“移动”指令拆分为单独的“加载”和“存储”指令。
- 算术运算。这些当然包括加法、减法、乘法和除法，但也可能包括其他运算，如一元否定、倒数、平方根等。
- 按位运算符。这些包括AND，OR，排他OR（缩写为XOR或EOR）和按位补码。
- 移位/循环运算符。这些指令允许将数据字中的位向左或向右移动，无论是否影响状态寄存器中的进位，或者进行循环移动（从字的一端滚出的位“环绕”到另一端）。
- 比较。这些指令允许比较两个值，以确定其中一个值小于、大于还是等于另一个值。在大多数 CPU 中，比较指令使用 ALU 将两个输入值相减，从而设置状态寄存器中的相应位，但减法的结果会被丢弃。
- 跳转和分支指令。这些指令允许通过将新地址存储到指令指针中来改变程序流程。这可以是无条件执行的（在这种情况下称为“跳转”指令），也可以是基于状态寄存器中各个标志位的状态有条件执行的（在这种情况下通常称为“分支”指令）。例如，“为零时分支”指令仅在状态寄存器中的“Z”位置位时才会改变IP的内容。

metic or logical operation on the data). Normally the instructions in a program are executed sequentially, although some instructions can alter this sequential flow of control by “jumping” to a new spot within the program’s overall instruction stream.

3.4.7.1 Instruction Set Architecture (ISA)

CPU designs vary widely from one manufacturer to the next. The set of all instructions supported by a given CPU, along with various other details of the CPU’s design like its addressing modes and the in-memory instruction format, is called its *instruction set architecture* or ISA. (This is not to be confused with a programming language’s *application binary interface* or ABI, which defines higher-level protocols like calling conventions.) We won’t attempt to cover the details of any one CPU’s ISA here, but the following categories of instruction types are common to pretty much every ISA:

- *Move*. These instructions move data between registers, or between memory and a register. Some ISAs break the “move” instruction into separate “load” and “store” instructions.
- *Arithmetic operations*. These of course include addition, subtraction, multiplication and division, but may also include other operations like unary negation, inversion, square root, and so on.
- *Bitwise operators*. These include AND, OR, exclusive OR (abbreviated XOR or EOR) and bitwise complement.
- *Shift/rotate operators*. These instructions allow the bits within a data word to be shifted left or right, with or without affecting the carry bit in the status register, or rotated (where the bits rolling off one end of the word “wrap around” to the other end).
- *Comparison*. These instructions allow two values to be compared, in order to determine if one is less than, greater than or equal to the other. In most CPUs, the comparison instructions use the ALU to subtract the two input values, thereby setting the appropriate bits in the status register, but the result of the subtraction is simply discarded.
- *Jump and branch*. These allow program flow to be altered by storing a new address into the instruction pointer. This can be done either unconditionally (in which case it is called a “jump” instruction) or conditionally based on the state of various flags in the status register (in which case it is often called a “branch”). For example, a “branch if zero” instruction alters the contents of the IP if and only if the “Z” bit in the status register is set.

• 推送和弹出。大多数 CPU 提供特殊指令，用于将寄存器的内容推送到程序堆栈，以及将堆栈顶部的当前值弹出到寄存器中。

• 函数调用和返回。一些 ISA 提供了用于调用函数（也称为过程或子例程）并从中返回的显式指令。但是，函数调用和返回语义也可以通过 push、pop 和 jump 指令的组合来提供。

• 中断。“中断”指令会在 CPU 内部触发数字信号，使其暂时跳转到预先设置的中断服务例程（通常不包含在正在运行的程序中）。中断用于通知操作系统或用户程序事件，例如外围设备上的输入可用。用户程序也可以触发中断，以“调用”操作系统内核的例程。更多详情，请参见 4.4.2 节。

• 其他指令类型。大多数 ISA 支持各种不属于上述类别的指令类型。例如，“无操作”指令（通常称为 NOP）除了引入短暂延迟外没有任何作用。NOP 指令也会消耗内存，在某些 ISA 中，它们用于在内存中正确对齐后续指令。

我们不可能在这里列出所有指令类型，但如果您好奇，您可以随时阅读真实处理器（如 Intel x86）的 ISA 文档（可在 http://intel.ly/2woVFQ8 上找到）。

3.4.7.2 机器语言

计算机只能处理数字。因此，程序指令流中的每条指令都必须以数字形式编码。当程序以这种方式编码时，我们称它是用机器语言（简称 ML）编写的。当然，机器语言并非单一的语言——它实际上是多种语言的集合，每种语言对应不同的 CPU/ISA。

每条机器语言指令都由三个基本部分组成：

• 操作码，告诉 CPU 执行哪个操作（加、减、移动、跳转等），

• 零个或多个操作数，用于指定指令的输入和/或输出，以及

• 某种选项字段，指定指令的寻址模式和可能的其他标志等内容。

- *Push and pop*. Most CPUs provide special instructions for pushing the contents of a register onto the program stack and for popping the current value at the top of the stack into a register.
- *Function call and return*. Some ISAs provide explicit instructions for calling a function (also known as a procedure or subroutine) and returning from it. However, function call and return semantics can also be provided by a combination of push, pop and jump instructions.
- *Interrupts*. An “interrupt” instruction triggers a digital signal within the CPU that causes it to jump temporarily to a pre-installed *interrupt service routine* routine which is often not part of the program being run. Interrupts are used to notify the operating system or a user program of events such as an input becoming available on a peripheral device. Interrupts can also be triggered by user programs in order to “call” into the operating system kernel’s routines. See section 4.4.2 for more details.
- *Other instruction types*. Most ISAs support a variety of instruction types that don’t fall into one of the categories listed above. For example, the “no-op” instruction (often called NOP) is an instruction that has no effect other than to introduce a short delay. NOP instructions also consume memory, and on some ISAs they are used to align subsequent instructions properly in memory.

We can’t possibly list all instruction types here, but if you’re curious you can always read through the ISA documentation of a real processor like the Intel x86 (which is available at http://intel.ly/2woVFQ8).

3.4.7.2 Machine Language

Computers can only deal with numbers. As such, each instruction in a program’s instruction stream must be encoded numerically. When a program is encoded in this way, we say it is written in *machine language*, or ML for short. Of course, machine language isn’t a single language—it’s really a multitude of languages, one for each distinct CPU/ISA.

Every machine language instruction is comprised of three basic parts:

- an *opcode*, which tells the CPU which operation to perform (add, subtract, move, jump, etc.),
- zero or more *operands* which specify the inputs and/or outputs of the instruction, and
- some kind of *options field*, specifying things like the *addressing mode* of the instruction and possibly other flags.

图 3.21 两种假设的机器语言指令编码方案。上图：在变宽编码方案中，不同的指令可能占用不同数量的内存字节。下图：定宽指令编码对指令流中的每条指令使用相同数量的字节。

操作数有多种类型。有些指令可能将一个或多个寄存器的名称（编码为数字 ID）作为操作数。其他指令可能期望将字面值作为操作数（例如，“将值 5 加载到寄存器 R2”或“跳转到地址 0x0102ED5C”）。CPU 解释和使用指令操作数的方式称为该指令的寻址模式。我们将在 3.4.7.4 节中更详细地讨论寻址模式。

ML 指令的操作码和操作数（如果有）被打包成一个连续的位序列，称为指令字。一个假设的 CPU 可能会按照图 3.21 所示对其指令进行编码，其中第一个字节可能包含操作码、寻址模式和各种选项标志，后面跟着一些用于操作数的字节。每个 ISA 对指令字宽度（即每条指令占用的位数）的定义不同。在某些 ISA 中，所有指令都占用固定数量的位数；这是精简指令集计算机（RISC）的典型特征。在其他 ISA 中，不同类型的指令可以编码成不同大小的指令字；这在复杂指令集计算机（CISC）中很常见。

在某些微控制器中，指令字可以短至四位，也可以长达数字节。指令字通常是 32 位或 64 位的倍数，因为这与 CPU 寄存器和/或数据总线的宽度相匹配。在超长指令字 (VLIW) CPU 设计中，并行性是通过将多个操作编码成一个非常宽的指令字来实现的，以便并行执行。因此，VLIW ISA 中的指令宽度可以高达数百字节。

Figure 3.21. Two hypothetical machine language instruction encoding schemes. Top: In a variable-width encoding scheme, different instructions may occupy different numbers of bytes in memory. Bottom: A fixed-width instruction encoding uses the same number of bytes for every instruction in an instruction stream.

Operands come in many flavors. Some instructions might take the names (encoded as numeric ids) of one or more registers as operands. Others might expect a literal value as an operand (e.g., “load the value 5 into register R2,” or “jump to address 0x0102ED5C”). The way in which an instruction’s operands are interpreted and used by the CPU is known as the instruction’s *addressing mode*. We’ll discuss addressing modes in more detail in Section 3.4.7.4.

The opcode and operands (if any) of an ML instruction are packed into a contiguous sequence of bits called an *instruction word*. A hypothetical CPU might encode its instructions as shown in Figure 3.21 with perhaps the first byte containing the opcode, addressing mode and various options flags, followed by some number of bytes for the operands. Each ISA defines the width of an instruction word (i.e., the number of bits occupied by each instruction) differently. In some ISAs, all instructions occupy a fixed number of bits; this is typical of *reduced instruction set computers* (RISC). In other ISAs, different types of instructions may be encoded into differently-sized instruction words; this is common in *complex instruction set computers* (CISC).

Instruction words can be as small as four bits in some microcontrollers, or they may be many bytes in size. Instruction words are often multiples of 32 or 64 bits, because this matches the width of the CPU’s registers and/or data bus. In *very long instruction word* (VLIW) CPU designs, parallelism is achieved by allowing multiple operations to be encoded into a single very wide instruction word, for the purpose of executing them in parallel. Instructions in a VLIW ISA can therefore be upwards of hundreds of bytes in width.

有关 Intel x86 ISA 上的机器语言指令编码的具体示例，请参阅 http://aturing.umcs.maine.edu/~meadow/courses/cos335/Asm07-MachineLanguage.pdf。

3.4.7.3 汇编语言

直接用机器语言编写程序非常繁琐且容易出错。为了方便程序员，开发了一种简单的基于文本的机器语言版本，称为汇编语言。在汇编语言中，给定 CPU ISA 中的每条指令都有一个助记符 - 一个简短的英文单词或缩写，比相应的数字操作码更容易记住。指令操作数也可以方便地指定：可以用名称引用寄存器（例如 R0 或 EAX），内存地址可以用十六进制编写，或者分配符号名称，就像高级语言的全局变量一样。汇编程序中的位置可以用人类可读的标签标记，跳转/分支指令引用这些标签而不是原始内存地址。

汇编语言程序由一系列指令组成，每条指令由一个助记符和零个或多个操作数组成，这些指令列在一个文本文件中，每行一条指令。一种称为汇编器的工具会读取程序源文件，并将其转换为 CPU 能够理解的数字 ML 表示形式。例如，一个汇编语言程序实现了以下一段 C 代码：

```
if (a > b)
    return a + b;
else
    return 0;
```

可以通过如下所示的汇编语言程序来实现：

```
; if (a > b)
cmp   eax, ebx   ; compare the values
jle   ReturnZero ; jump if less than or equal

; return a + b;
add   eax, ebx   ; add & store result in EAX
ret              ; (EAX is the return value)
```


For a concrete example of machine language instruction encoding on the Intel x86 ISA, see http://aturing.umcs.maine.edu/~meadow/courses/cos335/Asm07-MachineLanguage.pdf.

3.4.7.3 Assembly Language

Writing a program directly in machine language is tedious and error-prone. To make life easier for programmers, a simple text-based version of machine language was developed called *assembly language*. In assembly language, each instruction within a given CPU's ISA is given a *mnemonic*—a short English word or abbreviation that is much easier to remember than corresponding numeric opcode. Instruction operands can be specified conveniently as well: Registers can be referred to by name (e.g., `R0` or `EAX`), and memory addresses can be written in hex, or assigned symbolic names much like the global variables of higher-level languages. Locations in the assembly program can be tagged with human-readable labels, and jump/branch instructions refer to these labels rather than to raw memory addresses.

An assembly language program consists of a sequence of instructions, each comprised of a mnemonic and zero or more operands, listed in a text file with one instruction per line. A tool known as an *assembler* reads the program source file and converts it into the numeric ML representation understood by the CPU. For example, an assembly language program that implements the following snippet of C code:

```
if (a > b)
    return a + b;
else
    return 0;
```

could be implemented by an assembly language program that looks something like this:

```
; if (a > b)
cmp   eax, ebx   ; compare the values
jle   ReturnZero ; jump if less than or equal

; return a + b;
add   eax, ebx   ; add & store result in EAX
ret              ; (EAX is the return value)
```

```
ReturnZero:
    ; else return 0;
    xor   eax, eax   ; set EAX to zero
    ret              ; (EAX is the return value)
```

让我们分解一下。cmp 指令比较寄存器 EAX 和 EBX 中的值（我们假设它们包含 C 代码片段中的值 a 和 b）。接下来，jle 指令分支到标签 ReturnZero，但只有当 EAX 中的值小于或等于 EBX 时才会执行。否则会失败。

如果 EAX 大于 EBX（a > b），则执行 add 指令，该指令计算 a + b 并将结果存回 EAX，我们假设该值作为返回值。然后执行 ret 指令，控制权返回给调用函数。

如果 EAX 小于或等于 EBX（a <= b），则执行分支，并在标签 ReturnZero 之后立即继续执行。这里，我们使用了一个小技巧，通过对 EAX 进行异或运算将 EAX 设置为零。然后，我们发出 ret 指令将这个零返回给调用者。

有关英特尔汇编语言编程的更多详细信息，请参阅http://www.cs.virginia.edu/~evans/cs216/guides/x86.html。

3.4.7.4 寻址模式

像“move”（在寄存器和内存之间传输数据）这样看似简单的指令，却有很多不同的变体。我们是将一个值从一个寄存器移动到另一个寄存器吗？我们是将一个字面值（比如 5）加载到寄存器吗？我们是将一个值从内存加载到寄存器吗？还是将一个值从寄存器存储到内存？所有这些变体都称为寻址模式。我们不会在这里详细介绍所有可能的寻址模式，但以下列表应该能让您很好地了解在实际 CPU 上会遇到的各种寻址模式：

- 寄存器寻址。此模式允许将值从一个寄存器传输到另一个寄存器。在这种情况下，指令的操作数指定哪些寄存器参与操作。
- 立即数寻址。此模式允许将字面值或“立即数”加载到寄存器中。在这种情况下，操作数是目标寄存器和要加载的立即数。
- 直接寻址。此模式允许将数据移入或移出内存。
  在这种情况下，操作数指定移动的方向（到内存或从内存）和相关的内存地址。


```
ReturnZero:
    ; else return 0;
    xor   eax, eax   ; set EAX to zero
    ret              ; (EAX is the return value)
```

Let's break this down. The `cmp` instruction compares the values in registers EAX and EBX (which we're assuming contain the values `a` and `b` from the C snippet). Next, the `jle` instruction branches to the label `ReturnZero`, but it does so if and *only if* the value in EAX is less than or equal to EBX. Otherwise it falls through.

If EAX is *greater* than EBX (`a > b`), we fall through to the `add` instruction, which calculates `a + b` and stores the result back into EAX, which we're assuming serves as the return value. We issue a `ret` instruction, and control is returned to the calling function.

If EAX is less than or equal to EBX (`a <= b`), the branch is taken and we continue execution immediately after the label `ReturnZero`. Here, we use a little trick to set EAX to zero by XORing it with itself. Then we issue a `ret` instruction to return that zero to the caller.

For more details on Intel assembly language programming, see http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

3.4.7.4 Addressing Modes

A seemingly simple instruction like "move" (which transfers data between registers and memory) has many different variants. Are we moving a value from one register to another? Are we loading a literal value like 5 into a register? Are we loading a value from memory into a register? Or are we storing a value in a register out to memory? All of these variants are called *addressing modes*. We won't go over all the possible addressing modes here, but the following list should give you a good feel for the kinds of addressing modes you will encounter on a real CPU:

- *Register addressing*. This mode allows values to be transferred from one register to another. In this case, the operands of the instruction specify which registers are involved in the operation.
- *Immediate addressing*. This mode allows a literal or "immediate" value to be loaded into a register. In this case, the operands are the target register and the immediate value to be loaded.
- *Direct addressing*. This mode allows data to be moved to or from memory. In this case, the operands specify the direction of the move (to or from memory) and the memory address in question.

• 寄存器间接寻址。在此模式下，目标内存地址取自寄存器，而不是编码为指令操作数中的字面值。C 和 C++ 等语言中指针解引用正是通过这种方式实现的。指针的值（地址）被加载到寄存器中，然后使用寄存器间接“移动”指令来解引用该指针，并将其指向的值加载到目标寄存器中，或将源寄存器中的值存储到该内存地址中。

• 相对寻址。在此模式下，目标内存地址被指定为操作数，存储在指定寄存器中的值则作为相对于该目标内存地址的偏移量。这就是 C 或 C++ 等语言中实现索引数组访问的方式。

• 其他寻址模式。还有许多其他变体和组合，其中一些是几乎所有 CPU 所共有的，而另一些则是特定于某些 CPU 的。

3.4.7.5 汇编语言的进一步阅读

在前面的章节中，我们只是对汇编语言进行了简单的了解。有关 x86 汇编编程的浅显易懂的描述，请参阅 http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html。有关调用约定和 ABI 的更多信息，请参阅 https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames。

3.5 内存架构

在简单的冯 · 诺依曼计算机架构中，内存被视为单个同质块，所有块都可供 CPU 平等访问。实际上，计算机内存的架构几乎从来不会如此简单。首先，CPU 的寄存器也是一种内存，但在汇编语言程序中，寄存器通常通过名称引用，而不是像普通 ROM 或 RAM 那样进行寻址。此外，即使是“常规”内存通常也会被分成具有不同特性和不同用途的块。这种划分出于多种原因，包括成本管理和整体系统性能优化。在本节中，我们将研究当今个人电脑和游戏机中常见的一些内存架构，并探讨它们采用这种架构的一些关键原因。

- *Register indirect addressing*. In this mode, the target memory address is taken from a register, rather than being encoded as a literal value in the operands of the instruction. This is how pointer dereferences are implemented in languages like C and C++. The value of the pointer (an address) is loaded into a register, and then a register-indirect “move” instruction is used to dereference the pointer and either load the value it points to into a target register, or store a value in a source register into that memory address.

- *Relative addressing*. In this mode, the target memory address is specified as an operand, and the value stored in a specified register is used as an offset from that target memory address. This is how indexed array accesses are implemented in a language like C or C++.

- *Other addressing modes*. There are numerous other variations and combinations, some of which are common to virtually all CPUs, and some of which are specific to certain CPUs.

3.4.7.5 Further Reading on Assembly Language

In the preceding sections, we've had just a tiny taste of assembly language. For an easy-to-digest description of x86 assembly programming, see http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html. For more on calling conventions and ABIs, see https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames.

3.5 Memory Architectures

In a simple von Neumann computer architecture, memory is treated as a single homogeneous block, all of which is equally accessible to the CPU. In reality, a computer's memory is almost never architected in such a simplistic manner. For one thing, the CPU's registers are a form of memory, but registers are typically referenced by name in an assembly language program, rather than being addressed like ordinary ROM or RAM. Moreover, even “regular” memory is typically segregated into blocks with different characteristics and different purposes. This segregation is done for a variety of reasons, including cost management and optimization of overall system performance. In this section, we'll investigate some of the memory architectures commonly found in today's personal computers and gaming consoles, and explore some of the key reasons why they are architected the way they are.

3.5.1 内存映射

n 位地址总线使 CPU 可以访问理论上大小为 2n 字节的地址空间。单个存储设备（ROM 或 RAM）始终作为连续的存储单元块进行寻址。因此，计算机的地址空间通常分为多个连续的段。这些段中的一个或多个对应于 ROM 内存模块，其他段对应于 RAM 模块。例如，在 Apple II 上，0xC100 到 0xFFFF 范围内的 16 位地址分配给 ROM 芯片（包含计算机的固件），而 0x0000 到 0xBFFF 范围内的地址分配给 RAM。每当将物理存储设备分配给计算机地址空间中的某个地址范围时，我们就说该地址范围已映射到存储设备。

当然，计算机安装的内存可能远小于其地址总线理论上所能寻址的内存量。64 位地址总线可以访问 16 EiB 的内存，因此我们不太可能完全填满这样的地址空间！（即使是惠普的原型超级计算机“The Machine”的物理内存容量也只有 160 TiB，远不及这个数字。）因此，计算机地址空间中某些部分未分配的情况很常见。

3.5.1.1 内存映射 I/O

地址范围不必全部映射到内存设备——地址范围也可能映射到其他外围设备，例如游戏手柄或网络接口卡 (NIC)。这种方法称为内存映射 I/O，因为 CPU 可以通过读取或写入地址对外围设备执行 I/O 操作，就像它们是普通 RAM 一样。在底层，特殊电路会检测到 CPU 正在读取或写入已映射到非内存设备的地址范围，并将读取或写入请求转换为针对相关设备的 I/O 操作。举一个具体的例子，A□□pple II 将 I/O 设备映射到 0xC000 到 0xC0FF 的地址范围，允许程序执行诸如控制组切换 RAM、读取和控制主板上游戏控制器插座引脚上的电压，以及仅通过读取和写入此范围内的地址来执行其他 I/O 操作。

或者，CPU 可以通过称为端口的特殊寄存器与非内存设备通信。在这种情况下，每当 CPU 请求从端口寄存器读取或写入数据时，硬件都会将该请求转换为目标设备上的 I/O 操作。这种方法称为端口映射 I/O。在 Arduino 系列微控制器中，端口映射 I/O 提供了

3.5.1 Memory Mapping

An n-bit address bus gives the CPU access to a theoretical *address space* that is 2^n bytes in size. An individual memory device (ROM or RAM) is always addressed as a contiguous block of memory cells. So a computer's address space is typically divided into various contiguous segments. One or more of these segments correspond to ROM memory modules, and others correspond to RAM modules. For example on the Apple II, the 16-bit addresses in the range 0xC100 through 0xFFFF were assigned to ROM chips (containing the computer's firmware), while the addresses from 0x0000 through 0xBFFF were assigned to RAM. Whenever a physical memory device is assigned to a range of addresses in a computer's address space, we say that the address range has been *mapped* to the memory device.

Of course a computer may not have as much memory installed as could be theoretically addressed by its address bus. A 64-bit address bus can access 16 EiB of memory, so we'd be unlikely to ever fully populate such an address space! (At 160 TiB, even HP's prototype supercomputer called "The Machine" isn't anywhere close to that amount of physical memory.) Therefore, it's commonplace for some segments of a computer's address space to remain unassigned.

3.5.1.1 Memory-Mapped I/O

Address ranges needn't all map to memory devices—an address range might also be mapped to other *peripheral devices*, such as a joypad or a network interace card (NIC). This approach is called *memory-mapped I/O* because the CPU can perform I/O operations on a peripheral device by reading or writing to addresses, just as if they were oridinary RAM. Under the hood, special circuitry detects that the CPU is reading from or writing to a range of addresses that have been mapped to a non-memory device, and converts the read or write request into an I/O operation directed at the device in question. As a concrete example, the Apple II mapped I/O devices into the address range 0xC000 through 0xC0FF, allowing programs to do things like control bank-switched RAM, read and control voltages on the pins of a game controller socket on the motherboard, and perform other I/O operations by merely reading from and writing to the addresses in this range.

Alternatively, a CPU might communicate with non-memory devices via special registers known as *ports*. In this case, whenever the CPU requests that data be read from or written to a port register, the hardware converts the request into an I/O operation on the target device. This approach is called *port-mapped I/O*. In the Arduino line of microcontrollers, port-mapped I/O gives a

程序直接控制芯片某些引脚的数字输入和输出。

3.5.1.2 视频 RAM

基于光栅的显示设备通常读取一段硬连接的物理内存地址，以确定屏幕上每个像素的亮度/颜色。同样，早期基于字符的显示器会通过从内存块中读取 ASCII 码来确定在屏幕上每个位置显示哪个字符。分配给视频控制器使用的一段内存地址称为视频 RAM (VRAM)。

在早期的计算机（例如 Apple II 和 IBM PC）中，视频 RAM 通常与主板上的内存芯片相对应，并且 CPU 可以像读取和写入任何其他内存位置一样读取和写入 VRAM 中的内存地址。在 PlayStation 4 和 Xbox One 等游戏主机上也是如此，CPU 和 GPU 共享访问单个大型统一内存块。

在个人电脑中，GPU 通常位于单独的电路板上，插入主板上的扩展槽。显存 (RAM) 通常位于显卡上，以便 GPU 能够尽快访问。PCI、AGP 或 PCI Express (PCIe) 等总线协议用于通过扩展槽总线在“主 RAM”和显存 (VRAM) 之间来回传输数据。主 RAM 和显存之间的这种物理分离可能会造成严重的性能瓶颈，也是导致渲染引擎和图形 API（如 OpenGL 和 DirectX 11）复杂性的主要原因之一。

3.5.1.3 案例研究：Apple II 内存映射

为了说明内存映射的概念，我们来看一个简单的实际示例。Apple II 具有 16 位地址总线，这意味着其地址空间大小仅为 64 KiB。该地址空间映射到 ROM、RAM、内存映射的 I/O 设备和视频 RAM 区域，如下所示：

```
0xC100 - 0xFFFF  ROM (Firmware)
0xC000 - 0xC0FF  Memory-Mapped I/O
0x6000 - 0xBFFF  General-purpose RAM
0x4000 - 0x5FFF  High-res video RAM (page 2)
0x2000 - 0x3FFF  High-res video RAM (page 1)
0x0C00 - 0x1FFF  General-purpose RAM
0x0800 - 0x0BFF  Text/lo-res video RAM (page 2)
0x0400 - 0x07FF  Text/lo-res video RAM (page 1)
```


program direct control over the digital inputs and outputs at some of the pins of the chip.

3.5.1.2 Video RAM

Raster-based display devices typically read a hard-wired range of physical memory addresses in order to determine the brightness/color of each pixel on the screen. Likewise, early charcter-based displays would determine which character glyph to display at each location on the screen by reading ASCII codes from a block of memory. A range of memory addresses assigned for use by a video controller is known as *video RAM* (VRAM).

In early computers like the Apple II and early IBM PCs, video RAM used to correspond to memory chips on the motherboard, and memory addresses in VRAM could be read from and written to by the CPU in exactly the same way as any other memory location. This is also the case on game consoles like the PlayStation 4 and the Xbox One, where both the CPU and GPU share access to a single, large block of unified memory.

In personal computers, the GPU often lives on a separate circuit board, plugged into an expansion slot on the motherboard. Video RAM is typically located on the video card so that the GPU can access it as quickly as possible. A bus protocol such as PCI, AGP or PCI Express (PCIe) is used to transfer data back and forth between “main RAM” and VRAM, via the expansion slot’s bus. This physical separation between main RAM and VRAM can be a significant performance bottleneck, and is one of the primary contributors to the complexity of rendering engines and graphics APIs like OpenGL and DirectX 11.

3.5.1.3 Case Study: The Apple II Memory Map

To illustrate the concepts of memory mapping, let’s look at a simple real-world example. The Apple II had a 16-bit address bus, meaning that its address space was only 64 KiB in size. This address space was mapped to ROM, RAM, memory-mapped I/O devices and video RAM regions as follows:

```
0xC100 - 0xFFFF  ROM (Firmware)
0xC000 - 0xC0FF  Memory-Mapped I/O
0x6000 - 0xBFFF  General-purpose RAM
0x4000 - 0x5FFF  High-res video RAM (page 2)
0x2000 - 0x3FFF  High-res video RAM (page 1)
0x0C00 - 0x1FFF  General-purpose RAM
0x0800 - 0x0BFF  Text/lo-res video RAM (page 2)
0x0400 - 0x07FF  Text/lo-res video RAM (page 1)
```

```
0x0200 - 0x03FF  General-purpose and reserved RAM
0x0100 - 0x01FF  Program stack
0x0000 - 0x00FF  Zero page (mostly reserved for DOS)
```

需要注意的是，Apple II 内存映射中的地址直接对应于主板上的内存芯片。在当今的操作系统中，程序是基于虚拟地址而不是物理地址运行的。

我们将在下一节探讨虚拟内存。

3.5.2 虚拟内存

大多数现代 CPU 和操作系统都支持内存重新映射功能，即虚拟内存系统。在这些系统中，程序使用的内存地址并不直接映射到计算机中安装的内存模块。相反，每当程序读取或写入某个地址时，该地址首先由 CPU 通过操作系统维护的查找表重新映射。重新映射的地址最终可能指向内存中的实际单元（具有完全不同的数字地址）。它也可能指向磁盘上的数据块。或者它可能根本没有映射到任何物理存储。在虚拟内存系统中，程序使用的地址称为虚拟地址，而内存控制器为了访问 RAM 或 ROM 模块而通过地址总线实际传输的位模式称为物理地址。

虚拟内存是一个强大的概念。它允许程序使用比计算机实际安装的内存更多的内存，因为数据可能会从物理 RAM 溢出到磁盘。虚拟内存还可以提高操作系统的稳定性和安全性，因为每个程序都有自己独立的内存“视图”，可以防止其侵占其他程序或操作系统本身拥有的内存块。我们将在 4.4.5 节中详细讨论操作系统如何管理正在运行的程序的虚拟内存空间。

3.5.2.1 虚拟内存页面

为了理解这种重新映射是如何进行的，我们需要假设整个可寻址内存空间（如果地址总线为 n 位宽，则为 2n 个字节大小的单元）在概念上被划分为大小相等的连续块，这些块被称为页面。页面大小因操作系统而异，但始终是 2 的幂次方——典型的页面大小为 4 KiB 或 8 KiB。假设页面大小为 4 KiB，则 32 位地址空间将被划分为 1,048,576 个不同的页面，编号从 0x0 到 0xFFFFF，如表 3.2 所示。


```
0x0200 - 0x03FF  General-purpose and reserved RAM
0x0100 - 0x01FF  Program stack
0x0000 - 0x00FF  Zero page (mostly reserved for DOS)
```

We should note that the addresses in the Apple II memory map corresponded directly to memory chips on the motherboard. In today's operating systems, programs work in terms of *virtual* addresses rather than *physical* addresses. We'll explore virtual memory in the next section.

3.5.2 Virtual Memory

Most modern CPUs and operating systems support a memory remapping feature known as a *virtual memory system*. In these systems, the memory addresses used by a program don't map directly to the memory modules installed in the computer. Instead, whenever a program reads from or writes to an address, that address is first *remapped* by the CPU via a look-up table that's maintained by the OS. The remapped address might end up referring to an actual cell in memory (with a totally different numerical address). It might also end up referring to a block of data on-disk. Or it might turn out not to be mapped to any physical storage at all. In a virtual memory system, the addresses used by programs are called *virtual addresses*, and the bit patterns that are actually transmitted over the address bus by the memory controller in order to access a RAM or ROM module are called *physical addresses*.

Virtual memory is a powerful concept. It allows programs to make use of more memory than is actually installed in the computer, because data can overflow from physical RAM onto disk. Virtual memory also improves the stability and security of the operating system, because each program has its own private "view" of memory and is prevented from stomping on memory blocks that are owned by other programs or the operating system itself. We'll talk more about how the operating system manages the virtual memory spaces of running programs in Section 4.4.5.

3.5.2.1 Virtual Memory Pages

To understand how this remapping takes place, we need to imagine that the entire addressable memory space (that's 2^n byte-sized cells if the address bus is n bits wide) is conceptually divided into equally-sized contiguous chunks known as *pages*. Page sizes differ from OS to OS, but are always a power of two—a typical page size is 4 KiB or 8 KiB. Assuming a 4 KiB page size, a 32-bit address space would be divided up into 1,048,576 distinct pages, numbered from 0x0 to 0xFFFFF, as shown in Table 3.2.

发件人地址	收件人地址	页面索引
0x00000000	0x00000FFF	页面 0x0
0x00001000	0x00001FFF	页面 0x1
0x00002000	0x00002FFF	页面 0x2
...		
0x7FFF2000	0x7FFF2FFF	页面 0x7FFF2
0x7FFF3000	0x7FFF3FFF	页面 0x7FFF3
...		
0xFFFFE000	0xFFFFEFFF	页面 0xFFFFE
0xFFFFF000	0xFFFFFFFF	页面 0xFFFFF

表 3.2. 将 32 位地址空间划分为 4 KiB 页面。

虚拟地址和物理地址之间的映射总是以页为单位。图 3.22 展示了虚拟地址和物理地址之间的一个假设映射。

3.5.2.2 虚拟地址到物理地址的转换

每当 CPU 检测到内存读写操作时，地址就会被拆分成两部分：页面索引和页面内的偏移量（以字节为单位）。对于 4 KiB 的页面大小，偏移量就是地址的低 12 位，而页面索引则是地址的高 20 位（经过屏蔽后右移 12 位）。例如，虚拟地址 0x1A7C6310 对应的偏移量为 0x310，页面索引为 0x1A7C6。

然后，CPU 的内存管理单元 (MMU) 会在页表中查找页面索引，该页表将虚拟页面索引映射到物理页面索引。（页表存储在 RAM 中，由操作系统管理。）如果相关页面恰好映射到物理内存页面，则虚拟页面索引会转换为相应的物理页面索引，该物理页面索引的位会根据需要左移，并与原始页面偏移量的位进行“或”运算，这样就得到了物理地址。继续我们的例子，如果虚拟页面 0x1A7C6 恰好映射到物理页面 0x73BB9，那么转换后的物理地址最终将为 0x73BB9310。这就是实际通过地址总线传输的地址。图 3.23 说明了 MMU 的运行方式。

如果页表指示某个页面未映射到物理 RAM（因为它从未被分配，或者因为该页面已被

From Address	To Address	Page Index
0x00000000	0x00000FFF	Page 0x0
0x00001000	0x00001FFF	Page 0x1
0x00002000	0x00002FFF	Page 0x2
...		
0x7FFF2000	0x7FFF2FFF	Page 0x7FFF2
0x7FFF3000	0x7FFF3FFF	Page 0x7FFF3
...		
0xFFFFE000	0xFFFFEFFF	Page 0xFFFFE
0xFFFFF000	0xFFFFFFFF	Page 0xFFFFF

Table 3.2. Division of a 32-bit address space into 4 KiB pages.

The mapping between virtual addresses and physical addresses is always done at the granularity of pages. One hypothetical mapping between virtual and physical addresses is illustrated in Figure 3.22.

3.5.2.2 Virtual to Physical Address Translation

Whenever the CPU detects a memory read or write operation, the address is split into two parts: the *page index* and an *offset* within that page (measured in bytes). For a page size of 4 KiB, the offset is just the lower 12 bits of the address, and the page index is the upper 20 bits, masked off and shifted to the right by 12 bits. For example, the virtual address 0x1A7C6310 corresponds to an offset of 0x310 and a page index of 0x1A7C6.

The page index is then looked up by the CPU's *memory management unit* (MMU) in a *page table* that maps virtual page indices to physical ones. (The page table is stored in RAM and is managed by the operating system.) If the page in question happens to be mapped to a physical memory page, the virtual page index is translated into the corresponding physical page index, the bits of this physical page index are shifted to the left as appropriate and ORed together with the bits of the original page offset, and voila! We have a physical address. So to continue our example, if virtual page 0x1A7C6 happens to map to physical page 0x73BB9, then the translated physical address would end up being 0x73BB9310. This is the address that would actually be transmitted over the address bus. Figure 3.23 illustrates the operation of the MMU.

If the page table indicates that a page is not mapped to physical RAM (either because it was never allocated, or because that page has since been

图 3.22 虚拟内存空间中的页面大小地址范围被重新映射到物理内存页面、磁盘上的交换文件，或者它们可能保持未映射状态。

当数据被换出到磁盘文件时，MMU 会触发一个中断，告诉操作系统内存请求无法满足。这被称为页面错误。（有关中断的更多信息，请参阅 4.4.2 节。）

3.5.2.3 处理页面错误

对于访问未分配的页面，操作系统通常会通过使程序崩溃并生成核心转储来响应页面错误。对于访问已换出到磁盘的页面，操作系统会暂时挂起当前正在运行的程序，将页面从交换文件读取到 RAM 的物理页面中，然后像往常一样将虚拟地址转换为物理地址。最后，它将控制权交还给被挂起的程序。从程序的角度来看，此操作完全无缝——它永远“不知道”页面是已经在内存中还是需要从磁盘换入。

通常情况下，只有当内存系统负载较高、物理页面供应不足时，才会将页面换出到磁盘。操作系统会尝试仅换出最不常用的内存页面，以避免程序在内存页面和磁盘页面之间不断“切换”。

Figure 3.22. Page-sized address ranges in a virtual memory space are remapped either to physical memory pages, a swap file on disk, or they may remain unmapped.

swapped out to a disk file), the MMU raises an *interrupt*, which tells the operating system that the memory request can't be fulfilled. This is called a *page fault*. (See Section 4.4.2 for more on interrupts.)

3.5.2.3 Handling Page Faults

For accesses to unallocated pages, the OS normally responds to a page fault by crashing the program and generating a core dump. For accesses to pages that have been swapped out to disk, the OS temporarily suspends the currently-running program, reads the page from the swap file into a physical page of RAM, and then translates the virtual address to a physical address as usual. Finally it returns control back to the suspended program. From the point of view of the program, this operation is entirely seamless—it never "knows" whether a page was already in memory or had to be swapped in from disk.

Normally pages are swapped out to disk only when the load on the memory system is high, and physical pages are in short supply. The OS tries to swap out only the least-frequently-used pages of memory to avoid programs continually "thrashing" between memory-based and disk-based pages.

图 3.23。MMU 拦截内存读取操作，并将虚拟地址分解为虚拟页索引和偏移量。虚拟页索引通过页表转换为物理页索引，并根据物理页索引和原始偏移量构建物理地址。最后，使用重新映射的物理地址执行指令。

3.5.2.4 转换后备缓冲区（TLB）

由于页面大小相对于可寻址内存的总大小（通常为 4 KiB 或 8 KiB）而言通常较小，因此页表可能会变得非常大。如果程序每次访问内存时都必须扫描整个页表，那么在页表中查找物理地址将会非常耗时。

为了加快访问速度，我们使用了一种缓存机制，其假设是，普通程序倾向于在相对较少的页面内重复使用地址，而不是在整个地址范围内随机读写。CPU 芯片上的 MMU 中维护着一个称为转换后备缓冲区 (TLB) 的小表，其中缓存了最近使用的地址的虚拟地址到物理地址的映射。由于这个缓冲区的位置靠近 MMU，因此访问速度非常快。

TLB 的作用与通用内存缓存层级很相似，只不过它仅用于缓存页表条目。有关缓存层级的工作原理，请参见第 3.5.4 节。

Figure 3.23. The MMU intercepts a memory read operation, and breaks the virtual address into a virtual page index and an offset. The virtual page index is converted to a physical page index via the page table, and the physical address is constructed from the physical page index and the original offset. Finally, the instruction is executed using the remapped physical address.

3.5.2.4 The Translation Lookaside Buffer (TLB)

Because page sizes tend to be small relative to the total size of addressable memory (typically 4 KiB or 8 KiB), the page table can become very large. Looking up physical addresses in the page table would be time-consuming if the entire table had to be scanned for every memory access a program makes.

To speed up access, a caching mechanism is used, based on the assumption that an average program will tend to reuse addresses within a relatively small number of pages, rather than read and write randomly across the entire address range. A small table known as the *translation lookaside buffer* (TLB) is maintained within the MMU on the CPU die, in which the virtual-to-physical address mappings of the most recently-used addresses are cached. Because this buffer is located in close proximity to the MMU, accessing it is very fast.

The TLB acts a lot like a general-purpose *memory cache hierarchy*, except that it is only used for caching page table entries. See Section 3.5.4 for a discussion of how cache hierarchies work.

3.5.2.5 关于虚拟内存的进一步阅读

请参阅 https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/virtual. html 和 https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swapping 中关于虚拟内存实现细节的两个很好的讨论。

Ulrich Drepper 的这篇论文《每个程序员都应该知道的内存知识》也是所有程序员必读的文章：https://www.akkadia.org/drepper/cpumemory.pdf。

3.5.3 减少延迟的内存架构

从存储设备访问数据的速度是一项重要特性。我们经常说内存访问延迟，它是指从 CPU 向内存系统请求数据到 CPU 实际接收到数据之间的时间间隔。

内存访问延迟主要取决于三个因素：

1. 用于实现单个存储单元的技术，
2. 内存支持的读取和/或写入端口的数量，以及
3. 这些内存单元与使用它们的 CPU 核心之间的物理距离。

静态 RAM (SRAM) 的访问延迟通常远低于动态 RAM (DRAM)。SRAM 采用更复杂的设计来实现更低的延迟，这使得每比特内存所需的晶体管数量比 DRAM 要多。这反过来又使得 SRAM 比 DRAM 更昂贵，无论是在每比特的财务成本方面，还是在每比特在芯片上占用的空间方面。

最简单的存储单元只有一个端口，这意味着它在任何给定时间只能执行一次读取或写入操作。多端口 RAM 允许同时执行多个读取和/或写入操作，从而减少多个核心或单个核心内的多个组件同时尝试访问一个内存组时因争用而导致的延迟。正如您所料，多端口 RAM 每位所需的晶体管数量比单端口设计更多，因此其成本更高，占用的芯片空间也比单端口内存更大。

CPU 和 RAM 组之间的物理距离也会影响该内存的访问延迟。这是因为电子信号在计算机内部的传播速度是有限的。理论上，电子

3.5.2.5 Further Reading on Virtual Memory

See https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/virtual.html and https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swapping for two good discussions of virtual memory implementation details.

This paper by Ulrich Drepper entitled, “What Every Programmer Should Know About Memory” is also a must-read for all programmers: https://www.akkadia.org/drepper/cpumemory.pdf.

3.5.3 Memory Architectures for Latency Reduction

The speed with which data can be accessed from a memory device is an important characteristic. We often speak of *memory access latency*, which is defined as the length of time between the moment the CPU requests data from the memory system and the moment that that data is actually received by the CPU. Memory access latency is primarily dependent on three factors:

1. the technology used to implement the individual memory cells,
2. the number of read and/or write ports supported by the memory, and
3. the physical distance between those memory cells and the CPU core that uses them.

The access latency of *static RAM* (SRAM) is generally much lower than that of *dynamic RAM* (DRAM). SRAM achieves its lower latency by utilizing a more complex design which requires more transistors per bit of memory than does DRAM. This in turn makes SRAM more expensive than DRAM, both in terms of financial cost per bit, and in terms of the amount of real estate per bit that it consumes on the die.

The simplest memory cell has a single *port*, meaing only one read or write operation can be performed by it at any given time. *Multi-ported* RAM allows multiple read and/or write operations to be performed simultaneously, thereby reducing the latency caused by contention when multiple cores, or multiple components within a single core, attempt to access a bank of memory simultaneously. As you’d expect, a multi-ported RAM requires more transistors per bit than a single-ported design, and hence it costs more and uses more real estate on the die than a single-ported memory.

The physical distance between the CPU and a bank of RAM also plays a role in determining the access latency of that memory. This is because electronic signals travel at a finite speed within the computer. In theory, electronic

信号由电磁波组成，因此传播速度接近光速。内存访问信号在传输过程中会经过各种开关和逻辑电路，从而带来额外的延迟。因此，内存单元距离使用它的 CPU 核心越近，其访问延迟往往越低。

3.5.3.1 记忆缺口

在计算发展的早期，内存访问延迟和指令执行延迟大致相同。例如，在 Intel 8086 上，基于寄存器的指令可以在 2 到 4 个周期内执行，而主存访问也大约需要 4 个周期。然而，在过去的几十年里，CPU 的原始时钟速度和有效指令吞吐量的提升速度远远快于内存访问速度。今天，主存的访问延迟相对于执行单条指令的延迟非常高：在 Intel Core i7 上，一条基于寄存器的指令仍然需要 1 到 10 个周期才能完成，而访问主 RAM 则需要 500 个周期才能完成！CPU 速度和内存访问延迟之间这种不断扩大的差异通常被称为内存差距。图 3.24 展示了内存差距随时间不断扩大的趋势。

程序员和硬件设计师共同开发了各种各样的技术来解决高内存访问延迟带来的问题。这些技术通常侧重于以下一项或多项：

1. 通过将更小、更快的内存组放置得更靠近 CPU 核心来减少平均内存延迟，以便可以更快地访问常用数据；

2. 通过安排 CPU 在等待内存操作完成时执行其他有用的工作来“隐藏”内存访问延迟；和/或

3. 针对需要使用数据完成的工作，以尽可能高效的方式安排程序数据，从而最大限度地减少对主内存的访问。

在本节中，我们将仔细研究用于降低平均延迟的内存架构。我们将讨论另外两种技术（延迟“隐藏”和

[7] 电子信号在铜线或光纤等传输介质中的传播速度始终略低于真空中的光速。每种互连材料都有各自的特征速度因子 (VF)，范围从真空光速的不到 50% 到 99%。

signals are comprised of electromagnetic waves, and hence travel at close[7] to the speed of light. Additional latencies are introduced by the various switching and logic circuits that a memory access signal encounters on its journey through the system. As such, the closer a memory cell is to the CPU core that uses it, the lower its access latency tends to be.

3.5.3.1 The Memory Gap

In the early days of computing, memory access latency and instruction execution latency were on roughly equal footing. For example, on the Intel 8086, register-based instructions could execute in two to four cycles, and a main memory access also took roughly four cycles. However, over the past few decades both raw clock speeds and the effective instruction throughput of CPUs have been improving at a much faster rate than memory access speeds. Today, the access latency of main memory is extremely high relative to the latency of executing a single instruction: Whereas a register-based instruction still takes between one and 10 cycles to complete on an Intel Core i7, an access to main RAM can take on the order of 500 cycles to complete! This ever-increasing discrepancy between CPU speeds and memory access latencies is often called the *memory gap*. Figure 3.24 illustrates the trend of an ever-increasing memory gap over time.

Programmers and hardware designers have together developed a wide range of techniques to work around the problems caused by high memory access latency. These techniques usually focus on one or more of the following:

1. *reducing* average memory latency by placing smaller, faster memory banks closer to the CPU core, so that frequently-used data can be accessed more quickly;

2. *“hiding”* memory access latency by arranging for the CPU to do other useful work while waiting for a memory operation to complete; and/or

3. *minimizing* accesses to main memory by arranging a program’s data in as efficient a manner as possible with respect to the work that needs to be done with that data.

In this section, we’ll have a closer look at memory architectures for average latency reduction. We’ll discuss the other two techniques (latency “hiding” and

[7] The speed that an electronic signal travels within a transmission medium such as copper wire or optical fiber is always somewhat lower than the speed of light in a vaccuum. Each interconnect material has its own characteristic *velocity factor* (VF) ranging from less than 50% to 99% the speed of light in a vaccuum.

通过合理的数据布局来最小化内存访问），同时在第 4 章中研究并行硬件设计和并发编程技术。

3.5.3.2 寄存器文件

CPU 的寄存器文件或许是旨在最大程度降低访问延迟的内存架构的最极端示例。寄存器通常使用多端口静态 RAM (SRAM) 实现，通常具有用于读写操作的专用端口，从而允许这些操作并行而非串行进行。此外，寄存器文件通常位于紧邻使用它的 ALU 电路的位置。此外，ALU 几乎直接访问寄存器，而访问主 RAM 通常必须经过虚拟地址转换系统、内存缓存层次结构和缓存一致性协议（参见第 3.5.4 节）、地址和数据总线，甚至可能还要经过交叉开关逻辑。这些事实解释了为什么寄存器的访问速度如此之快，以及为什么寄存器 RAM 的成本相对于通用 RAM 而言相对较高。这种较高的成本是合理的，因为寄存器是迄今为止任何计算机中使用最频繁的内存，并且寄存器文件的总大小与主 RAM 的大小相比非常小。

图 3.24。CPU 性能与内存性能之间不断扩大的差异被称为“内存差距”。（改编自 [23] John L. Hennessey 和 David A. Patterson 合著的《计算机体系结构：定量方法》）

the minimization of memory accesses via judicious data layout) while investigating parallel hardware design and concurrent programming techniques in Chapter 4.

3.5.3.2 Register Files

A CPU's register file is perhaps the most extreme example of a memory architecture designed to minimize access latency. Registers are typically implemented using multi-ported static RAM (SRAM), usually with dedicated ports for read and write operations, allowing these operations to occur in parallel rather than serially. What's more, the register file is typically located immediately adjacent to the circuitry for the ALU that uses it. Furthermore, registers are accessed pretty much directly by the ALU, whereas accesses to main RAM typically have to pass through the virtual address translation system, the memory cache hierarchy and cache coherence protocol (see Section 3.5.4), the address and data buses, and possibly also through crossbar switching logic. These facts explain why registers can be accessed so quickly, and also why the cost of register RAM is relatively high as compared to general-purpose RAM. This higher cost is justified because registers are by far the most frequently-used memory in any computer, and because the total size of the register file is very small when compared to the size of main RAM.

Figure 3.24. The ever-increasing difference between CPU performance and the performance of memory is called the *memory gap*. (Adapted from [23] *Computer Architecture: A Quantitative Approach* by John L. Hennessey and David A. Patterson.)

3.5.4 内存缓存层次结构

内存缓存层次结构是缓解当今个人电脑和游戏机中高内存访问延迟影响的主要机制之一。在缓存层次结构中，一块体积小但速度快的 RAM 被称为一级 (L1) 缓存，它被放置在非常靠近 CPU 核心的位置（位于同一芯片上）。由于它非常靠近 CPU 核心，因此它的访问延迟几乎与 CPU 的寄存器文件一样低。某些系统提供更大但速度稍慢的二级 (L2) 缓存，它位于距离核心较远的位置（通常也位于片上，并且通常在多核 CPU 上由两个或多个核心共享）。某些机器甚至具有更大但距离更远的 L3 或 L4 缓存。这些缓存协同工作，自动保留最常用数据的副本，从而将对位于系统主板上的非常大但速度非常慢的主 RAM 组的访问保持在最低限度。

缓存系统通过在缓存中保存程序最常访问的数据块的本地副本来提高内存访问性能。如果 CPU 请求的数据已经在缓存中，则可以非常快速地（大约几十个周期）将其提供给 CPU。这称为缓存命中。如果数据尚未存在于缓存中，则必须从主 RAM 中将其提取到缓存中。这称为缓存未命中。从主 RAM 读取数据可能需要数百个周期，因此缓存未命中的代价确实非常高昂。

3.5.4.1 缓存行

内存缓存利用了实际软件中的内存访问模式倾向于表现出两种引用局部性的事实：

1. 空间局部性。如果程序访问内存地址 N，则很可能也会访问附近的地址，例如 N + 1、N + 2 等等。顺序迭代存储在数组中的数据就是具有高度空间局部性的内存访问模式的一个例子。

2. 时间局部性。如果程序访问了内存地址 N，那么很可能在不久的将来会再次访问该地址。从变量或数据结构中读取数据，对其进行转换，然后将更新的结果写入同一变量或数据结构，这就是具有高度时间局部性的内存访问模式的一个例子。

为了利用引用局部性，内存缓存系统将数据以连续的块（称为缓存行）的形式移动到缓存中，而不是单独缓存数据项。

3.5.4 Memory Cache Hierarchies

A *memory cache hierarchy* is one of the primary mechanisms for mitigating the impacts of high memory access latencies in today's personal computers and game consoles. In a cache hierarchy, a small but fast bank of RAM called the *level 1* (L1) cache is placed very near to the CPU core (on the same die). Its access latency is almost as low as the CPU's register file, because it is so close to the CPU core. Some systems provide a larger but somewhat slower *level 2* (L2) cache, located further away from the core (usually also on-chip, and often shared between two or more cores on a multicore CPU). Some machines even have larger but more-distant L3 or L4 caches. These caches work in concert to automatically retain copies of the most frequently-used data, so that accesses to the very large but very slow bank of main RAM located on the system motherboard are kept to a minimum.

A caching system improves memory access performance by keeping *local copies* in the cache of those chunks of data that are most frequently accessed by the program. If the data requested by the CPU is already in the cache, it can be provided to the CPU very quickly—on the order of tens of cycles. This is called a *cache hit*. If the data is not already present in the cache, it must be fetched into the cache from main RAM. This is called a *cache miss*. Reading data from main RAM can take hundreds of cycles, so the cost of a cache miss is very high indeed.

3.5.4.1 Cache Lines

Memory caching takes advantage of the fact that memory access patterns in real software tend to exhibit two kinds of *locality of reference:*

1. *Spatial locality*. If memory address N is accessed by a program, it's likely that nearby addresses such as $N+1$, $N+2$ and so on will also be accessed. Iterating sequentially through the data stored in an array is an example of a memory access pattern with a high degree of spatial locality.

2. *Temporal locality*. If memory address N is accessed by a program, it's likely that that same address will be accessed again in the near future. Reading data from a variable or data structure, performing a transformation on it, and then writing an updated result into the same variable or data structure is an example of a memory access pattern with a high degree of temporal locality.

To take advantage of locality of reference, memory caching systems move data into the cache in contiguous blocks called *cache lines* rather than caching data items individually.

例如，假设程序正在访问某个类或结构体实例的数据成员。读取第一个成员时，内存控制器可能需要数百个周期才能访问主 RAM 并检索数据。然而，缓存控制器并非只读取这一个成员，它实际上会将一个更大的连续 RAM 块读入缓存。这样，后续读取该实例的其他数据成员就很可能获得低成本的缓存命中。

3.5.4.2 将缓存行映射到主 RAM 地址

缓存中的内存地址与主 RAM 中的内存地址之间存在简单的一对多对应关系。我们可以将缓存的地址空间视为以重复模式“映射”到主 RAM 地址空间，从主 RAM 中的地址 0 开始，一直向上，直到所有主 RAM 地址都被缓存“覆盖”。

举一个具体的例子，假设我们的缓存大小为 32 KiB，每条缓存行 128 字节。因此，缓存可以容纳 256 条缓存行（256 × 128 = 32,768 B = 32 KiB）。让我们进一步假设主 RAM 的大小为 256 MiB。因此主 RAM 是缓存的 8192 倍，因为（256 × 1024）/32 = 8192 。这意味着我们需要将缓存的地址空间覆盖到主 RAM 地址空间上 8192 次，以覆盖所有可能的物理内存位置。或者换句话说，缓存中的一行映射到主 RAM 的 8192 个不同的行大小块。

给定主 RAM 中的任意地址，我们可以通过用主 RAM 地址对缓存大小取模来找到其在缓存中的地址。因此，对于 32 KiB 的缓存和 256 MiB 的主 RAM，缓存地址 0x0000 到 0x7FFF（即 32 KiB）映射到主 RAM 地址 0x0000 到 0x7FFF。但这个缓存地址范围也映射到主 RAM 地址 0x8000 到

0xFFFF、0x10000 至 0x17FFF、0x18000 至 0x1FFFF 等等，一直到地址 0xFFF8000 至

0xFFFFFFF。图 3.25 说明了主 RAM 和缓存 RAM 之间的映射。

3.5.4.3 寻址缓存

让我们考虑一下当 CPU 从内存中读取单个字节时会发生什么。首先，主 RAM 中所需字节的地址被转换为缓存中的地址。然后，缓存控制器检查包含该字节的缓存行是否已存在于缓存中。如果存在，则为缓存命中，并从缓存中读取该字节，而不是从主 RAM 中读取。如果没有，则为缓存未命中，并从主 RAM 中读取行大小的数据块，并

For example, let's say the program is accessing the data members of an instance of a class or struct. When the first member is read, it might take hundreds of cycles for the memory controller to reach out into main RAM to retrieve the data. However, the cache controller doesn't just read that one member—it actually reads a larger contiguous block of RAM into the cache. That way, subsequent reads of the other data members of the instance will likely result in low-cost cache hits.

3.5.4.2 Mapping Cache Lines to Main RAM Addresses

There is a simple one-to-many correspondence between memory addresses in the cache and memory addresses in main RAM. We can think of the address space of the cache as being "mapped" onto the main RAM address space in a repeating pattern, starting at address 0 in main RAM and continuing on up until all main RAM addresses have been "covered" by the cache.

As a concrete example, let's say that our cache is 32 KiB in size, and that cache lines are 128 bytes each. The cache can therefore hold 256 cache lines ($256 \times 128 = 32{,}768$ B = 32 KiB). Let's further assume that main RAM is 256 MiB in size. So main RAM is 8192 times as big as the cache, because $(256 \times 1024)/32 = 8192$. That means we need to overlay the address space of the cache onto the main RAM address space 8192 times in order to cover all possible physical memory locations. Or put another way, a single line in the cache maps to 8192 distinct line-sized chunks of main RAM.

Given any address in main RAM, we can find its address in the cache by taking the main RAM address *modulo* the size of the cache. So for a 32 KiB cache and 256 MiB of main RAM, the cache addresses 0x0000 through 0x7FFF (that's 32 KiB) map to main RAM addresses 0x0000 through 0x7FFF. But this range of cache addresses *also* maps to main RAM addresses 0x8000 through 0xFFFF, 0x10000 through 0x17FFF, 0x18000 through 0x1FFFF and so on, all the way up to the last main RAM block at addresses 0xFFF8000 through 0xFFFFFFF. Figure 3.25 illustrates the mapping between main RAM and cache RAM.

3.5.4.3 Addressing the Cache

Let's consider what happens when the CPU reads a single byte from memory. The address of the desired byte in main RAM is first converted into an address within the cache. The cache controller then checks whether or not the cache line containing that byte already exists in the cache. If it does, it's a cache hit, and the byte is read from the cache rather than from main RAM. If it doesn't, it's a cache miss, and the line-sized chunk of data is read from main RAM and

图 3.25. 主内存地址和缓存行之间的直接映射。

加载到缓存中，以便后续读取附近的地址将会很快。

缓存只能处理与缓存行大小的倍数对齐的内存地址（有关内存对齐的讨论，请参见第 3.3.7.1 节）。换句话说，缓存实际上只能以行为单位寻址，而不能以字节为单位寻址。因此，我们需要将字节地址转换为缓存行索引。考虑一个总大小为 2 M 字节的缓存，包含大小为 2 n 的行。主 RAM 地址的 n 个最低有效位表示字节在缓存行内的偏移量。我们去掉这 n 个最低有效位，将单位从字节转换为行（即将地址除以缓存行大小，即 2 n ）。最后，我们将得到的地址分成两部分：（ M − n ）个最低有效位成为缓存行索引，所有剩余位告诉我们缓存行来自主 RAM 中的哪个缓存大小的块。块索引称为标签。

如果发生缓存未命中，缓存控制器会将一行大小的数据块从主 RAM 加载到缓存中的相应行中。缓存还会维护一个小型标签表，每个缓存行对应一个标签。这使得缓存系统能够跟踪缓存中每行数据来自哪个主 RAM 块。由于缓存中的内存地址与主 RAM 中的内存地址之间存在多对一的关系，因此这一点至关重要。图 3.26 说明了缓存如何将标签与其中的每行活动数据关联起来。

回到我们从主 RAM 读取一个字节的例子，完整的

Figure 3.25. Direct mapping between main memory addresses and cache lines.

loaded into the cache so that subsequent reads of nearby addresses will be fast.

The cache can only deal with memory addresses that are *aligned* to a multiple of the cache line size (see Section 3.3.7.1 for a discussion of memory alignment). Put another way, the cache can really only be addressed in units of lines, not bytes. Hence we need to convert our byte's address into a *cache line index*.

Consider a cache that is 2^M bytes in total size, containing lines that are 2^n in size. The n least-significant bits of the main RAM address represent the *offset* of the byte within the cache line. We strip off these n least-significant bits to convert from units of bytes to units of lines (i.e., we divide the address by the cache line size, which is 2^n). Finally we split the resulting address into two pieces: The $(M - n)$ least-significant bits become the *cache line index*, and all the remaining bits tell us from *which* cache-sized block in main RAM the cache line came from. The block index is known as the *tag*.

In the case of a cache miss, the cache controller loads a line-sized chunk of data from main RAM into the corresponding line within the cache. The cache also maintains a small table of tags, one for each cache line. This allows the caching system to keep track of from *which* main RAM block each line in the cache came. This is necessary because of the many-to-one relationship between memory addresses in the cache and memory addresses in main RAM. Figure 3.26 illustrates how the cache associates a tag with each active line within it.

Returning to our example of reading a byte from main RAM, the complete

图 3.26。缓存中的每一行都与一个标签相关联，指示相应行来自主 RAM 的哪个缓存大小的块。

事件顺序如下：CPU 发出读取操作。主 RAM 地址被转换为偏移量、行索引和标签。使用行索引检查缓存中的相应标签。如果缓存中的标签与请求的标签匹配，则为缓存命中。在这种情况下，行索引用于从缓存中检索行大小的数据块，并使用偏移量在行内定位所需的字节。如果标签不匹配，则为缓存未命中。在这种情况下，将主 RAM 中适当的行大小数据块读入缓存，并将相应的标签存储在缓存的标签表中。因此，后续读取附近的地址（位于同一缓存行内的地址）将导致更快的缓存命中。

3.5.4.4 集合关联和替换策略

上面描述的缓存行和主 RAM 地址之间的简单映射称为直接映射缓存。这意味着主 RAM 中的每个地址只映射到缓存中的一行。以我们的 32 KiB 缓存（包含 128 字节缓存行）为例，主 RAM 地址 0x203 映射到缓存行 4（因为 0x203 等于 515，且 □ 515/128 □ = 4 ）。但是，在我们的示例中，主 RAM 中有 8192 个唯一的缓存行大小的块，它们都映射到缓存行 4。具体来说，缓存行 4 对应于主 RAM 地址 0x200 到 0x27F，也对应于地址 0x8200 到 0x827F、0x10200 到 0x1027f，以及 8189 个其他缓存行大小的地址范围。

当发生缓存未命中时，CPU 必须将相应的缓存行从主内存加载到缓存中。如果缓存行中不包含有效的

Figure 3.26. A *tag* is associated with each line in the cache, indicating from which cache-sized block of main RAM the corresponding line came.

sequence of events is as follows: The CPU issues a read operation. The main RAM address is converted into an offset, line index and tag. The corresponding tag in the cache is checked, using the line index to find it. If the tag in the cache matches the requested tag, it's a cache hit. In this case, the line index is used to retrieve the line-sized chunk of data from the cache, and the offset is used to locate the desired byte within the line. If the tags do not match, it's a cache miss. In this case, the appropriate line-sized chunk of main RAM is read into the cache, and the corresponding tag is stored in the cache's tag table. Subsequent reads of nearby addresses (those that reside within the same cache line) will therefore result in much faster cache hits.

3.5.4.4 Set Associativity and Replacement Policy

The simple mapping between cache lines and main RAM addresses described above is known as a *direct-mapped* cache. It means that each address in main RAM maps to only *one* line in the cache. Using our 32 KiB cache with 128-byte lines as an example, the main RAM address 0x203 maps to cache line 4 (because 0x203 is 515, and $\lfloor 515/128 \rfloor = 4$). However, in our example there are 8192 unique cache-line-sized blocks of main RAM that all map to cache line 4. Specifically, cache line 4 corresponds to main RAM addresses 0x200 through 0x27F, but also to addresses 0x8200 through 0x827F, and 0x10200 through 0x1027f, and 8189 *other* cache-line-sized address ranges as well.

When a cache miss occurs, the CPU must load the corresponding cache line from main memory into the cache. If the line in the cache contains no valid

图 3.27。2 路组相联缓存。

数据，我们只需将数据复制到其中即可。但是，如果该行已经包含数据（来自不同的主内存块），则必须覆盖它。
这被称为驱逐缓存行。

直接映射缓存的问题在于它可能导致一些异常情况。例如，两个不相关的主内存块可能会像乒乓球一样不断相互驱逐。如果每个主内存地址可以映射到缓存中的两条或更多不同的缓存行，则可以获得更好的平均性能。在2路组相联缓存中，每个主RAM地址映射到两条缓存行。如图3.27所示。显然，4路组相联缓存的性能甚至优于2路缓存，而8路或16路缓存的性能可以优于4路缓存，依此类推。

一旦我们有多个“缓存路”，缓存控制器就会面临一个难题：当发生缓存未命中时，我们应该逐出哪些“路”，又应该允许哪些“路”继续驻留在缓存中？这个问题的答案因 CPU 设计而异，称为 CPU 的替换策略。一种常用的策略是非最近使用 (NMRU)。在此方案中，会跟踪最近使用的“路”，而逐出始终会影响非最近使用“路”的一个或多个路。其他策略包括先进先出 (FIFO)（这是直接映射缓存中的唯一选项）、最近最少使用 (LRU)、最不频繁使用 (LFU) 和伪随机。有关缓存替换策略的更多信息，请参阅 https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf。

3.5.4.5 多级缓存

命中率衡量的是程序命中缓存的频率，而不是缓存未命中带来的高昂成本。命中率越高，程序的性能就越好（在其他条件相同的情况下）。有一个基本

Figure 3.27. A 2-way set associative cache.

data, we simply copy the data into it and we're done. But if the line already contains data (from a different main memory block), we must overwrite it. This is known as *evicting* the cache line.

The problem with a direct-mapped cache is that it can result in pathological cases. For example, two unrelated main memory blocks might keep evicting one another in a ping-pong fashion. We can obtain better average performance if each main memory address can map to two or more distinct lines in the cache. In a *2-way set associative* cache, each main RAM address maps to two cache lines. This is illustrated in Figure 3.27. Obviously a 4-way set associative cache performs even better than a 2-way, and an 8-way or 16-way cache can outperform a 4-way cache and so on.

Once we have more than one "cache way," the cache controller is faced with a dilemma: When a cache miss occurs, which of the "ways" should we evict and which ones should we allow to stay resident in the cache? The answer to this question differs between CPU designs, and is known as the CPU's *replacement policy*. One popular policy is *not most-recently used* (NMRU). In this scheme, the most-recently used "way" is kept track of, and eviction always affects the "way" or "ways" that are *not* the most-recently used one. Other policies include *first in first out* (FIFO), which is the only option in a direct-mapped cache, least-recently used (LRU), least-frequently used (LFU), and pseudorandom. For more on cache replacement policies, see https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf.

3.5.4.5 Multilevel Caches

The *hit rate* is a measure of how often a program hits the cache, as opposed to incurring the large cost of a cache miss. The higher the hit rate, the better the program will perform (all other things being equal). There is a fundamental

缓存延迟和命中率之间的权衡。缓存越大，命中率越高——但较大的缓存无法靠近 CPU，因此它们往往比较小的缓存更慢。

大多数游戏机至少使用两级缓存。CPU 首先尝试在一级 (L1) 缓存中查找所需数据。该缓存容量较小，但访问延迟非常低。如果找不到所需数据，CPU 会尝试访问容量更大但延迟更高的二级 (L2) 缓存。只有当 L2 缓存中找不到所需数据时，才会产生主存访问的全部开销。由于主内存的延迟相对于 CPU 的时钟频率而言可能非常高，因此某些 PC 甚至包含三级 (L3) 和四级 (L4) 缓存。

3.5.4.6 指令缓存和数据缓存

在为游戏引擎或任何其他性能关键型系统编写高性能代码时，务必意识到数据和代码都会被缓存。指令缓存（I-cache，通常表示为 I$）用于在可执行机器代码运行前预加载，而数据缓存（D-cache，或 D$）用于加速该机器代码执行的读写操作。在一级 (L1) 缓存中，这两个缓存在物理上始终是不同的，因为不希望指令读取导致有效数据被挤出缓存，反之亦然。因此，在优化代码时，我们必须同时考虑 D-cache 和 I-cache 的性能（尽管我们将会看到，优化其中一个缓存往往会对另一个缓存产生积极的影响）。更高级别的缓存（L2、L3、L4）通常不会区分代码和数据，因为它们更大的容量往往会减轻代码驱逐数据或数据驱逐代码的问题。

3.5.4.7 写入策略

我们还没有讨论 CPU 将数据写入 RAM 时会发生什么。缓存控制器如何处理写入操作被称为其写入策略。最简单的缓存类型称为直写缓存；在这种相对简单的缓存设计中，所有对缓存的写入都会立即镜像到主 RAM。在写回（或回拷）缓存设计中，数据首先写入缓存，并且仅在某些情况下将缓存行刷新到主 RAM，例如，当需要清除脏缓存行以便从主 RAM 读取新的缓存行时，或者当程序明确请求进行刷新时。

3.5.4.8 缓存一致性：MESI、MOESI 和 MESIF

当多个 CPU 核心共享一个主内存存储时，情况会变得更加复杂。通常每个核心都有自己的 L1 缓存，但多个核心可能共享一个 L2 缓存，以及主内存。参见图 3.28

trade-off between cache latency and hit rate. The larger the cache, the higher the hit rate—but larger caches cannot be located as close to the CPU, so they tend to be slower than smaller ones.

Most game consoles employ at least two *levels* of cache. The CPU first tries to find the data it's looking for in the *level 1* (L1) cache. This cache is small but has a very low access latency. If the data isn't there, it tries the larger but higher-latency *level 2* (L2) cache. Only if the data cannot be found in the L2 cache do we incur the full cost of a main memory access. Because the latency of main RAM can be so high relative to the CPU's clock rate, some PCs even include *level 3* (L3) and *level 4* (L4) caches.

3.5.4.6 Instruction Cache and Data Cache

When writing high-performance code for a game engine or for any other performance-critical system, it is important to realize that both data and code are cached. The *instruction cache* (I-cache, often denoted I$) is used to preload executable machine code before it runs, while the *data cache* (D-cache, or D$) is used to speed up read and write operations performed by that machine code. In a level 1 (L1) cache, the two caches are always physically distinct, because it is undesirable for an instruction read to cause valid data to be bumped out of the cache or vice versa. So when optimizing our code, we must consider both D-cache and I-cache performance (although as we'll see, optimizing one tends to have a positive effect on the other). Higher-level caches (L2, L3, L4) typically do not make this distinction between code and data, because their larger size tends to mitigate the problems of code evicting data or data evicting code.

3.5.4.7 Write Policy

We haven't talked yet about what happens when the CPU *writes* data to RAM. How the cache controller handles writes is known as its *write policy*. The simplest kind of cache is called a *write-through cache*; in this relatively simple cache design, all writes to the cache are mirrored to main RAM immediately. In a *write-back* (or *copy-back*) cache design, data is first written into the cache and the cache line is only flushed out to main RAM under certain circumstances, such as when a dirty cache line needs to be evicted in order to read in a new cache line from main RAM, or when the program explicitly requests a flush to occur.

3.5.4.8 Cache Coherency: MESI, MOESI and MESIF

When multiple CPU cores share a single main memory store, things get more complicated. It's typical for each core to have its own L1 cache, but multiple cores might share an L2 cache, as well as sharing main RAM. See Figure 3.28

图 3.28。1 级和 2 级缓存。

图示为两级缓存架构，其中两个 CPU 核心共享一个主存储器和一个 L2 缓存。

在多核环境下，系统维护缓存一致性至关重要。这意味着确保多个核心的缓存数据彼此匹配，并与主内存的内容匹配。一致性无需时刻保持一致——重要的是，运行中的程序永远不会察觉到缓存内容不同步。

三种最常见的缓存一致性协议分别是 MESI（修改、独占、共享、无效）、MOESI（修改、拥有、独占、共享、无效）和 MESIF（修改、独占、共享、无效和转发）。我们将在 4.9.4.2 节讨论多核计算架构时更深入地讨论 MESI 协议。

3.5.4.9 避免缓存未命中

显然，缓存未命中无法完全避免，因为数据最终必须在主 RAM 中移动。在存在内存缓存层次结构的情况下编写高性能软件的技巧在于，在 RAM 中合理安排数据，并设计数据操作算法，以最大程度地减少缓存未命中的发生次数。

避免数据缓存未命中的最佳方法是将数据组织到尽可能小的连续块中，然后按顺序访问它们。当数据是连续的（即，不会在内存中频繁“跳转”）时，单次缓存未命中将一次性加载最大数量的相关数据。当

Figure 3.28. Level 1 and level 2 caches.

for an illustration of a two-level cache architecture with two CPU cores sharing one main memory store and an L2 cache.

In the presence of multiple cores, it's important for the system to maintain *cache coherency*. This amounts to making sure that the data in the caches belonging to multiple cores match one another and the contents of main RAM. Coherency doesn't have to be maintained at every moment—all that matters is that the running program can never *tell* that the contents of the caches are out of sync.

The three most common cache coherency protocols are known as MESI (modified, exclusive, shared, invalid), MOESI (modified, owned, exclusive, shared, invalid) and MESIF (modified, exclusive, shared, invalid and forward). We'll discuss the MESI protocol in more depth when we talk about multicore computing architectures in Section 4.9.4.2.

3.5.4.9 Avoiding Cache Misses

Obviously cache misses cannot be totally avoided, since data has to move to and from main RAM eventually. The trick to writing high performance software in the presence of a memory cache hierarchy is to arrange your data in RAM, and design your data manipulation algorithms, in such a way that the minimum number of cache misses occur.

The best way to avoid D-cache misses is to organize your data in *contiguous* blocks that are as *small* as possible and then access them *sequentially*. When the data is contiguous (i.e., you don't "jump around" in memory a lot), a single cache miss will load the maximum amount of relevant data in one go. When

如果数据较小，则更有可能放入单个缓存行（或至少是最小数量的缓存行）。并且，当您顺序访问数据时，可以避免多次驱逐和重新加载缓存行。

避免指令缓存未命中的基本原则与避免数据缓存未命中相同。然而，具体实现方法有所不同。最简单的方法是尽可能缩短高性能循环的代码长度，并避免在最内层循环中调用函数。如果确实需要调用函数，也请尽量缩短其代码长度。这有助于确保循环的整个主体（包括所有被调用的函数）在循环运行期间始终保留在指令缓存中。

谨慎使用内联函数。内联一个小函数可以显著提升性能。然而，过多的内联会导致代码体积膨胀，从而导致性能关键的代码段无法再放入缓存中。

3.5.5 非均匀内存访问（NUMA）

在设计多处理器游戏机或个人计算机时，系统架构师必须在两种根本不同的内存架构之间进行选择：统一内存访问 (UMA) 和非统一内存访问 (NUMA)。

在 UMA 设计中，计算机包含一个大型主 RAM 组，系统中的所有 CPU 核心均可访问该 RAM。每个核心的物理地址空间看起来相同，并且每个核心都可以读取和写入主 RAM 中的所有内存位置。UMA 架构通常利用缓存层次结构来缓解内存访问延迟问题。

UMA 架构的一个问题是，核心之间经常争夺对主 RAM 和任何共享缓存的访问权限。例如，PS4 包含八个核心，排列成两个集群。每个核心都有自己私有的 L1 缓存，但每个由四个核心组成的集群共享一个 L2 缓存，并且所有核心共享主 RAM。因此，核心之间经常会相互争夺对 L2 缓存和主 RAM 的访问权限。

解决核心争用问题的一种方法是采用非均匀内存访问 (NUMA) 设计。在 NUMA 系统中，每个核心都配备了一个相对较小的高速专用 RAM 组，称为本地存储。与 L1 缓存类似，本地存储通常与核心本身位于同一芯片上，并且只能由该核心访问。但与 L1 缓存不同的是，对本地存储的访问是明确的。本地存储可能映射到核心地址空间的一部分，而主 RAM 则映射到不同的地址范围。或者，某些核心可能只能看到其本地存储内的物理地址，并且可能依赖直接内存访问控制器 (DMAC) 来

the data is small, it is more likely to fit into a single cache line (or at least a minimum number of cache lines). And when you access your data sequentially, you avoid evicting and reloading cache lines multiple times.

Avoiding I-cache misses follows the same basic principle as avoiding D-cache misses. However, the implementation requires a different approach. The easiest thing to do is to keep your high-performance loops as small as possible in terms of code size, and avoid calling functions within your innermost loops. If you do opt to call functions, try to keep their code size small too. This helps to ensure that the entire body of the loop, including all called functions, will remain in the I-cache the entire time the loop is running.

Use inline functions judiciously. Inlining a small function can be a big performance boost. However, too much inlining bloats the size of the code, which can cause a performance-critical section of code to no longer fit within the cache.

3.5.5 Nonuniform Memory Access (NUMA)

When designing a multiprocessor game console or personal computer, system architects must choose between two fundamentally different memory architectures: *uniform memory access* (UMA) and *nonuniform memory access* (NUMA).

In a UMA design, the computer contains a single large bank of main RAM which is visible to all CPU cores in the system. The physical address space looks the same to each core, and each can read from and write to all memory locations in main RAM. A UMA architecture typically makes use of a cache hierarchy to mitigate memory access latency issues.

One problem with a UMA architecture is that the cores often contend for access to main RAM and any shared caches. For example, the PS4 contains eight cores arranged into two clusters. Each core has its own private L1 cache, but each cluster of four cores shares a single L2 cache, and all cores share main RAM. As such, the cores often contend with one another for access to the L2 cache and main RAM.

One way to address core contention problems is to employ a *non-uniform memory access* (NUMA) design. In a NUMA system, each core is provided with a relatively small bank of high-speed dedicated RAM called a *local store*. Like an L1 cache, a local store is typically located on the same die as the core itself, and is only accessible by that core. But unlike an L1 cache, access to the local store is explicit. A local store might be mapped to part of a core's address space, with main RAM mapped to a different range of addresses. Alternatively, certain cores might *only* be able to see the physical addresses within its local store, and might rely on a *direct memory access controller* (DMAC) to

图 3.29. PS3 蜂窝宽带架构的简化视图。

在本地存储器和主 RAM 之间传输数据。

3.5.5.1 PS3 上的 SPU 本地商店

PlayStation 3 是 NUMA 架构的经典示例。PS3 包含一个主 CPU，称为 Power 处理单元 (PPU)，八个协处理器，称为协同处理单元 (SPU)，以及一个 NVIDIA RSX 图形处理单元 (GPU)。PPU 独占访问 256 MiB 的主系统 RAM（带有 L1 和 L2 缓存），GPU 独占访问 256 MiB 的视频 RAM (VRAM)，每个 SPU 都有各自私有的 256 KiB 本地存储空间。

主 RAM、视频 RAM 和 SPU 本地存储的物理地址空间彼此完全隔离。这意味着，例如，PPU 无法直接寻址 VRAM 或任何 SPU 本地存储中的内存；任何给定的 SPU 也无法直接寻址主 RAM、视频 RAM 或任何其他 SPU 的本地存储，而只能寻址其自身的本地存储。PS3 的内存架构如图 3.29 所示。

3.5.5.2 PS2 暂存器 (SPR)

回顾更早的 PlayStation 2，我们可以了解到另一种旨在提高整体系统性能的内存架构——

8 只有六个 SPU 可供游戏应用程序使用 - 一个 SPU 保留供操作系统使用，另一个 SPU 完全禁止使用，以解决制造过程中不可避免的缺陷。

Figure 3.29. Simplified view of the PS3's cell broadband architecture.

transfer data between the local store and main RAM.

3.5.5.1 SPU Local Stores on the PS3

The PlayStation 3 is a classic example of a NUMA architecture. The PS3 contains a single main CPU known as the *Power processing unit* (PPU), eight[8] co-processors known as *synergistic processing units* (SPUs), and an NVIDIA RSX graphics processing unit (GPU). The PPU has exclusive access to 256 MiB of main system RAM (with an L1 and L2 cache), the GPU has exclusive access to 256 MiB of video RAM (VRAM), and each SPU has its own private 256 KiB local store.

The physical address spaces of main RAM, video RAM and the SPUs' local stores are all totally isolated from one another. This means that, for example, the PPU cannot directly address memory in VRAM or in any of the SPUs' local stores, and any given SPU cannot directly address main RAM, video RAM, or any of the other SPUs' local stores, only its own local store. The memory architecture of the PS3 is illustrated in Figure 3.29.

3.5.5.2 PS2 Scratchpad (SPR)

Looking even farther back to the PlayStation 2, we can learn about another memory architecture that was designed to improve overall system perfor-

[8]Only six of the SPUs are available for use by game applications—one SPU is reserved for use by the operating system, and one is entirely off-limits to account for inevitable flaws in the fabrication process.

图 3.30。PS2 内存架构的简化视图，展示了 16 KiB 暂存器 (SPR)、主 CPU (EE) 和两个矢量单元 (VU0 和 VU1) 的 L1 缓存、图形合成器 (GS) 可访问的 4 MiB 视频 RAM 组、32 MiB 主 RAM 组、DMA 控制器和系统总线。

曼斯。PS2 上的主 CPU 称为情感引擎 (EE)，它拥有一个称为暂存器（缩写为 SPR）的特殊 16 KiB 内存区域，此外还有一个 16 KiB 的一级指令缓存 (I-cache) 和一个 8 KiB 的一级数据缓存 (D-cache)。PS2 还包含两个矢量协处理器，分别称为 VU0 和 VU1，每个协处理器都有自己的一级 I 和 D 缓存，以及一个称为图形合成器 (GS) 的 GPU，它连接到一个 4 MiB 的视频 RAM 组。PS2 的内存架构如图 3.30 所示。

暂存器位于 CPU 芯片上，因此具有与 L1 缓存相同的低延迟。但与 L1 缓存不同的是，暂存器是内存映射的，因此在程序员看来，它就像是一段常规主 RAM 地址。PS2 上的暂存器本身没有缓存，这意味着对它的读写操作是直接的；它们完全绕过了 EE 的 L1 缓存。

暂存器的主要优势实际上并非其低访问延迟，而是 CPU 无需使用系统总线即可访问暂存器内存。因此，即使系统的地址和数据总线被用于其他用途，也可以对暂存器进行读写操作。例如，游戏可能会设置一系列 DMA 请求，以在主 RAM 和 PS2 中的两个矢量处理单元 (VU) 之间传输数据。当这些 DMA 正在由 DMAC 处理时，以及/或者当 VU 忙于执行计算时（这两种情况都会大量使用系统总线），EE 可以对暂存器中的数据进行计算，而不会干扰 DMA 或 VU 的运行。数据移入和移出暂存器可以通过常规内存移动指令（或 C/C++ 中的 memcpy()）完成，但此任务

Figure 3.30. Simplified view of the PS2's memory architecture, illustrating the 16 KiB scratchpad (SPR), the LI caches of the main CPU (EE) and the two vector units (VU0 and VUI), the 4 MiB bank of video RAM accessible to the graphics synthesizer (GS), the 32 MiB bank of main RAM, the DMA controller and the system buses.

mance. The main CPU on the PS2, called the emotion engine (EE), has a special 16 KiB area of memory called the *scratchpad* (abbreviated SPR), in addition to a 16 KiB L1 instruction cache (I-cache) and an 8 KiB L1 data cache (D-cache). The PS2 also contains two vector coprocessors known as VU0 and VU1, each with their own L1 I- and D-caches, and a GPU known as the *graphics synthesizer* (GS) connected to a 4 MiB bank of video RAM. The PS2's memory architecture is illustrated in Figure 3.30.

The scratchpad is located on the CPU die, and therefore enjoys the same low latency as L1 cache memory. But unlike the L1 cache, the scratchpad is memory-mapped so that it appears to the programmer to be a range of regular main RAM addresses. The scratchpad on the PS2 is itself uncached, meaning that reads and writes from and to it are direct; they bypass the EE's L1 cache entirely.

The scratchpad's main benefit isn't actually its low access latency—it's the fact that the CPU can access scratchpad memory without making use of the system buses. As such, reads and writes from and to the scratchpad can happen while the system's address and data bus are being used for other purposes. For example, a game might set up a chain of DMA requests to transfer data between main RAM and the two vector processing units (VUs) in the PS2. While these DMAs are being processed by the DMAC, and/or while the VUs are busy performing calculations (both of which would make heavy use of the system buses), the EE can be performing calculations on data that resides within the scratchpad, without interfering with the DMAs or the operation of the VUs. Moving data into and out of the scratchpad can be done via regular memory move instructions (or `memcpy()` in C/C++), but this task

也可以通过 DMA 请求来实现。因此，PS2 的暂存器为程序员提供了极大的灵活性和能力，可以最大限度地提高游戏引擎的数据吞吐量。

can also be accomplished via DMA requests. The PS2's scratchpad therefore gives the programmer a lot of flexibility and power in order to maximize a game engine's data throughput.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

4
并行和并发编程

计算性能——通常以每秒百万条指令 (MIPS) 或每秒浮点运算次数 (FLOPS) 来衡量——在过去四十年中一直以惊人的速度持续提升。20 世纪 70 年代末，英特尔 8087 浮点协处理器的运算速度仅为 50 kFLOPS（5 × 10 4 FLOPS），而大约在同一时期，一台大型冰箱大小的 Cray-1 超级计算机的运算速度却能达到 160 MFLOPS（1.6 × 10 8 FLOPS）。如今，Playstation 4 或 Xbox One 等游戏机的 CPU 处理能力约为 100 GFLOPS（10 11 FLOPS），而目前最快的超级计算机——中国的神威太湖之光超级计算机，其 LINPACK 基准测试得分高达 93 PFLOPS（千万亿次浮点运算，即每秒 9.3 × 10 16 次浮点运算）。这对于个人电脑而言是七个数量级的提升，对于超级计算机而言是八个数量级的提升。

许多因素促成了这种快速的改进。早期，从真空管到固态晶体管的转变使得计算硬件得以小型化。随着新型晶体管、新型数字逻辑、新型基板材料和新制造工艺的开发，单个芯片上可蚀刻的晶体管数量急剧增加。这些进步也促进了

4
Parallelism and Concurrent Programming

Computing performance—typically measured in millions of instructions per second (MIPS) or floating-point operations per second (FLOPS)—has been improving at a staggeringly rapid and consistent rate over the past four decades. In the late 1970s, the Intel 8087 floating-point coprocessor could muster only about 50 kFLOPS (5×10^4 FLOPS), while at roughly the same time a Cray-1 supercomputer the size of a large refrigerator could opeate at a rate of roughly 160 MFLOPS (1.6×10^8 FLOPS). Today, the CPU in game consoles like the Playstation 4 or the Xbox One produces roughly 100 GFLOPS (10^{11} FLOPS) of processing power, and the fastest supercomputer, currently China's Sunway TaihuLight, has a LINPACK benchmark score of 93 PFLOPS (peta-FLOPS, or a staggering 9.3×10^{16} floating-point operations per second). This is an improvement of seven orders of magnitude for personal computers, and eight orders of magnitude for supercomputers.

Many factors contributed to this rapid rate of improvement. In the early days, the move from vacuum tubes to solid-state transistors permitted the miniaturization of computing hardware. The number of transistors that could be etched onto a single chip rose dramatically as new kinds of transistors, new types of digital logic, new substrate materials and new manufacturing processes were developed. These advances also contributed to improvements in

图 4.1. 对独立数据进行操作的两个控制流不被视为并发，因为它们不易发生数据竞争。

功耗和 CPU 时钟速度的大幅提升。从 20 世纪 90 年代开始，计算机硬件制造商越来越多地采用并行技术来提高计算性能。

编写在并行计算硬件上正确高效运行的软件比为过去的串行计算机编写软件要困难得多。这需要深入了解硬件的实际工作原理。此外，要充分利用现代计算平台中的多核 CPU，需要一种称为并发编程的软件设计方法。在并发软件系统中，多个控制流协同解决一个共同的问题。这些控制流必须仔细协调。许多在串行程序中运行良好的技术在应用于并发程序时就会失效。因此，对于现代程序员（包括游戏在内的所有行业）来说，深入了解并行计算硬件并精通并发编程技术非常重要。

4.1 定义并发和并行

4.1.1 并发

并发软件利用多个控制流来解决问题。这些控制流可以实现为在单个进程上下文中运行的多个线程，也可以实现为在一台或多台计算机上运行的多个协作进程。多个控制流也可以在一个进程内使用其他技术（例如纤程或协程）来实现。

并发编程与顺序编程的主要区别在于共享数据的读写。如图所示

Figure 4.1. Two flows of control operating on *independent* data are not considered concurrent because they are not prone to data races.

power consumption and dramatic increases in CPU clock speeds. And starting in the 1990s, computer hardware manufacturers have increasingly turned to *parallelism* as a means of improving computing performance.

Writing software that runs *correctly* and *efficiently* on parallel computing hardware is significantly more difficult than writing software for the serial computers of yesteryear. It requires a deep understanding of how the hardware actually works. What's more, taking full advantage of the multicore CPUs found in modern computing platforms requires an approach to software design called *concurrent programming*. In a concurrent software system, multiple flows of control cooperate to solve a common problem. These flows of control must be carefully coordinated. Many of the techniques that work well in serial programs break down when applied to concurrent programs. As such, it's important for modern programmers (in all industries, including games) to have a solid understanding of parallel computing hardware, and to be well versed in concurrent programming techniques.

4.1 Defining Concurrency and Parallelism

4.1.1 Concurrency

A concurrent piece of software utilizes *multiple flows of control* to solve a problem. These flows of control might be implemented as multiple *threads* running within the context of a single process, or multiple cooperating processes running either on a single computer or multiple computers. Multiple flows of control can also be implemented within a process using other techniques such as *fibers* or *coroutines*.

The primary factor that distinguishes *concurrent programming* from *sequential programming* is the reading and/or writing of *shared data*. As shown in

图 4.2。从共享数据文件读取和/或写入共享数据文件的两个控制流都是并发的示例。

图 4.1，如果我们有两个或多个控制流，每个控制流都对完全独立的数据块进行操作，那么从技术上讲这不是并发的一个例子——它只是“同时计算”。

并发编程的核心问题是如何协调共享数据文件的多个读取者和/或多个写入者，以确保可预测的正确结果。这个问题的核心是一种特殊的竞争条件，称为数据竞争，即两个或多个控制流“竞争”成为第一个读取、修改和写入共享数据块的控制流。并发问题的关键在于识别并消除数据竞争。

图 4.2 展示了两个并发示例。

从 4.5 节开始，我们将探讨程序员用来避免数据竞争并编写可靠并发程序的技术。在此之前，我们先来看看并行计算机硬件如何既能为并发软件的运行提供有效的平台，又能提升顺序程序的执行速度。

4.1.2 并行性

在计算机工程中，“并行”一词指的是两个或多个不同的硬件组件同时运行的情况。换句话说，并行计算机硬件可以同时执行多个任务。相比之下，串行计算机硬件一次只能执行一项任务。

1989 年之前，消费级计算设备完全是串行机器。例如，用于 Apple II 和 Commodore 64 个人电脑的 MOS Technology 6502 CPU，以及早期 IBM PC 及其克隆机的核心——英特尔 8086、80286 和 80386 CPU。

Figure 4.2. Two flows of control both reading from a shared data file and/or writing to a shared data file are examples of *concurrency*.

Figure 4.1, if we have two or more flows of control, each operating on a totally independent block of data, then this is not technically an example of concurrency—it's just "computing at the same time."

The central problem of concurrent programming is how to coordinate multiple readers and/or multiple writers of a shared data file, in such a way as to ensure predictable, correct results. At the heart of this problem is a special kind of *race condition* known as a *data race*, in which two or more flows of control "compete" to be the first to read, modify and write a chunk of shared data. The crux of the concurrency problem is to identify and eliminate data races. Two examples of concurrency are illustrated in Figure 4.2.

We'll explore the techiques programmers use to avoid data races and thereby write reliable concurrent programs starting in Section 4.5. But before we do that, let's take a look at how parallel computer hardware can both provide an effective platform for running concurrent software, and improve the execution speed of sequential programs as well.

4.1.2 Parallelism

In computer engineering, the term *parallelism* refers to any situation in which two or more distinct hardware components are operating simultaneously. In other words, *parallel* computer hardware can perform more than one task at a time. In contrast, *serial* computer hardware is capable of doing only one thing at a time.

Prior to 1989, consumer-grade computing devices were exclusively serial machines. Examples include the MOS Technology 6502 CPU, which was used in the Apple II and Commodore 64 personal computers, and the Intel 8086, 80286 and 80386 CPUs which were at the heart of early IBM PCs and clones.

如今，并行计算硬件已无处不在。硬件并行性的一个典型例子是多核 CPU，例如 Intel Core™ i7 或 AMD Ryzen™ 7。但并行性的应用范围非常广泛。例如，单个 CPU 可能包含多个 ALU，因此能够并行执行多个独立计算。另一方面，一组计算机协同工作以解决一个共同的问题，这也是硬件并行性的一个例子。

4.1.2.1 隐式并行与显式并行

对计算机硬件设计中各种并行形式进行分类的一种方法是考虑并行在每种设计中的作用。换句话说，并行在给定的设计中解决了什么问题？沿着这个思路思考，我们可以将并行粗略地分为两类：

- **隐式并行性，以及**
- **明确的并行性**。

隐式并行是指在 CPU 中使用并行硬件组件来提升单个指令流的性能。这也称为指令级并行 (ILP)，因为 CPU 执行来自单个流（单个线程）的指令，但每条指令都以一定程度的硬件并行性执行。隐式并行的示例包括：

- 流水线，
- 超标量架构，以及
- 超长指令字（VLIW）架构。

我们将在4.2节探讨隐式并行。GPU也广泛使用隐式并行；我们将在4.11节更深入地探讨GPU的设计和编程。

显式并行是指在 CPU、计算机或计算机系统中使用重复的硬件组件，以便同时运行多个指令流。换句话说，显式并行硬件旨在比串行计算平台更高效地运行并发软件。最常见的显式并行示例包括：

- 超线程 CPU，
- 多核 CPU，

Today, parallel computing hardware is ubiquitous. One obvious example of hardware parallelism is a multicore CPU, such as the Intel Core™ i7 or the AMD Ryzen™ 7. But parallelism can be employed at a wide range of granularities. For example, a single CPU might contain multiple ALUs and therefore be capable of performing multiple independent calculations in parallel. And at the other end of the spectrum, a cluster of computers working in tandem to solve a common problem is also an example of hardware parallelism.

4.1.2.1 Implicit versus Explicit Parallelism

One way to classify the various forms of parallelism in computer hardware design is to consider the *purpose* of parallelism in each. In other words, what problem does parallelism solve in a given design? Thinking along these lines, we can divide parallelism coarsely into two categories:

- implicit parallelism, and
- explicit parallelism.

Implicit parallelism refers to the use of parallel hardware components within a CPU for the purpose of improving the performance of a *single* instruction stream. This is also known as *instruction level parallelism* (ILP), because the CPU executes instructions from a single stream (a single thread) but each instruction is executed with some degree of hardware parallelism. Examples of implicit parallelism include:

- pipelining,
- superscalar architectures, and
- very long instruction word (VLIW) architectures.

We'll explore implicit parallelism in Section 4.2. GPUs also make extensive use of implicit parallelism; we'll take a closer look at the design and programming of GPUs in Section 4.11.

Explicit parallelism refers to the use of duplicated hardware components within a CPU, computer or computer system, for the purpose of running *more than one instruction stream* simultaneously. In other words, explicitly parallel hardware is designed to run *concurrent* software more efficiently than would be possible on a serial computing platform. The most common examples of explicit parallelism are:

- hyperthreaded CPUs,
- multicore CPUs,

多处理器计算机，
• 计算机集群，
• 网格计算，以及
• 云计算。

我们将在第 4.3 节进一步研究这些明确的并行架构。

4.1.3 任务并行与数据并行

理解并行性的另一种方法是根据并行完成的工作类型将其分为两大类。

• 任务并行。当多个异构操作并行执行时，我们称之为任务并行。例如，在一个核心上执行动画计算，同时在另一个核心上执行碰撞检查，就属于这种并行形式。

• 数据并行性。当单个操作并行执行于多个数据项时，这被称为数据并行性。通过在四个核心上分别运行 250 个矩阵计算来计算 1000 个蒙皮矩阵，这就是数据并行性的一个例子。

大多数实际并发程序在不同程度上利用任务并行和数据并行。

4.1.4 弗林分类法

另一种对我们在计算硬件中遇到的不同并行度进行分类的方法是使用 Flynn 分类法。这种方法由斯坦福大学的 Michael J. Flynn 于 1966 年提出，它将并行性分解为一个二维空间。沿着一个轴，我们有并行控制流的数量（Flynn 将其称为在任何给定时刻并行运行的指令数量）。在另一个轴上，我们有程序中每条指令操作的不同数据流的数量。

因此，空间被划分为四个象限：

• 单指令、单数据（SISD）：对单个数据流进行操作的单个指令流。

• 多指令、多数据（MIMD）：多个指令流对多个独立数据流进行操作。

• 单指令、多数据 (SIMD)：对多个数据流进行操作的单个指令流（即同时对多个独立的数据流执行相同的操作序列）。

- multiprocessor computers,
- computer clusters,
- grid computing, and
- cloud computing.

We'll investigate these explicitly parallel architectures further in Section 4.3.

4.1.3 Task versus Data Parallelism

Another way to understand parallelism is to divide it into two broad categories based on the kind of work being done in parallel.

- *Task parallelism*. When multiple heterogeneous operations are performed in parallel, we call this task parallelism. Performing animation calculations on one core while performing collision checks on another would be an example of this form of parallelism.

- *Data parallelism*. When a *single* operation is performed on multiple data items in parallel, it is called data parallelism. Calculating 1000 skinning matrices by running 250 matrix calculations on each of four cores would be an example of data parallelism.

Most real concurrent programs make use of both task and data parallelism to varying degrees.

4.1.4 Flynn's Taxonomy

Yet another way to classify the varying degrees of parallelism we'll encounter in computing hardware is to use *Flynn's Taxonomy*. Proposed by Michael J. Flynn of Stanford University in 1966, this approach breaks down parallelism into a two-dimensional space. Along one axis, we have the number of parallel flows of control (which Flynn refers to as the number of instructions running in parallel at any given moment). On the other axis, we have the number of distinct data streams being operated on by each instruction in the program. The space is thus divided into four quadrants:

- *Single instruction, single data (SISD)*: A single instruction stream operating on a single data stream.
- *Multiple instruction, multiple data (MIMD)*: Multiple instruction streams operating on multiple independent data streams.
- *Single instruction, multiple data (SIMD)*: A single instruction stream operating on multiple data streams (i.e., performing the same sequence of operations on multiple independent streams of data simultaneously).

图 4.3 SISD 示例。单个 ALU 首先执行乘法，然后执行除法。

• 多指令单数据 (MISD)：多个指令流均在单个数据流上运行。（MISD 在游戏中很少使用，但一种常见的应用是通过冗余提供容错能力。）

4.1.4.1 单一数据与多数据

这里需要注意的是，“数据流”不仅仅是一个数字数组。大多数算术运算符都是二进制的——它们对两个输入进行运算，产生一个输出。当应用于二进制算术时，“单个数据”一词指的是一对输入，一个输出。例如，让我们看看如何在四个 Flynn 类别下分别完成两个二进制算术运算：乘法（a × b）和除法（c / d）：

• 在 SISD 架构中，单个 ALU 首先执行乘法，然后执行除法。如图 4.3 所示。

• 在 MIMD 架构中，两个 ALU 并行执行操作，分别对两个独立的指令流进行操作。如图 4.4 所示。

• MIMD 分类也适用于单个 ALU 通过时间分片处理两个独立指令流的情况，如图 4.5 所示。

• 在 SIMD 架构中，一个称为向量处理单元 (VPU) 的“宽 ALU”首先执行乘法，然后执行除法，但每条指令都对一对四元素输入向量进行操作，并产生一个四元素输出向量。图 4.6 展示了这种方法。

Figure 4.3. Example of SISD. A single ALU performs the multiply first, followed by the divide.

- *Multiple instruction, single data (MISD)*: Multiple instruction streams all operating on a single data stream. (MISD is rarely used in games, but one common application is to provide fault tolerance via redundancy.)

4.1.4.1 Single versus Multiple Data

It's important to realize here that a "data stream" isn't just an array of numbers. Most arithmetic operators are binary—they operate on two inputs to produce a single output. When applied to binary arithmetic, the term "single data" refers to a single *pair* of inputs, with a single output. As an example, let's have a look at how two binary arithmetic operations, a multiply ($a \times b$) and a divide (c/d), might be accomplished under each of the four Flynn categories:

- In a SISD architecture, a single ALU performs the multiply first, followed by the divide. This is illustrated in Figure 4.3.

- In a MIMD architecture, two ALUs perform operations in parallel, operating on two independent instruction streams. This is shown in Figure 4.4.

- The MIMD classification also applies to the case in which a single ALU processes two independent instruction streams via *time-slicing*, as shown in Figure 4.5.

- In a SIMD architecture, a single "wide ALU" known as a vector processing unit (VPU) performs the multiply first, followed by the divide, but each instruction operates on a pair of four-element input vectors and produces a four-element output vector. Figure 4.6 illustrates this approach.

图 4.4 MIMD 示例。两个 ALU 并行执行操作。

图 4.5 时间片 MIMD 示例。单个 ALU 代表两个独立的指令流执行操作，可能通过在它们之间交替进行。

- 在 MISD 架构中，两个 ALU 处理相同的指令流（先乘法，后除法），理想情况下会产生相同的结果。如图 4.7 所示，该架构主要用于通过冗余实现容错。ALU 1 充当 ALU 0 的“热备用”，反之亦然，这意味着如果其中一个 ALU 发生故障，系统可以无缝切换到另一个 ALU。

4.1.4.2 GPU并行性：SIMT

近年来，Flynn 分类法中又添加了第五种分类，以解释图形处理单元 (GPU) 的设计。单指令多线程 (SIMT) 本质上是 SIMD 和 MIMD 的混合体，主要用于 GPU 的设计。它混合了 SIMD 处理（单指令多数据流）和 MIMD 处理（多数据流）。

Figure 4.4. Example of MIMD. Two ALUs perform operations in parallel.

Figure 4.5. Example of time-sliced MIMD. A single ALU performs operations on behalf of two independent instruction streams, perhaps by alternating between them.

- In a MISD architecture, two ALUs process the same instruction stream (multiply first, followed by divide) and ideally produce identical results. Illustrated in Figure 4.7, this architecture is primarily useful for implementing fault tolerance via redundancy. ALU 1 acts as a “hot spare” for ALU 0 and vice-versa, meaning that if one of the ALUs experiences a failure, the system can seamlessly switch to the other.

4.1.4.2 GPU Parallelism: SIMT

In recent years, a fifth classification has been added to Flynn’s taxonomy to account for the design of graphics processing units (GPU). *Single instruction multiple thread* (SIMT) is essentially a hybrid between SIMD and MIMD, used primarily in the design of GPUs. It mixes SIMD processing (a single instruc-

图 4.6 SIMD 示例。单个向量处理单元 (VPU) 首先执行乘法，然后执行除法，但每条指令对一对四元素输入向量进行操作，并产生一个四元素输出向量。

图 4.7 MISD 示例。两个 ALU 处理相同的指令流（先乘法，然后除法），理想情况下会产生相同的结果。

多线程（通过时间分片共享一个处理器的多个指令流）可以实现对多个数据流同时进行操作。

“SIMT”一词由 NVIDIA 首创，但它可用于描述任何 GPU 的设计。“众核”一词也常用于指代 SIMT 设计（即 GPU 由相对大量的轻量级 SIMD 核心组成），而“多核”一词则指代 MIMD 设计（即 CPU 具有相对较少数量的重量级通用核心）。我们将在 4.11 节中仔细探讨 GPU 采用的 SIMT 设计。

Figure 4.6. Example of SIMD. A single vector processing unit (VPU) performs the multiply first, followed by the divide, but each instruction operates on a pair of four-element input vectors and produces a four-element output vector.

Figure 4.7. Example of MISD. Two ALUs process the same instruction stream (multiply first, followed by divide) and ideally produce identical results.

tion operating on multiple data streams simultaneously) with multithreading (more than one instruction stream sharing a processor via time-slicing).

The term “SIMT” was coined by NVIDIA, but it can be used to describe the design of any GPU. The term *manycore* is also often used to refer to a SIMT design (i.e., a GPU consisting of a relatively large number of lightweight SIMD cores) whereas the term *multicore* refers to MIMD designs (i.e., a CPU with a relatively smaller number of heavyweight general purpose cores). We'll take a closer look at the SIMT design employed by GPUs in Section 4.11.

4.1.5 并发与并行的正交性

这里需要强调的是，并发软件并不需要并行硬件，并行硬件也不仅仅用于运行并发软件。例如，一个并发多线程程序可以通过抢占式多任务处理在单个串行 CPU 核心上运行（参见 4.4.4 节）。同样，指令级并行旨在提升单线程的性能，因此对并发和顺序软件都有好处。因此，虽然并发和并行密切相关，但它们实际上是两个相互独立的概念。

只要我们的系统涉及共享数据对象的多个读取者和/或多个写入者，我们就拥有一个并发系统。并发可以通过抢占式多任务（在串行或并行硬件上）或真正的并行（每个线程在不同的内核上执行）来实现——本章将要学习的技术适用于任何一种方式。

4.1.6 本章路线图

在接下来的章节中，我们将首先关注隐式并行，以及如何优化软件以充分利用它。接下来，我们将回顾最常见的显式并行形式。然后，我们将探索用于驾驭显式并行计算平台的各种并发编程技术。最后，我们将通过讨论 SIMD 矢量处理及其在 GPU 设计和通用 GPU 编程 (GPGPU) 技术中的应用来完善并行编程的讨论。

4.2 隐式并行

在 4.1.2.1 节中，我们提到隐式并行是指利用并行计算硬件来提升单线程的执行速度。CPU 制造商自 20 世纪 80 年代末开始在其消费产品中使用隐式并行，旨在使现有代码在特定产品线的新一代 CPU 上获得更快的运行速度。

有多种方法可以将并行性应用于提升 CPU 单线程性能的问题。最常见的是流水线 CPU 架构、超标量设计和超长指令字 (VLIW) 架构。我们将首先了解流水线 CPU 的工作原理，然后再讨论其他两种隐式并行的变体。

4.1.5 Orthogonality of Concurrency and Parallelism

We should stress here that concurrent software doesn't *require* parallel hardware, and parallel hardware isn't *only* for running concurrent software. For example, a concurrent multithreaded program can run on a single, serial CPU core via preemptive multitasking (see Section 4.4.4). Likewise, instruction level parallelism is intended to improve the performance of a single thread, and therefore benefits both concurrent and sequential software. So while they are closely related, concurrency and parallelism are really orthogonal concepts.

As long as our system involves *multiple readers* and/or *multiple writers* of a shared data object, we have a concurrent system. Concurrency can be achieved via preemptive multitasking (on serial *or* parallel hardware) or via true parallelism (in which each thread executes on a distinct core)—the techniques we'll learn in this chapter will be applicable either way.

4.1.6 Roadmap of the Chapter

In the following sections, we'll first turn our attention to implicit parallelism, and how best to optimize our software to take advantage of it. Next, we'll review the most common forms of explicit parallelism. Then we'll explore the various concurrent programming techniques used to harness explicitly parallel computing platforms. Finally, we'll round out the discussion of parallel programming by discussing SIMD vector processing, and how it applies to GPU design and general-purpose GPU programming (GPGPU) techniques.

4.2 Implicit Parallelism

In Section 4.1.2.1 we said that *implicit parallelism* is the use of parallel computing hardware for the purpose of improving the execution speed of a single thread. CPU manufacturers began using implicit parallelism in their consumer products in the late 1980s, in an attempt to make *existing code* run faster on each new generation of CPUs within a given product line.

There are a number of ways to apply parallelism to the problem of improving a CPU's single-threaded performance. The most common are *pipelined* CPU architectures, *superscalar* designs, and *very long instruction word* (VLIW) architectures. We'll begin by having a look at how a pipelined CPU functions, and then address the other two variants of implicit parallelism.

Clock Cycle	Fetch	Dec	Exec	Mem	WB
0	A				
1		A			
2			A		
3				A	
4					A
5	B				
6		B			

图 4.8. 在非流水线 CPU 中，指令阶段大部分时间处于空闲状态。

4.2.1 流水线

为了使一条机器语言指令能够被 CPU 执行，它必须经过多个不同的阶段。每个 CPU 的设计都略有不同——有些 CPU 设计采用大量的粒度阶段，而另一些则采用较少量的粗粒度阶段。但是，每个 CPU 都会以某种方式实现以下基本阶段：

- 取指。从指令指针寄存器（IP）指向的内存位置读取要执行的指令。
- 解码。指令字分解为其操作码、寻址模式和可选操作数。
- 执行。根据操作码，选择 CPU 内相应的功能单元（ALU、FPU、内存控制器等）。指令连同相关操作数数据一起被发送到选定的组件进行处理。然后，功能单元执行其操作。
- 内存访问。如果指令涉及读取或写入内存，则内存控制器在此阶段执行相应的操作。
- 寄存器写回。执行指令的功能单元（ALU、FPU 等）将其结果写回到目标寄存器。

图 4.8 追踪了两条指令“A”和“B”在串行 CPU 的五个执行阶段中的路径。你会立即注意到，该图包含大量空白：当一个阶段忙于执行一条指令时，所有其他阶段都无所事事，无所事事。

Clock Cycle	Fetch	Dec	Exec	Mem	WB
0	A				
1		A			
2			A		
3				A	
4					A
5	B				
6		B			

Figure 4.8. In a non-pipelined CPU, instruction stages are idle much of the time.

4.2.1 Pipelining

In order for a single machine language instruction to be executed by the CPU, it must pass through a number of distinct *stages*. Every CPU's design is a bit different—some CPU designs employ a large number of granular stages, while others utilize a smaller number of coarse-grained stages. However every CPU implements the following basic stages in one way or another:

- *Fetch*. The instruction to be executed is read from the memory location pointed to by the instruction pointer register (IP).
- *Decode*. The instruction word is decomposed into its opcode, addressing mode and optional operands.
- *Execute*. Based on the opcode, the appropriate functional unit within the CPU is selected (ALU, FPU, memory controller, etc.). The instruction is dispatched to the selected component for processing, along with any relevant operand data. The functional unit then performs its operation.
- *Memory access*. If the instruction involves reading or writing memory, the memory controller performs the appropriate operation during this stage.
- *Register write-back*. The functional unit executing the instruction (ALU, FPU, etc.) writes its results back into the destination register.

Figure 4.8 traces the path of two instructions named “A” and “B” through the five execution phases of a serial CPU. You'll notice right away that this diagram contains a lot of blank space: While one stage is busy doing its thing to an instruction, all the other stages are twiddling their thumbs, doing nothing.

图 4.9。流水线标量 CPU 的组件。

Clock Cycle	Fetch	Dec	Exec	Mem	WB
0	A				
1	B	A			
2	C	B	A		
3	D	C	B	A	
4	E	D	C	B	A
5	F	E	D	C	B
6	G	F	E	D	C

图 4.10. 通过流水线 CPU 的理想指令流。

指令执行的每个阶段实际上由 CPU 内的不同硬件处理，如图 4.9 所示。控制单元 (CU) 和内存控制器负责指令提取阶段。然后，CU 内的另一个电路负责解码阶段。ALU、FPU 或 VPU 负责执行阶段的大部分工作。内存阶段由内存控制器执行。最后，写回阶段主要涉及寄存器。CPU 内不同电路之间的这种分工是提高 CPU 效率的关键：我们只需让所有阶段的硬件始终保持忙碌即可。

这种解决方案被称为流水线。我们不再等待每条指令完成所有五个阶段后再开始执行下一条指令，而是在每个时钟周期开始执行一条新指令。因此，多条指令可以同时“飞行”。该过程如图 4.10 所示。

流水线有点像洗衣服。如果你有大量衣物需要洗，那么等到每件衣物都洗完后再进行处理效率不高。

Figure 4.9. Components of a pipelined scalar CPU.

Clock Cycle	Fetch	Dec	Exec	Mem	WB
0	A				
1	B	A			
2	C	B	A		
3	D	C	B	A	
4	E	D	C	B	A
5	F	E	D	C	B
6	G	F	E	D	C

Figure 4.10. Ideal flow of instructions through a pipelined CPU.

Each of the stages of instruction execution is actually handled by different hardware within the CPU, as shown in Figure 4.9. The control unit (CU) and memory controller handle the instruction fetch stage. A different circuit within the CU then handles the decode stage. The ALU, FPU or VPU handles the lion's share of the execute stage. The memory stage is performed by the memory controller. And finally, the write-back stage primarily involves the registers. This division of labor amongst different circuits within the CPU is the key to making the CPU more efficient: We just need to keep all the stages' hardware busy all the time.

The solution is known as *pipelining*. Instead of waiting for each instruction to complete all five stages before starting to execute the next one, we begin the execution of a new instruction on *every* clock cycle. Multiple instructions are therefore "in flight" simultaneously. This process is illustrated in Figure 4.10.

Pipelining is a bit like doing laundry. If you have a large number of loads to do, it wouldn't be very efficient to wait until each load has been washed and

先烘干衣物，再启动下一批衣物——洗衣机忙碌时，烘干机就会闲置，反之亦然。最好让两台机器始终保持忙碌状态，比如第一批衣物刚放进烘干机，就启动第二批衣物的洗涤，以此类推。

流水线是一种称为指令级并行 (ILP) 的并行形式。ILP 的设计原则是，对程序员来说，它是透明的。理想情况下，在给定的时钟速度下，在标量 CPU 上正常运行的程序应该能够在流水线 CPU 上正确运行，而且速度更快，前提是这两个处理器支持相同的指令集架构 (ISA)。理论上，具有 N 级流水线的 CPU 执行程序的速度比串行 CPU 快 N 倍。然而，正如我们将在 4.2.4 节中探讨的那样，由于指令流中指令之间存在各种依赖关系，流水线的性能并不总是如我们预期的那样好。因此，致力于编写高性能代码的程序员不能忽视 ILP。我们必须接受它、理解它，有时甚至调整代码和/或数据的设计，以便最大限度地利用流水线 CPU。

4.2.2 延迟与吞吐量

流水线的延迟是指完成一条指令所需的时间。它等于流水线中所有阶段延迟的总和。用时间变量 T 表示延迟，我们可以这样写：

$$T_{\text{pipeline}} = \sum_{i=0}^{N-1} T_i \tag{4.1}$$

对于具有 N 个阶段的管道。

流水线的吞吐量或带宽衡量的是单位时间内能够处理的指令数量。流水线的吞吐量由其最慢阶段的延迟决定——就像链条的强度取决于其最薄弱的环节一样。吞吐量可以被认为是频率 f，以每秒指令数来衡量。它可以写成如下形式：

$$f = \frac{1}{\max(T_i)}. \tag{4.2}$$

4.2.3 管道深度

我们说过，CPU 中的每个阶段都可能具有不同的延迟（T i），并且延迟最长的阶段决定了整个处理器的吞吐量。在每个时钟周期，其他阶段处于空闲状态，等待最长的

dried before starting the next one—while the washer is busy, the dryer would be sitting idle, and vice versa. It's much better to keep both machines busy at all times, by starting the second load washing just as soon as the first load goes into the dryer, and so on.

Pipelining is a form of parallelism known as *instruction-level parallelism* (ILP). For the most part, ILP is designed to be transparent to the programmer. Ideally, at a given clock speed, a program that runs properly on a scalar CPU should be able to run correctly—but faster—on a pipelined CPU, as long as the two processors support the same instruction set architecture (ISA) of course. In theory, a CPU with a pipeline that is N stages deep can execute a program N times faster than its serial counterpart. However, as we'll explore in Section 4.2.4, pipelining doesn't always perform as well as we would expect, thanks to various kinds of *dependencies* between instructions in the instruction stream. Programmers who are interested in writing high-performance code therefore cannot remain oblivious to ILP. We must embrace it, understand it, and sometimes adjust the design of our code and/or data in order to get the most out of a pipelined CPU.

4.2.2 Latency versus Throughput

The *latency* of a pipeline is the amount of time required to completely process a single instruction. This is just the sum of the latencies of all the stages in the pipeline. Denoting latencies with the time variable T, we can write:

$$T_{\text{pipeline}} = \sum_{i=0}^{N-1} T_i \tag{4.1}$$

for a pipeline with N stages.

The *throughput* or *bandwidth* of a pipeline is a measure of how many instructions it can process per unit time. The throughput of a pipeline is determined by the latency of its slowest stage—much as a chain is only as strong as its weakest link. The throughput can be thought of as a frequency f, measured in instructions per second. It can be written as follows:

$$f = \frac{1}{\max(T_i)}. \tag{4.2}$$

4.2.3 Pipeline Depths

We said that each stage in a CPU can potentially have a different latency (T_i), and that the stage with the longest latency dictates the throughput of the entire processor. On each clock cycle, the other stages sit idle waiting for the longest

阶段完成。理想情况下，我们希望 CPU 中的所有阶段都具有大致相同的延迟。

这一目标可以通过增加流水线的总级数来实现：如果某个级的执行时间比其他级长得多，可以将其拆分成两个或多个较短的级，以使所有级的延迟大致相等。然而，我们不能一直细分级数。级数越多，总体指令延迟就越高。这会增加流水线停顿的成本（参见第 4.2.4 节）。因此，CPU 制造商试图在通过更深的流水线提高吞吐量和控制总体指令延迟之间取得平衡。因此，实际的 CPU 流水线级数范围从至少 4 或 5 级到最多 30 级。

4.2.4 摊位

有时，CPU 无法在特定时钟周期发出新指令。这种情况称为停顿。在这样的时钟周期，流水线的第一级处于空闲状态。在下一个时钟周期，第二级将处于空闲状态，依此类推。因此，停顿可以被认为是一个空闲时间的“气泡”，它以每个时钟周期一个级的速度在流水线中传播。这些气泡有时被称为延迟槽。

4.2.5 数据依赖关系

停顿是由正在执行的指令流中的指令之间的依赖关系引起的。例如，考虑以下指令序列：

```
mov  ebx,5      ;; load the value 5 into register EBX
imul eax,10     ;; multiply the contents of EAX by 10
                ;; (result stored in EAX)
add  eax,7      ;; add 7 to EAX (result stored in EAX)
```

理想情况下，我们希望在三个连续的时钟周期内发出 mov 、 imul 和 add 指令，以尽可能保持流水线的繁忙。但在这种情况下， imul 指令的结果会被其后的 add 指令使用，因此 CPU 必须等到 imul 指令完全执行完流水线后才能发出 add 指令。如果流水线包含五个阶段，则意味着浪费了四个周期（参见图 4.11）。指令之间的这种依赖关系称为 数据依赖 。

实际上，指令之间存在三种依赖关系，可能会导致停顿：

- 数据依赖性，

stage to complete. Ideally, then, we'd like all of the stages in our CPU to have roughly the same latency.

This goal can be achieved by increasing the total number of stages in the pipeline: If one stage is taking much longer than the others, it can be broken into two or more shorter stages in an attempt to make all the latencies roughly equal. However, we can't just keep subdividing stages forever. The larger the number of stages, the higher the overall instruction latency will be. This increases the cost of pipeline stalls (see Section 4.2.4). Therefore CPU manufacturers try to strike a balance between increasing throughput via deeper pipelines, and keeping the overall instruction latency in check. As a result, real CPU pipelines range from 4 or 5 stages at minimum, to something on the order of 30 stages at most.

4.2.4 Stalls

Sometimes the CPU is unable to issue a new instruction on a particular clock cycles. This is called a *stall*. On such a clock cycle, the first stage in the pipeline lies idle. On the next clock cycle, the second stage will be idle, and so on. A stall can therefore be thought of as a "bubble" of idle time that propagates through the pipeline at a rate of one stage per clock cycle. These bubbles are sometimes called *delay slots*.

4.2.5 Data Dependencies

Stalls are caused by *dependencies* between instructions in the instruction stream being executed. For example, consider the following sequence of instructions:

```
mov  ebx,5      ;; load the value 5 into register EBX
imul eax,10     ;; multiply the contents of EAX by 10
                ;; (result stored in EAX)
add  eax,7      ;; add 7 to EAX (result stored in EAX)
```

Ideally, we'd like to issue the `mov`, `imul` and `add` instructions on three consecutive clock cycles, to keep the pipeline as busy as possible. But in this case, the results of the `imul` instruction are used by the `add` instruction that follows it, so the CPU must wait until the `imul` has made it all the way through the pipeline before issuing the `add`. If the pipeline contains five stages, that means four cycles are wasted (see Figure 4.11). These kinds of dependencies between instructions are called *data dependencies*.

There are actually three kinds of dependencies between instructions that can cause stalls:

- data dependencies,

图 4.11 指令之间的数据依赖导致流水线停顿。

- 控制依赖关系（也称为分支依赖关系），以及
- 结构依赖性（也称为资源依赖性）。

首先，我们将讨论如何避免数据依赖，然后我们将研究分支依赖以及如何减轻其影响。最后，我们将介绍超标量 CPU 架构，并讨论它们如何在流水线 CPU 中产生结构依赖。

4.2.5.1 指令重新排序

为了减轻数据依赖性的影响，我们需要在 CPU 等待依赖指令通过流水线时，为其找到一些其他指令来执行。这通常可以通过重新排序程序中的指令来实现（同时注意不要在此过程中改变程序的行为）。对于任何给定的相互依赖的指令对，我们希望找到一些不依赖于它们的邻近指令，并将这些指令上移或下移，使它们最终在依赖指令对之间运行，从而用有用的工作填充“气泡”。

当然，指令重排序可以由一位勇于创新、不介意深入研究汇编语言的程序员手动完成。不过，值得庆幸的是，这通常没有必要：如今的优化编译器非常擅长自动重排序指令，以减少或消除数据依赖性的影响。

作为程序员，我们当然不应该盲目地相信编译器能够完美地优化我们的代码——在编写高性能代码时，查看反汇编代码并验证编译器是否做了合理的工作始终是一个好主意。但话虽如此，我们也应该记住 80/20 规则

Figure 4.11. A data dependency between instructions causes a pipeline stall.

- control dependencies (also known as branch dependencies), and
- structural dependencies (also known as resource dependencies).

First we'll discuss how to avoid data dependencies, then we'll have a look at branch dependencies and how to mitigate their effects. Finally, we'll introduce superscalar CPU architectures and discuss how they can give rise to structural dependencies in a pipelined CPU.

4.2.5.1 Instruction Reordering

To mitigate the effects of a data dependency, we need to find some other instructions for the CPU to execute while it waits for the dependent instruction to make its way through the pipeline. This can often be accomplished by *reordering* the instructions in the program (while taking care not to change the *behavior* of the program in the process). For any given pair of interdependent instructions, we want to find some nearby instructions that are *not* dependent on them, and move those instructions up or down so that they end up running *between* the dependent instruction pair, thus filling the "bubble" with useful work.

Instruction reordering may of course be done by hand, by an adventurous programmer who doesn't mind diving into assembly language programming. Thankfully, however, this is often not necessary: Today's optimizing compilers are very good at reordering instructions automatically to reduce or eliminate the impact of data dependencies.

As programmers, of course we shouldn't blindly trust the compiler to optimize our code perfectly—when writing high-performance code, it's always a good idea to have a look at the disassembly and verify that the compiler did a reasonable job. But that said, we should also remember the 80/20 rule

（第 2.3 节）并且只花时间优化对整体性能有明显影响的 20% 或更少的代码。

4.2.5.2 乱序执行

编译器和程序员并非唯一能够重新排序机器语言指令序列以防止停顿的人。当今的许多 CPU 都支持一种称为乱序执行的功能，该功能使它们能够动态检测指令之间的数据依赖关系，并自动解决它们。

为了实现这一功能，CPU 会提前查看指令流，并分析指令的寄存器使用情况，以检测它们之间的依赖关系。当发现依赖关系时，CPU 会在“提前查看窗口”中搜索另一条不依赖于任何当前正在执行的指令的指令。如果找到一条指令，则会将其执行（乱序！）以保持流水线繁忙。乱序执行的具体工作原理超出了我们的讨论范围。简而言之，作为程序员，我们不能依赖 CPU 按照我们（或编译器）编写的顺序执行指令。

编译器的优化器和 CPU 的乱序执行逻辑都非常谨慎地确保程序的行为不会因指令重排序而改变。然而，正如我们将在 4.9.3 节中看到的那样，编译器优化和乱序执行可能会导致并发程序（即由多个共享数据的线程组成的程序）中出现错误。这也是并发编程比串行编程需要更加谨慎的众多原因之一。

4.2.6 分支依赖关系

当流水线 CPU 遇到条件分支指令（例如，if 语句，或者 for 或 while 循环末尾的条件表达式）时会发生什么？为了回答这个问题，让我们考虑以下 C/C++ 代码：

```
int SafeIntegerDivide(int a, int b, int defaultVal)
{
    return (b != 0) ? a / b : defaultVal;
}
```

如果我们查看此函数的反汇编，它在 Intel x86 CPU 上可能看起来像这样：

(Section 2.3) and only spend time optimizing the 20% or less of our code that actually has a noticable impact on overall performance.

4.2.5.2 Out-of-Order Execution

Compilers and programmers aren't the only ones capable of reordering a sequence of machine language instructions to prevent stalls. Many of today's CPUs support a feature known as *out-of-order execution*, which enables them to dynamically detect data dependencies between instructions, and automatically resolve them.

To accomplish this feat, the CPU looks ahead in the instruction stream and analyzes the instructions' register usage in order to detect dependencies between them. When a dependency is found, the "look-ahead window" is searched for another instruction that is *not* dependent on any of the currently-executing instructions. If one is found, it is issued (out of order!) to keep the pipeline busy. The details of how out-of-order execution works is beyond our scope here. Suffice it to say that as programmers, we cannot rely on the CPU to execute the instructions in the same order we (or the compiler) wrote them.

Both the compiler's optimizer and the CPU's out-of-order execution logic take great care to ensure that the behavior of the program doesn't change as a result of instruction reordering. However, as we'll see in Section 4.9.3, compiler optimizations and out-of-order execution *can* cause bugs in a concurrent program (i.e., a program consisting of multiple threads that share data). This is one of the many reasons why concurrent programming requires more care than serial programming.

4.2.6 Branch Dependencies

What happens when a pipelined CPU encounters a conditional branch instruction (e.g., an `if` statement, or the conditional expression at the end of a `for` or `while` loop)? To answer this question, let's consider the following C/C++ code:

```
int SafeIntegerDivide(int a, int b, int defaultVal)
{
    return (b != 0) ? a / b : defaultVal;
}
```

If we looked at the disassembly for this function, it might look something like this on an Intel x86 CPU:

图 4.12。比较指令和条件分支指令之间的依赖关系称为分支依赖关系。

```
    ; function preamble omitted for clarity...

    ; first, put the default into the return register
    mov  eax,dword ptr [defaultVal]

    mov  esi,dword ptr [b] ; check (b != 0)
    cmp  esi,0
    jz   SkipDivision

    mov  eax, dword ptr[a] ; divisor (a) must be in EDX:EAX
    cdq                    ; ... so sign-extend into EDX
    idiv esi               ; quotient lands in EAX

SkipDivision:
    ; function postamble omitted for clarity...
    ret                    ; EAX is the return value
```

这里的依赖关系是指 cmp（比较）指令和 jz（等于零时跳转）指令之间的依赖关系。CPU 在知道比较结果之前无法发出条件跳转指令。这被称为分支依赖关系（也称为控制依赖关系）。有关分支依赖关系的说明，请参见图 4.12。

4.2.6.1 推测执行

CPU 处理分支依赖关系的一种方法是通过一种称为推测执行（speculative execution）的技术，也称为分支预测（branch predict）。每当遇到分支指令时，CPU 都会尝试猜测将要执行哪个分支。它会继续从所选分支发出指令，希望自己的猜测是正确的。当然，直到依赖指令在流水线末端弹出之前，CPU 都无法确定自己的猜测是否正确。如果猜测最终是错误的，CPU 就会执行

Figure 4.12. A dependency between a comparison instruction and a conditional branch instruction is called a branch dependency.

```
    ; function preamble omitted for clarity...

    ; first, put the default into the return register
    mov  eax,dword ptr [defaultVal]

    mov  esi,dword ptr [b] ; check (b != 0)
    cmp  esi,0
    jz   SkipDivision

    mov  eax, dword ptr[a] ; divisor (a) must be in EDX:EAX
    cdq                    ; ... so sign-extend into EDX
    idiv esi               ; quotient lands in EAX

SkipDivision:
    ; function postamble omitted for clarity...
    ret                    ; EAX is the return value
```

The dependency here is between the `cmp` (compare) instruction and the `jz` (jump if equal to zero) instruction. The CPU cannot issue the conditional jump until it knows the results of the comparison. This is called a *branch dependency* (also known as a *control dependency*). See Figure 4.12 for an illustration of a branch dependency.

4.2.6.1 Speculative Execution

One way CPUs deal with branch dependencies is via a technique known as *speculative execution*, also known as *branch prediction*. Whenever a branch instruction is encountered, the CPU tries to *guess* at which branch is going to be taken. It continues to issue the instructions from the selected branch, in the hopes that its guess was correct. Of course, the CPU won't know for sure whether it guessed correctly until the dependent instruction pops out at the end of the pipeline. If the guess ends up being wrong, the CPU has executed

根本不应该执行的指令。因此，流水线必须刷新，并在正确分支的第一条指令处重新启动。这被称为分支惩罚。

CPU 能做的最简单的猜测就是假设分支永远不会被执行。CPU 只会继续按顺序执行指令，只有当猜测被证明错误时，才会将指令指针跳转到新的位置。这种方法对指令缓存友好，因为 CPU 总是优先选择指令最有可能在缓存中的分支。

另一种稍微高级一点的分支预测方法是假设后向分支总是被执行，而前向分支永远不会被执行。后向分支指的是在 while 或 for 循环末尾出现的分支，因此这类分支比前向分支更常见。

大多数高质量 CPU 都包含分支预测硬件，可以显著提高这些“静态”猜测的质量。分支预测器可以跟踪循环多次迭代中分支指令的结果，并发现有助于其在后续迭代中做出更准确猜测的模式。

PS3 游戏程序员不得不一直应对“分支”代码的糟糕性能，因为 Cell 处理器上的分支预测器实在太糟糕了。但 PS4 和 Xbox One 上的 AMD Jaguar CPU 拥有非常先进的分支预测硬件，因此游戏程序员在为 PS4 编写代码时可以稍微轻松一些。

4.2.6.2 谓词

减轻分支依赖影响的另一种方法是完全避免分支。再次考虑 SafeIntegerDivide() 函数，不过我们会对其进行一些修改，使其以浮点值而不是整数的形式工作：

```
float SafeFloatDivide(float a, float b, float d)
{
    return (b != 0.0f) ? a / b : d;
}
```

这个简单的函数根据条件测试 b != 0 的结果计算出两个答案中的一个。与其使用条件分支语句来返回这两个答案中的一个，不如让我们的条件测试生成一个位掩码，如果条件为假，则该掩码由全零 (0x0U) 组成；如果条件为真，则该掩码由全一 (0xFFFFFFFFU) 组成。然后，我们可以执行这两个分支，生成两个不同的答案。最后，我们使用该掩码来生成函数返回的最终答案。

instructions that shouldn't have been executed at all. So the pipeline must be *flushed* and restarted at the first instruction of the correct branch. This is called a *branch penalty*.

The simplest guess a CPU can make is to assume that branches are never taken. The CPU just keeps executing instructions in sequential order, and only jumps the instruction pointer to a new location when its guess is proven wrong. This approach is I-cache friendly, in the sense that the CPU always prefers the branch whose instructions are most likely to be in the cache.

Another slightly more advanced approach to branch prediction is to assume that backward branches are always taken and forward branches are never taken. A backward branch is the kind found at the end of a `while` or `for` loop, so such branches tend to be more prevalent than forward branches.

Most high-quality CPUs include *branch prediction* hardware that can improve the quality these "static" guesses significantly. A branch predictor can track the results of a branch instruction over multiple iterations of a loop and discover patterns that help it make better guesses on subsequent iterations.

PS3 game programmers had to deal with the poor performance of "branchy" code all the time, because the branch predictors on the Cell processor were frankly pretty terrible. But the AMD Jaguar CPU found in the PS4 and Xbox One has highly advanced branch prediction hardware, so game programmers can breathe a little easier when writing code for the PS4.

4.2.6.2 Predication

Another way to mitigate the effects of branch dependencies is to simply avoid branching altogether. Consider again the `SafeIntegerDivide()` function, although we'll modify it slightly to work in terms of floating-point values instead of integers:

```
float SafeFloatDivide(float a, float b, float d)
{
    return (b != 0.0f) ? a / b : d;
}
```

This simple function calculates one of two answers, depending on the results of the conditional test `b != 0`. Instead of using a conditional branch statement to return one of these two answers, we can instead arrange for our conditional test to generate a *bit mask* consisting of all zeros (0x0U) if the condition is false, and all ones (0xFFFFFFFFU) if it is true. We can then execute *both* branches, generating two alterate answers. Finally, we use the mask to produce the final answer that we'll return from the function.

以下伪代码说明了谓词背后的思想。（请注意，此代码无法直接运行。具体来说，您无法用无符号整数掩码浮点数并获得浮点数结果——您需要使用联合体在应用掩码时将浮点数的位模式重新解释为无符号整数。）

```
int SafeFloatDivide_pred(float a, float b, float d)
{
    // convert Boolean (b != 0.0f) into either 1U or 0U
    const unsigned condition = (unsigned)(b != 0.0f);

    // convert 1U -> 0xFFFFFFFFU
    // convert 0U -> 0x00000000U
    const unsigned mask = 0U - condition;

    // calculate quotient (will be QNaN if b == 0.0f)
    const float q = a / b;

    // select quotient when mask is all ones, or default
    // value d when mask is all zeros (NOTE: this won't
    // work as written -- you'd need to use a union to
    // interpret the floats as unsigned for masking)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

让我们仔细看看它是如何工作的：

• 测试 b != 0.0f 产生一个布尔值结果。我们通过简单的类型转换将其转换为无符号整数。结果要么是 1U （对应 true ），要么是 0U （对应 false ）。

• 我们将这个无符号结果从 0U 中减去，将其转换为位掩码。零减零仍然是零，零减一是 – 1，即
32 位无符号整数运算中的 0xFFFFFFFFU。

• 接下来，我们继续计算商。无论非零测试的结果如何，我们都会运行这段代码，从而避免任何分支依赖问题。

• 现在我们已经准备好两个答案：商 q 和默认值 d 。我们想应用掩码来“选择”其中一个值。但要做到这一点，我们需要将 q 和 d 的浮点位模式重新解释为无符号整数。在 C/C++ 中，实现此目的最便捷的方法是使用包含

The following pseudocode illustrates the idea behind predication. (Note that this code won't run as-is. In particular, you can't mask a `float` with an `unsigned int` and obtain a `float` result—you'd need to use a union to reinterpret the bit patterns of the `floats` as if they were unsigned integers when applying the mask.)

```
int SafeFloatDivide_pred(float a, float b, float d)
{
    // convert Boolean (b != 0.0f) into either 1U or 0U
    const unsigned condition = (unsigned)(b != 0.0f);

    // convert 1U -> 0xFFFFFFFFU
    // convert 0U -> 0x00000000U
    const unsigned mask = 0U - condition;

    // calculate quotient (will be QNaN if b == 0.0f)
    const float q = a / b;

    // select quotient when mask is all ones, or default
    // value d when mask is all zeros (NOTE: this won't
    // work as written -- you'd need to use a union to
    // interpret the floats as unsigned for masking)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

Let's take a closer look at how this works:

- The test `b != 0.0f` produces a `bool` result. We convert this into an `unsigned` integer by simply casting it. This results in either the value `1U` (corresponding to `true`) or `0U` (corresponding to `false`).
- We convert this unsigned result into a bit mask by subtracting it from `0U`. Zero minus zero is still zero, and zero minus one is −1, which is `0xFFFFFFFFU` in 32-bit unsigned integer arithmetic.
- Next, we go ahead and calculate the quotient. We run this code *regardless* of the result of the non-zero test, thereby side-stepping any branch dependency issues.
- We now have our two answers ready to go: The quotient `q` and the default value `d`. We want to apply the mask in order to "select" one or the other value. But to do this, we need to *reinterpret* the floating-point bit patterns of `q` and `d` *as if* they were unsigned integers. The most portable way to accomplish this in C/C++ is to use a union containing

两个成员，其中一个将 32 位值解释为 float ，另一个将其解释为 unsigned 。

- 掩码的应用方式如下：我们将商 q 与掩码进行按位与运算，如果掩码全为 1，则生成的位模式与 q 匹配；如果掩码全为 0，则生成的位模式与 q 匹配。我们将默认值 d 与掩码的补码进行按位与运算，如果掩码全为 1，则生成的位模式与 d 匹配；如果掩码全为 0，则生成的位模式与 d 匹配。最后，将这两个值进行按位或运算，最终选择 q 的值或 d 的值。

像这样使用掩码从两个可能值中选择一个，这种做法被称为“预测”，因为我们会运行两条代码路径（一条返回 a / b ，另一条返回 d ），但每条代码路径都基于测试结果 (a != 0)，并通过掩码进行预测。由于我们是在两个可能值中选择一个，因此这通常也被称为“选择”操作。

为了避免分支而费尽心思，这看起来似乎有些矫枉过正。事实也的确如此——它的实用性取决于分支指令相对于目标硬件上已断言替代方案的相对成本。当 CPU 的 ISA 提供用于执行选择操作的特殊机器语言指令时，断言才能真正发挥作用。例如，PowerPC ISA 提供了整数选择指令 isel 、浮点选择指令 fsel ，甚至还有 SIMD 矢量选择指令 vecsel ，它们的使用无疑可以提升基于 PowerPC 的平台（例如 PS3）的性能。

重要的是要意识到，只有当两个分支都能安全执行时，谓词才有效。在浮点数中执行除以零的运算会生成一个非数字（QNaN），但整数除以零会引发异常，导致游戏崩溃（除非被捕获）。这就是为什么我们在应用谓词之前先将此示例转换为浮点数。

4.2.7 超标量 CPU

我们在 4.2.1 节中描述的流水线 CPU 被称为标量处理器。这意味着它每个时钟周期最多可以执行一条指令。没错，在任何给定时刻都有多条指令“在运行”，但每个时钟周期只有一条新指令被发送到流水线。

并行的核心在于同时利用多个硬件组件。因此，将 CPU 吞吐量翻倍（至少理论上！）的一种方法是在芯片上复制大部分组件，这样每个时钟周期就可以启动两条指令。这被称为超标量架构。

two members, one of which interprets a 32-bit value as a `float`, the other of which interprets it as an `unsigned`.

- The mask is applied as follows: We bitwise AND the quotient `q` with the mask, producing a bit pattern that matches `q` if the mask is all ones, but is all zeros if the mask is all zeros. We bitwise AND the default value `d` with the *complement* of the mask, yielding all zeros if the mask is all ones, or the bit pattern of `d` if the mask is all zeros. Finally, we bitwise OR these two values together, effectively selecting *either* the value of `q` *or* the value of `d`.

The use of a mask to select one of two possible values like this is called *predication* because we run both code paths (the one that returns `a / b` and the one that returns `d`) but each code path is *predicated* on the results of the test `(a != 0)`, via the mask. Because we are selecting one of two possible values, this is also often called a *select* operation.

Going to all this trouble to avoid a branch may seem like overkill. And it can be—its usefulness depends on the relative cost of a branch versus the predicated alternative on your target hardware. Predication really shines when a CPU's ISAs provide special machine language instructions for performing a select operation. For example, the PowerPC ISA offers an integer select instruction `isel`, a floating-point select instruction `fsel`, and even a SIMD vector select instruction `vecsel`, and their use can definitely result in performance improvements on PowerPC based platforms (like the PS3).

It's important to realize that predication only works when both branches can be executed *safely*. Performing a divide by zero operation in floating-point generates a quiet not-a-number (QNaN), but an integer divide by zero throws an exception that will crash your game (unless it is caught). That's why we converted this example to floating-point before applying predication to it.

4.2.7 Superscalar CPUs

The pipelined CPU we described in Section 4.2.1 is what is called a *scalar* processor. This means that it can start executing at most one instruction per clock cycle. Yes, multiple instructions are "in flight" at any given moment, but only one *new* instruction is sent down the pipeline every clock cycle.

At its core, parallelism is about making use of multiple hardware components simultaneously. So one way to double the throughput of a CPU (at least in theory!) would be to *duplicate* most of the components on the chip, in such a way that *two* instructions could be launched each clock cycle. This is called a *superscalar* architecture.

图 4.13 流水线超标量 CPU 包含多个执行组件（ALU、FPU 和/或 VPU），由单个指令调度程序提供，该调度程序通常支持乱序执行和推测执行。

在超标量 CPU 中，片上存在两个（或更多）管理流水线 1 各级的电路实例。CPU 仍然从单个指令流中获取指令，但不是在每个时钟周期发出 IP 指向的一条指令，而是在每个时钟周期获取并调度接下来的两条指令。图 4.13 展示了双向超标量 CPU 中的硬件组件，图 4.14 追踪了十条指令（从“A”到“N”）在该 CPU 的两个并行流水线中的移动路径。

4.2.7.1 超标量设计的复杂性

实现超标量 CPU 并非像将两个相同的 CPU 内核“复制粘贴”到一块芯片上那么简单。虽然将超标量 CPU 设想为两条并行的指令流水线是合理的，但这两条流水线由一条指令流提供数据。因此，这些并行流水线的前端需要某种控制逻辑。与支持乱序执行的 CPU 一样，超标量 CPU 的控制逻辑会提前查看指令流，尝试识别指令之间的依赖关系，然后乱序执行指令，以减轻其影响。

除了数据和分支依赖关系外，超标量 CPU 还容易出现第三种依赖关系，即资源依赖关系。当两条或多条连续指令都需要

1 从技术上讲，流水线和超标量设计是两种独立的并行形式。流水线 CPU 不一定是超标量的。同样，超标量 CPU 也不一定需要流水线，尽管大多数 CPU 都是流水线的。

Figure 4.13. A pipelined superscalar CPU contains multiple execution components (ALUs, FPUs and/or VPUs) fed by a single instruction scheduler which typically supports out-of-order and speculative execution.

In a superscalar CPU, two (or more) instances of the circuitry that manages each stage of the pipeline[1] is present on-chip. The CPU still fetches instructions from a single instruction stream, but instead of issuing the one instruction pointed to by the IP during each clock cycle, the next *two* instructions are fetched and dispatched during each clock cycle. Figure 4.13 illustrates the hardware components found in a two-way superscalar CPU, and Figure 4.14 traces the path of ten instructions, “A” through “N,” as they move through this CPU’s two parallel pipelines.

4.2.7.1 Complexity of Superscalar Designs

Implementing a superscalar CPU isn’t quite as simple as “copying and pasting” two identical CPU cores onto a die. Although it’s reasonable to envision a superscalar CPU as two parallel instruction pipelines, these two pipelines are fed from a single instruction stream. Some some kind of control logic is therefore required at the front end of these parallel pipes. Just as on a CPU that supports out-of-order execution, a superscalar CPU’s control logic looks ahead in the instruction stream in an attempt to identify *dependencies* between instructions, and then issues instructions out of order in an attempt to mitigate their effects.

In addition to data and branch dependencies, a superscalar CPU is prone to a third kind of dependency known as a *resource dependency*. This kind of dependency arises when two or more consecutive instructions all require the

[1] Technically pipelining and superscalar designs are two independent forms of parallelism. A pipelined CPU needn’t be superscalar. Likewise, a superscalar CPU needn’t be pipelined, although the majority of them are.

钟循环	F_0	F_1	D_0	D_1	E_0	E_1	M_0	M_1	W_0	W_1
0	A	B								
1	C	D	A	B						
2	E	F	C	D	A	B				
3	G	H	E	F	C	D	A	B		
4	I	J	G	H	E	F	C	D	A	B
5	K	L	I	J	G	H	E	F	C	D
6	M	N	K	L	I	J	G	H	E	F

图 4.14. 在超标量流水线 CPU 上，七个时钟周期内执行 14 条指令“A”到“N”的最佳情况。

CPU 内相同的功能单元。例如，假设我们有一个超标量 CPU，它有两个整数 ALU，但只有一个浮点运算单元 (FPU)。这样的处理器能够在每个时钟周期发出两条整数算术指令。但是，如果在指令流中遇到两条浮点算术指令，它们就不能同时在同一个时钟周期发出，因为第二条指令（FPU）所需的资源已经被第一条指令占用。因此，超标量 CPU 上管理指令调度的控制逻辑比支持乱序执行的标量 CPU 上的控制逻辑更加复杂。

4.2.7.2 超标量和 RISC

双向超标量 CPU 所需的硅片面积大约是同类标量 CPU 设计的两倍。因此，为了释放晶体管，大多数超标量 CPU 都是精简指令集 (RISC) 处理器。RISC 处理器的 ISA 提供的指令集相对较小，每条指令都有非常明确的用途。更复杂的操作是通过构建这些简单指令的序列来执行的。相比之下，复杂指令集计算机 (CISC) 的 ISA 提供的指令种类要丰富得多，每条指令都能够执行更复杂的操作。

4.2.8 超长指令字（VLIW）

我们在 4.2.7.1 节中看到，超标量 CPU 包含高度复杂的指令调度逻辑。这些逻辑占用了 CPU 上宝贵的空间

Clock Cycle	F_0	F_1	D_0	D_1	E_0	E_1	M_0	M_1	W_0	W_1
0	A	B								
1	C	D	A	B						
2	E	F	C	D	A	B				
3	G	H	E	F	C	D	A	B		
4	I	J	G	H	E	F	C	D	A	B
5	K	L	I	J	G	H	E	F	C	D
6	M	N	K	L	I	J	G	H	E	F

Figure 4.14. Best-case execution of 14 instructions “A” through “N” on a superscalar pipelined CPU over seven clock cycles.

same functional unit within the CPU. For example, let’s imagine we have a superscalar CPU with two integer ALUs but only one FPU. Such a processor is capable of issuing two integer arithmetic instructions on each clock. But if two floating-point arithmetic instructions are encountered in the instruction stream, they cannot both be issued on the same clock cycle because the resource required by the second (the FPU) will already be in use by the first. As such, the control logic that manages instruction dispatch on a superscalar CPU is even more complex than that found on a scalar CPU that supports out-of-order execution.

4.2.7.2 Superscalar and RISC

A two-way superscalar CPU requires roughly two times the silicon real-estate of a comparable scalar CPU design. In order to free up transistors, most superscalar CPUs are therefore *reduced instruction set* (RISC) processors. The ISA of a RISC processor provides a comparatively small set of instructions, each with a very focused purpose. More complex operations are peformed by building up sequences of these simpler instructions. In contrast, the ISA of a *complex instruction set computer* (CISC) offers a much wider variety of instructions, each of which may be capable of performing more complex operations.

4.2.8 Very Long Instruction Word (VLIW)

We saw in Section 4.2.7.1 that a superscalar CPU contains highly complex instruction dispatch logic. This logic takes up valuable real-estate on the CPU

死。此外，在分析依赖关系并寻找乱序和/或超标量指令调度的机会时，CPU 只能在指令流中提前预测相对较少的指令。这限制了 CPU 执行动态优化的有效性。

实现指令级并行的一种更简单的方法是设计一个片上具有多个计算单元（ALU、FPU、VPU）的 CPU，但将向这些计算单元调度指令的任务完全交给程序员和/或编译器。这样，所有复杂的指令调度逻辑都可以被省去，而那些晶体管则可以用于实现更多的计算单元或更大的缓存。这样做的附带好处是，程序员和编译器应该能够比 CPU 更好地优化程序中的指令调度，因为它们可以从更宽的窗口（通常是整个函数的指令集）中选择要调度的指令。

为了允许程序员和/或编译器在每个时钟周期将指令分发到多个计算单元，指令字被扩展，使其包含两个或多个“槽”，每个槽对应芯片上的一个计算单元。例如，如果我们假设的 CPU 包含两个整数 ALU 和两个 FPU，则程序员或编译器需要能够在每个指令字中编码最多两个整数运算和两个浮点运算。我们称之为超长指令字 (VLIW) 设计。VLIW 架构如图 4.15 所示。

我们可以将 PlayStation 2 作为 VLIW 架构的一个具体示例：PS2 包含两个称为矢量单元（VU0 和 VU1）的协处理器，每个单元每个时钟周期能够发送两条指令。每个指令字由两个称为低位和高位的槽位组成。虽然已经开发出一些工具来帮助程序员将每时钟周期一条指令的程序转换为高效的每时钟周期两条指令的格式，但在用汇编语言手工编写代码时，有效地填充这两个槽位通常是一项挑战。

超标量和 VLIW 方法之间存在权衡。由于 VLIW 处理器无需像超标量 CPU 那样执行复杂的调度、乱序执行和分支预测逻辑，因此更加简单，因此可以比超标量处理器更高效地利用并行性。然而，将串行程序转换为能够充分利用 VLIW 并行性的形式可能非常困难。这使得程序员和/或编译器的工作更加困难。尽管如此，人们已经取得了一些进展来克服这些限制，包括可变宽度 VLIW 设计。例如，请参阅 http://researcher.watson。

die. Also, CPUs are only capable of looking ahead in the instruction stream by a relatively small number of instructions when analyzing dependencies and looking for opportunities for out-of-order and/or superscalar instruction dispatch. This limits the effectiveness of the dynamic optimizations a CPU can perform.

A somewhat simpler way to implement instruction level parallelism is to design a CPU that has multiple compute elements (ALUs, FPUs, VPUs) on-chip, but leaves the task of *dispatching* instructions to those compute elements entirely to the programmer and/or compiler. That way, all the complex instruction-dispatch logic can be eliminated, and those transistors devoted instead to implementing more compute elements or a larger cache. As a side benefit, programmers and compilers ought to be better at optimizing the dispatching of the instructions in their programs than the CPU could ever be, because they can select instructions for dispatch from a much wider window (typically an entire function's worth of instructions).

To allow programmers and/or compilers to dispatch instructions to multiple compute elements on each clock cycle, the instruction word is extended so that it contains two or more "slots," each corresponding to a compute element on the chip. For example, if our hypothetical CPU contained two integer ALUs and two FPUs, a programmer or compiler would need to be able to encode up to two integer and two floating-point operations within each instruction word. We call this a *very long instruction word* (VLIW) design. The VLIW architecture is illustrated in Figure 4.15.

We can look to the Playstation 2 for a concrete example of VLIW architecture: The PS2 contained two coprocessors called *vector units* (VU0 and VU1), each of which was capable of dispatching two instructions per clock cycle. Each instruction word was comprised of two slots called *low* and *high*. It was often a challenge to fill both slots effectively when hand-coding in assembly language, although tools were developed that helped programmers to convert a one-instruction-per-clock program into an efficient two-instructions-per-clock format.

There are trade-offs between the superscalar and VLIW approaches. Because it lacks the complex scheduling, out-of-order execution and branch prediction logic of a superscalar CPU, a VLIW processor is much simpler, and can therefore potentially make heavier use of parallelism than its superscalar counterparts. However, it can be very tough to transform a serial program into a form that takes full advantage of the parallelism in a VLIW. This makes the job of the programmer and/or compiler more difficult. That said, a number of advances have been made to overcome some of these limitations, including variable-width VLIW designs. For example, see http://researcher.watson.

图 4.15。流水线 VLIW CPU 架构由两个整数 ALU 和两个浮点 FPU 组成。每个超长指令字包含两个整数运算和两个浮点运算，这些运算会被分派到相应的功能单元。请注意，它没有超标量 CPU 中常见的复杂指令调度逻辑。

ibm.com/researcher/view_group_subpage.php?id=2834。

4.3 显式并行

显式并行旨在提高并发软件的运行效率。因此，所有显式并行硬件设计都允许并行处理多个指令流。我们将在下面列出一些常见的显式并行设计，粒度依次从最细粒度的超线程到最粗粒度的云计算。

4.3.1 超线程

正如我们在 4.2.5.2 节中看到的，一些流水线 CPU 能够乱序执行指令，以减少流水线停顿。通常，流水线 CPU 会按程序顺序执行指令；但有时，由于依赖于正在执行的指令，指令流中的下一条指令无法执行。这会创建一个延迟槽，理论上可以将另一条指令放入该延迟槽中。乱序执行 CPU 可以“向前看”指令流，并选择一条指令以乱序方式放入这样的延迟槽中。

Figure 4.15. A pipelined VLIW CPU architecture consisting of two integer ALUs and two floating-point FPUs. Each very long instruction word consists of two integer and two floating-point operations, which are dispatched to the corresponding functional units. Notice the absence of the complex instruction scheduling logic that would be present in a superscalar CPU.

ibm.com/researcher/view_group_subpage.php?id=2834.

4.3 Explicit Parallelism

Explicit parallelism is designed to make concurrent software run more efficiently. Hence all explicitly parallel hardware designs permit more than one instruction stream to be processed in parallel. We'll list a few common explicitly parallel designs below, increasing in granularity from *hyperthreading* at the most fine-grained end of the spectrum to *cloud computing* at the most coarse-grained end.

4.3.1 Hyperthreading

As we saw in Section 4.2.5.2, some pipelined CPUs are capable of issuing instructions *out of order* as a means of reducing pipeline stalls. Normally a pipelined CPU executes instructions in program order; but sometimes the next instruction in the instruction stream cannot be issued due to a dependency on an in-flight instruction. This creates a *delay slot* into which another instruction could theoretically be issued. An OOO CPU can "look ahead" in the instruction stream and select an instruction to issue out-of-order into such a delay slot.

图 4.16。一个超线程 CPU，包含两个前端（每个前端包含一个取指/解码单元和一个寄存器文件），但只有一个后端，其中包含 ALU、FPU、内存控制器、一级缓存和乱序指令调度器。调度器将来自两个前端线程的指令发送到共享的后端组件。

由于只有单个指令流，CPU 在选择将指令发送到延迟槽时，选项会受到一定限制。但如果 CPU 可以同时从两个独立的指令流中选择指令，情况会怎样呢？这就是超线程 (HT) CPU 内核背后的原理。

从技术角度来看，HT 内核由两个寄存器文件和两个指令解码单元组成，但只有一个用于执行指令的“后端”，以及一个共享的 L1 缓存。这种设计使 HT 内核能够运行两个独立的线程，同时由于共享后端和 L1 缓存，所需的晶体管数量比双核 CPU 更少。当然，这种硬件组件的共享也会导致指令吞吐量低于同类双核 CPU，因为线程会争用这些共享资源。图 4.16 展示了典型超线程 CPU 设计中的关键组件。

4.3.2 多核 CPU

CPU 核心可以定义为一个独立的单元，能够执行至少一个指令流中的指令。因此，我们迄今为止研究过的所有 CPU 设计都可以称为“核心”。当单个 CPU 芯片上包含多个核心时，我们称之为多核 CPU。

Figure 4.16. A hyperthreaded CPU containing two front ends (each consisting of a fetch/decode unit and a register file), but with a single back end containing ALUs, FPUs, a memory controller, an L1 cache, and an out-of-order instruction scheduler. The scheduler issues instructions from both front-end threads to the shared back-end components.

With only a single instruction stream, the CPU's options are somewhat limited when selecting an instruction to issue into a delay slot. But what if the CPU could select its instructions from two separate instruction streams at once? This is the principle behind a *hyperthreaded* (HT) CPU core.

Technically speaking, an HT core consists of two register files and two instruction decode units, but with a single "back end" for executing instructions, and a single shared L1 cache. This design enables an HT core to run two independent threads, while requiring fewer transistors than a dual core CPU, thanks to the shared back end and L1 cache. Of course, this sharing of hardware components also results in lower instruction throughput relative to a comparable dual core CPU, because the threads contend for these shared resources. Figure 4.16 illustrates the key components in a typical hyperthreaded CPU design.

4.3.2 Multicore CPUs

A CPU *core* can be defined as a self-contained unit capable of executing instructions from at least one instruction stream. Every CPU design we've looked at until now could therefore qualify as a "core." When more than one core is included on a single CPU die, we call it a *multicore* CPU.

图 4.17.一个简单的多核 CPU 设计。

每个核心内的具体设计可以是我们迄今为止研究过的任何设计 - 每个核心可能采用简单的串行设计、流水线设计、超标量架构、VLIW 设计，或者可能是超线程核心。

图 4.17 展示了一个多核 CPU 设计的简单示例。

PlayStation 4 和 Xbox One 游戏机均搭载多核 CPU。每款游戏机都包含一个加速处理单元 (APU)，该单元由两个四核 AMD Jaguar 模块组成，与 GPU、内存控制器和视频编解码器集成在一个芯片上。（八个核心中，七个可供游戏应用程序使用。然而，第七个核心上大约一半的带宽保留给操作系统使用。）Xbox One X 也搭载一个八核 APU，但其核心基于与 AMD 合作开发的专有技术，而不是像其前代产品那样基于 Jaguar 微架构。图 4.18 展示了 PS4 硬件架构的框图，图 4.19 展示了 Xbox One 硬件架构的框图。

4.3.3 对称与非对称多处理

并行计算平台的对称性与操作系统如何处理机器中的 CPU 核心有关。在对称多处理 (SMP) 中，机器中可用的 CPU 核心（由超线程、多核 CPU 或单个 CPU 上的多个 CPU 的任意组合提供）

Figure 4.17. A simple multicore CPU design.

The specific design within each core can be any of the designs we've looked at thus far—each core might employ a simple serial design, a pipelined design, a superscalar architecture, a VLIW design, or might be a hyperthreaded core. Figure 4.17 illustrates a simple example of a multicore CPU design.

The PlayStation 4 and Xbox One game consoles both contain multicore CPUs. Each contains an *accelerated processing unit* (APU) consisting of two quad-core AMD Jaguar modules, integrated onto a single die with a GPU, memory controller and video codec. (Of these eight cores, seven are available for use by game applications. However, roughly half of the bandwidth on the seventh core is reserved for operating system use.) The Xbox One X also contains an eight-core APU, but its cores are based on proprietary technology developed in partnership with AMD, rather than on the Jaguar microarchitecture like its predecessor. Figure 4.18 shows a block diagram of the PS4 hardware architecture, and Figure 4.19 presents a block diagram of the Xbox One hardware architecture.

4.3.3 Symmetric versus Asymmetric Multiprocessing

The *symmetry* of a parallel computing platform has to do with how the CPU cores in the machine are treated by the operating system. In *symmetric multiprocessing* (SMP), the available CPU cores in the machine (provided by any combination of hyperthreading, multicore CPUs or multiple CPUs on a single

图 4.18. PS4 架构的简化视图。

主板（例□□如，CPU）在设计和 ISA 方面是同构的，并且操作系统会平等对待它们。任何线程都可以被调度到任何核心上执行。（不过请注意，在这样的系统中，可以为线程指定亲和性，使其更有可能（甚至保证）被调度到特定核心上。）

PlayStation 4 和 Xbox One 就是 SMP 的例子。这两款游戏机都包含八个核心，其中七个可供程序员使用，应用程序可以在任何可用核心上运行线程。

在非对称多处理 (AMP) 中，CPU 核心不一定是同构的，操作系统也不会平等对待它们。在 AMP 中，一个"主"CPU 核心通常运行操作系统，而其他核心则被视为"从属"核心，由主核心将工作负载分配给这些核心。

Figure 4.18. Simplified view of the PS4's architecture.

motherboard) are homogeneous in terms of design and ISA, and are treated equally by the operating system. Any thread can be scheduled to execute on any core. (Note, however, that it is possible in such systems to specify an *affinity* for a thread, causing it to be more likely, or even guaranteed, to be scheduled on a particular core.)

The PlayStation 4 and the Xbox One are examples of SMP. Both of these consoles contain eight cores, of which seven are available for use by the programmer, and the application is free to run threads on any of the available cores.

In *asymmetric multiprocessing* (AMP), the CPU cores are not necessarily homogeneous, and the operating system does not treat them equally. In AMP, one "master" CPU core typically runs the operating system, and the other cores are treated as "slaves" to which workloads are distributed by the master core.

图 4.19.Xbox One 架构的简化视图。

PlayStation 3 中使用的单元宽带引擎 (CBE) 就是 AMP 的一个例子；它采用了一个基于 PowerPC ISA 的主 CPU，称为“电源处理单元”(PPU)，以及八个基于完全不同 ISA 的协处理器，称为“协同处理单元”(SPU)。（有关 PS3 硬件架构的更多信息，请参阅第 3.5.5 节。）

4.3.4 分布式计算

实现计算并行性的另一种方法是利用多台独立计算机协同工作。这通常被称为分布式计算。构建分布式计算系统的方法有很多，包括：

- 计算机集群，
- 网格计算，以及

Figure 4.19. Simplified view of the Xbox One's architecture.

The *cell broadband engine* (CBE) used in the PlayStation 3 is an example of AMP; it employs a main CPU known as the "power processing unit" (PPU) which is based on the PowerPC ISA, along with eight coprocessors known as "synergystic processing units" (SPUs) which are based around a completely different ISA. (See Section 3.5.5 for more on the PS3's hardware architecture.)

4.3.4 Distributed Computing

Yet another way to achieve parallelism in computing is to make use of multiple stand-alone computers working in concert. This is known as *distributed computing* in the most general sense. There are various ways to architect a distributed computing system, including:

- computer clusters,
- grid computing, and

图 4.20。内核和设备驱动程序直接位于硬件之上，并以特权模式运行。所有其他操作系统软件和所有用户程序都在内核和驱动程序层之上实现，并以某种受限的用户模式运行。

- 云计算。

在本书中，我们将专注于单台计算机内的并行性，但您可以通过在线搜索上述术语来阅读有关分布式计算的更多信息。

4.4 操作系统基础知识

现在我们已经对并行计算机硬件的基础知识有了深入的了解，让我们将注意力转向操作系统提供的使并发编程成为可能的服务。

4.4.1 内核

现代操作系统处理各种类型的任务，涵盖各种粒度。这些任务的范围很广，从处理键盘和鼠标事件或调度程序以实现抢占式多任务处理（一端），到管理打印机队列或网络堆栈（另一端）。操作系统的“核心”——处理所有最基本、最底层操作的部分——称为内核。操作系统的其余部分以及所有用户程序都构建在内核提供的服务之上。该架构如图 4.20 所示。

4.4.1.1 内核模式与用户模式

内核及其设备驱动器运行在一种称为保护模式、特权模式或内核模式的特殊模式下，而系统中的所有其他程序（包括操作系统中不属于内核的所有其他部分）则运行在用户模式下。顾名思义，在特权模式下运行的软件

Figure 4.20. The kernel and device drivers sit directly on top of the hardware, and run in privileged mode. All other operating system software and all user programs are implemented on top of the kernel and driver layer, and run in a somewhat restricted user mode.

- cloud computing.

We'll focus exclusively on parallelism within a single computer in this book, but you can read more about distributed computing by searching for the above terms online.

4.4 Operating System Fundamentals

Now that we have a solid understanding of the basics of parallel computer hardware, let's turn our attention to the services provided by the operating system that make concurrent programming possible.

4.4.1 The Kernel

Modern operating systems handle a wide variety of tasks, across a wide range of granularities. These range from the handling of keyboard and mouse events or the scheduling of programs for *preemptive multitasking* at one end of the spectrum, to managing a printer queue or the network stack at the other end. The "core" of the operating system—the part that handles all of the most fundamental and lowest-level operations—is called the *kernel*. The rest of the operating system, and all user programs, are built atop the services provided by the kernel. This architecture is illustrated in Figure 4.20.

4.4.1.1 Kernel Mode versus User Mode

The kernel and its device drives run in a special mode called *protected mode*, *privileged mode* or *kernel mode*, while all other programs in the system (including all other parts of the operating system that aren't part of the kernel) operate in *user mode*. As the name suggests, software running in privileged mode has

图 4.21 CPU 保护环示例，显示四个环。内核在 Ring 0 中运行，设备驱动程序在 Ring 1 中运行，具有 I/O 权限的可信程序在 Ring 2 中运行，所有其他用户程序在 Ring 3 中运行。

系统模式可以完全访问计算机中的所有硬件，而用户模式软件则受到各种限制，以确保整个计算机系统的稳定性。在用户模式下运行的软件只能通过特殊的内核调用（请求内核代表用户程序执行低级操作）来访问低级服务。这确保了程序不会无意或恶意地破坏系统的稳定性。

在实际中，操作系统可能会实现多个保护环。内核运行在 Ring 0 中，这是最受信任的 Ring，拥有系统内所有可能的权限。设备驱动程序可能运行在 Ring 1 中，具有 I/O 权限的受信任程序可能运行在 Ring 2 中，而所有其他“不受信任”的用户程序则运行在 Ring 3 中。但这仅仅是一个例子——不同 CPU 和操作系统的 Ring 数量各不相同，子系统在各个 Ring 中的分配也各不相同。保护环的概念如图 4.21 所示。

4.4.1.2 内核模式权限

内核模式（ring 0）软件可以访问 CPU ISA 定义的所有机器语言指令。这其中包括一组强大的指令，称为特权指令。这些特权指令可能允许修改某些通常被禁止的寄存器（例如，控制虚拟内存映射，或屏蔽和取消屏蔽中断）。它们也可能允许访问某些内存区域，或允许其他常规操作。

Figure 4.21. Example of CPU protection rings, showing four rings. The kernel runs in ring 0, device drivers run in ring 1, trusted programs with I/O permissions run in ring 2, and all other user programs run in ring 3.

full access to all of the hardware in the computer, whereas user mode software is *restricted* in various ways in order to ensure stability of the computer system as a whole. Software running in user mode can access low-level services only by making a special *kernel call*—a request for the kernel to perform a low-level operation on the user program's behalf. This ensures that a program can't inadvertently or maliciously destabilize the system.

In practice, operating systems may implement multiple *protection rings*. The kernel runs in ring 0, which is the most trusted ring and has all possible privileges within the system. Device drivers might run in ring 1, trusted programs with I/O permissions might run in ring 2, while all other "untrusted" user programs run in ring 3. But this is just one example—the number of rings varies from CPU to CPU and from OS to OS, as does the assignment of subsystems to the various rings. The protection ring concept is illustrated in Figure 4.21.

4.4.1.2 Kernel Mode Privileges

Kernel mode (ring 0) software has access to *all* of the machine language instructions defined by the CPU's ISA. This includes a powerful subset of instructions called *privileged* instructions. These privileged instructions might allow certain registers to be modified that are normally off-limits (e.g., to control virtual memory mapping, or to mask and unmask interrupts). Or they might allow certain regions of memory to be accessed, or allow other nor-

执行的操作通常受到严格限制。英特尔 x86 处理器上的特权指令示例包括 wrmsr（写入特定型号的寄存器）和 cli（清除中断）。通过将这些强大的指令限制在内核等“受信任”的软件范围内使用，可以提高系统的稳定性和安全性。

利用这些特权ML指令，内核可以实现安全措施。例如，内核通常会锁定某些虚拟内存页面，以防止用户程序写入。内核的软件及其所有内部记录数据都保存在受保护的内存页面中。这确保了用户程序不会破坏内核，从而导致整个系统崩溃。

4.4.2 中断

中断是发送给 CPU 的信号，用于通知其重要的低级事件，例如键盘上的按键、来自外围设备的信号或计时器到期。当此类事件发生时，会发出中断请求 (IRQ)。如果操作系统希望响应该事件，它会暂停（中断）正在进行的所有处理，并调用一种称为中断服务例程 (ISR) 的特殊函数。ISR 函数会执行某些操作来响应该事件（理想情况下，它会尽快执行），然后将控制权返回到中断发出之前正在运行的程序。

中断有两种：硬件中断和软件中断。
通过在 CPU 的某个引脚上施加非零电压来请求硬件中断。硬件中断可能由键盘或鼠标等设备触发，也可能由主板或 CPU 内部的周期性定时器电路触发。由于硬件中断是由外部设备触发的，因此它随时可能发生，甚至可能在执行 CPU 指令的过程中发生。因此，在硬件中断物理触发和 CPU 处于合适的处理状态之间可能会存在微小的延迟。

软件中断由软件触发，而不是由 CPU 引脚上的电压触发。它的基本作用与硬件中断相同，即中断 CPU 的运行并调用服务例程。软件中断可以通过执行“中断”机器语言指令明确触发。或者，软件中断也可能由 CPU 在运行软件时检测到的错误情况触发——这些情况被称为陷阱，有时也被称为异常（但后者不应与语言级异常处理混淆）。例如，如果一个 ALU

mally restricted operations to be performed. Examples of privileged instructions on the Intel x86 processor include `wrmsr` (write to model-specific register) and `cli` (clear interrupts). By restricting the use of these powerful instructions only to “trusted” software like the kernel, system stability and security is improved.

With these privileged ML instructions, the kernel can implement security measures. For example, the kernel typically locks down certain pages of virtual memory so that they cannot be written to by a user program. Both the kernel’s software and all of its internal record-keeping data are kept in protected memory pages. This ensures that a user program won’t stomp on the kernel, and thereby crash the entire system.

4.4.2 Interrupts

An *interrupt* is a signal sent to the CPU in order to notify it of an important low-level event, such as a keypress on the keyboard, a signal from a peripheral device, or the expiration of a timer. When such an event occurs, an *interrupt request* (IRQ) is raised. If the operating system wishes to respond to the event, it pauses (interrupts) whatever processing had been going on, and calls a special kind of function called an *interrupt service routine* (ISR). The ISR function performs some operation in response to the event (ideally doing so as quickly as possible) and then control is returned back to whatever program had been running prior to the interrupt having been raised.

There are two kinds of interrupt: *hardware interrupts* and *software interrupts*. A hardware interrupt is requested by placing a non-zero voltage onto one of the pins of the CPU. Hardware interrupts might be raised by devices such as a keyboard or mouse, or by a periodic timer circuit on the motherboard or within the CPU itself. Because it’s triggered by an external device, a hardware interrupt can happen at any time—even right in the middle of executing a CPU instruction. As such, there may be a tiny delay between the moment when a hardware interrupt is physically raised and when the CPU is in a suitable state to handle it.

A software interrupt is triggered by software rather than by a voltage on a CPU pin. It has the same basic effect as a hardware interrupt, in that it causes the operation of the CPU to be interrupted and a service routine to be called. A software interrupt can be triggered explicitly by executing an “interrupt” machine language instruction. Or one may be triggered in response to an erroneous condition detected by the CPU while running a piece of software—these are called *traps* or sometimes *exceptions* (although the latter term should not be confused with language-level exception handling). For example, if an ALU is

如果被指示执行除零运算，则会引发软件中断。操作系统通常会通过使相关程序崩溃并生成核心转储文件来处理此类中断。但是，连接到该程序的调试器可能会捕获此中断，并导致程序中断并进入调试器进行检查。

4.4.3 内核调用

为了使用户软件能够执行特权操作（例如，映射或取消映射虚拟内存系统中的物理内存页面，或访问原始网络套接字），用户程序必须向内核发出请求。内核会以安全的方式代表用户程序执行该操作。此类请求称为 内核调用 或 系统调用 。

在大多数系统上，内核调用是通过软件中断完成的。2 在中断触发的系统调用的情况下，用户程序将任何输入参数放在特定位置（内存或寄存器中），然后发出带有整数参数的“软件中断”指令，该整数参数指定正在请求哪个内核操作。这会导致 CPU 进入具有提升权限的模式，保持调用程序的状态，然后导致调用适当的内核中断服务例程。假设内核允许请求继续，它将执行请求的操作（在特权模式下），然后将控制权返回给调用者（首先恢复其执行状态）。从用户模式程序切换到内核是上下文切换的一个示例。有关上下文切换的更多信息，请参见第 4.4.6.5 节。

在大多数现代操作系统中，用户程序不会手动执行软件中断或系统调用指令（例如通过内联汇编代码）。那样会很混乱，而且容易出错。相反，用户程序会调用内核 API 函数，该函数会整理参数并触发软件中断。这就是为什么从用户程序的角度来看，系统调用看起来像是常规函数调用。

4.4.4 抢占式多任务

最早的小型计算机和个人计算机每次只能运行一个程序。它们本质上是串行计算机，能够从单个指令流中读取程序，并一次执行该指令流中的一条指令。当时的磁盘操作系统（DOS）并不多见。

2 在某些系统上，使用 call 指令的特殊变体来调用内核。例如，在 MIPS 处理器上，此指令名为 syscall 。

instructed to perform a divide-by-zero operation, a software interrupt will be raised. The operating system normally handles such interrupts by crashing the program in question and producing a *core dump* file. However, a debugger attached to the program could catch this interrupt and instead cause the program to break into the debugger for inspection.

4.4.3 Kernel Calls

In order for user software to perform a privileged operation, such as mapping or unmapping physical memory pages in the virtual memory system or accessing a raw network socket, the user program must make a request to the kernel. The kernel responds by performing the operation in a safe manner on behalf of the user program. Such a request is called a *kernel call* or *system call*.

On most systems, calling into the kernel is accomplished via a software interrupt.[2] In the case of an interrupt-triggered system call, the user program places any input arguments in a specific place (either in memory or in registers) and then issues a “software interrupt” instruction with an integer argument that specifies which kernel operation is being requested. This causes the CPU to be put into a mode with elevated privileges, staves the state of the calling program, and then causes the appropriate kernel interrupt service routine to be called. Presuming that the kernel allows the request to proceed, it performs the requested operation (in privileged mode) and then control is returned to the caller (after first restoring its execution state). This switch from a user mode program into the kernel is an example of *context switching*. See Section 4.4.6.5 for more on context switching.

On most modern operating systems, the user program doesn’t execute a software interrupt or system call instruction manually, for example via inline assembly code. That would be messy and error-prone. Instead, a user program calls a *kernel API* function, which in turn marshalls the arguments and trips the software interrupt. This is why system calls appear to be regular function calls from the point of view of the user program.

4.4.4 Preemptive Multitasking

The earliest minicomputers and personal computers ran a single program at a time. They were inherently *serial* computers, capable of reading a program from a single instruction stream, and executing one instruction from this stream at a time. The *disk operating systems* (DOS) in those days weren’t much

[2] On some systems, a special variant of the `call` instruction is used to call into the kernel. For example, on MIPS processors this instruction is named `syscall`.

图 4.22。Apple II 电脑上运行的三个全屏程序。Apple II 上的程序始终是全屏的，因为它一次只能运行一个程序。从左到右：Copy II Plus、AppleWorks 文字处理器和 The Locksmith。

不仅仅是美化设备驱动程序，它还允许程序与磁带、软盘和硬盘驱动器等设备进行交互。整台计算机将一次只运行一个程序。图 4.22 展示了在 Apple II 计算机上运行的几个全屏程序。

随着操作系统和计算机硬件变得越来越先进，在串行计算机上同时运行多个程序成为可能。在共享大型计算机系统上，一种称为多道程序设计的技术允许一个程序运行时另一个程序正在等待外围设备满足耗时的请求。经典 Mac OS 和 Windows NT 和 Windows 95 之前的 Windows 版本使用一种称为协作式多任务处理的技术，其中机器上一次只能运行一个程序，但每个程序都会定期放弃 CPU，以便其他程序有机会运行。通过这种方式，每个程序最终都会获得一段定期的 CPU 时间。从技术上讲，这种技术称为时分复用 (TDM) 或时间多线程 (TMT)。

非正式地，它被称为时间分片。
协作式多任务处理存在一个大问题：时间片机制需要系统中所有程序的协作。如果一个“流氓”程序未能定期让步给其他程序，它可能会耗费所有 CPU 时间。PDP-6 Monitor 和 Multics 操作系统通过引入一种称为抢占式多任务处理的技术解决了这个问题。这项技术后来被 UNIX 操作系统及其所有变体以及 Mac OS 和 Windows 的后续版本所采用。

在抢占式多任务中，程序仍然通过时间分片共享 CPU。
然而，程序的调度是由操作系统控制的，而不是通过程序之间的协作。因此，每个程序

Figure 4.22. Three full-screen programs running on the Apple II computer. Programs on the Apple II were always full-screen because it could only run one program at a time. From left to right: Copy II Plus, the AppleWorks word processor, and The Locksmith.

more than glorified device drivers, allowing programs to interface with devices such as tape, floppy and hard disk drives. The entire computer would be devoted to running a single program at a time. Figure 4.22 shows a few full-screen programs running on an Apple II computer.

As operating systems and computer hardware became more advanced, it became possible to run more than one program on a serial computer at a time. On shared mainframe computer systems, a technique known as *multiprogramming* would allow one program to run while another was waiting for a time-consuming request to be satisifed by a peripheral device. Classic Mac OS and versions of Windows prior to Windows NT and Windows 95 used a technique known as *cooperative multitasking*, in which only one program would be running on the machine at a time, but each program would regularly *yield* the CPU so that another program could get a chance to run. In this way, each program ended up with a periodic “slice” of CPU time. Technically, this technique is known as *time division multiplexing* (TDM) or *temporal multithreading* (TMT). Informally it’s called *time-slicing*.

Cooperative multitasking suffered from one big problem: Time-slicing required the cooperation of each and every program in the system. One “rogue” program could consume all of the CPU’s time if it failed to yield to other programs periodically. The PDP-6 Monitor and Multics operating systems solved this problem by introducing a technique known as *preemptive multitasking*. This technology was later adopted by the UNIX operating system and all of its variants, along with later versions of Mac OS and Windows.

In preemptive multitasking, programs still share the CPU by time-slicing. However the scheduling of programs is controlled by the operating system, not via cooperation between the programs themselves. As a result, each pro-

gram 在 CPU 上获得一个规律、一致且可靠的时间片。允许某个特定程序在 CPU 上运行的时间片有时被称为该程序的“量子”。为了实现抢占式多任务处理，操作系统会响应定时硬件中断，以便在系统上运行的不同程序之间定期进行上下文切换。我们将在下一节 (4.4.6.5) 中更深入地了解上下文切换的工作原理。

这里需要注意的是，即使在多核机器上也会使用抢占式多任务处理，因为通常线程数大于核心数。例如，如果我们有 100 个线程，但只有 4 个 CPU 核心，那么内核就会使用抢占式多任务处理，在每个核心上的 25 个线程之间进行时间片分配。

4.4.5 流程

进程是操作系统管理可执行文件（Windows 上为 .exe，Linux 上为 .elf）中程序运行实例的方式。进程仅在其程序实际运行时存在——当程序实例退出、被终止或崩溃时，操作系统会销毁与该实例关联的进程。计算机系统上可以随时运行多个进程。这可能包括同一程序的多个实例。

程序员通过操作系统提供的 API 与进程交互。这些 API 的细节因操作系统而异，但关键概念大致相同。本书不打算全面讨论任何一种操作系统的进程 API，但为了阐明这些概念，我们将主要关注类 UNIX 操作系统（例如 Linux、BSD 和 MacOS）的 API 风格。但我们会记录 Windows 或游戏机操作系统与类 UNIX 进程 API 的核心理念存在显著差异的情况。

4.4.5.1 流程剖析

本质上，一个过程包括：

- 进程 ID (PID)，用于在操作系统中唯一标识该进程；
- 一组权限，例如哪个用户“拥有”每个流程以及它属于哪个用户组；
- 对进程父进程的引用（如果有），
- 包含进程对物理内存“视图”的虚拟内存空间（有关更多信息，请参阅第 4.4.5.2 节）；

gram gets a regular, consistent and reliable time slice on the CPU. The time slice during which one particular program is allowed to run on the CPU is sometimes called the program's *quantum*. To implement preemptive multitasking, the operating system responds to a regularly-timed hardware interrupt in order to periodically *context switch* between the different programs running on the system. We'll see how context switching works in more depth in the next section (4.4.6.5).

We should note here that preemptive multitasking is used even on multicore machines, because typically the number of threads is greater than the number of cores. For example, if we were to have 100 threads and only four CPU cores, then the kernel would use preemptive multitasking to time-slice between 25 threads on each core.

4.4.5 Processes

A *process* is the operating system's way of managing a *running instance* of a program contained in an *executable file* (.exe on Windows, .elf on Linux). A process only exists while its program is actually running—when an instance of a program exits, is killed, or crashes, the OS destroys the process associated with that instance. Multiple processes can be running on a computer system at any given time. This might include multiple instances of the *same* program.

Programmers interact with processes via an API provided by the operating system. The details of this API differ from OS to OS, but the key concepts are roughly the same across all of them. A complete discussion of any one operating system's process API is beyond the scope of this book, but for the purposes of illustrating the concepts we'll focus primarily on the API style of UNIX-like operating systems such as Linux, BSD and MacOS. But we'll make note of situations in which Windows or game console operating systems deviate significantly from the core ideas of a UNIX-like process API.

4.4.5.1 Anatomy of a Process

Under the hood, a process consists of:

- a *process id (PID)* that uniquely identifies the process within the operating system;
- a set of *permissions*, such as which user "owns" each process and to which user group it belongs;
- a reference to the process's *parent process*, if any,
- a *virtual memory space* containing the process's "view" of physical memory (see Section 4.4.5.2 for more information);

- 所有定义的环境变量的值；
- 进程正在使用的所有打开文件句柄的集合；
- 进程的当前工作目录，
- 用于管理系统中进程之间的同步和通信的资源，例如消息队列、管道和信号量；
- 一个或多个线程。

线程封装了单个机器语言指令流的运行实例。默认情况下，一个进程包含一个线程。但正如我们将在 4.4.6 节深入讨论的那样，一个进程内可以创建多个线程，从而允许多个指令流并发运行。内核会调度系统中所有线程（来自所有当前正在运行的进程）在可用的核心上运行。当线程数量超过核心数量时，内核会使用抢占式多任务在线程之间进行时间片分配。

这里需要强调的是，线程是操作系统中程序执行的基本单位，而不是进程。进程仅仅提供了一个供其线程运行的环境，包括虚拟内存映射和一组资源，这些资源由该进程内的所有线程使用和共享。每当一个线程被调度到某个核心上运行时，它的进程就会变为活动状态，并且该进程的资源和环境在线程运行时可供其使用。因此，当我们说一个线程在某个核心上运行时，请记住它始终在一个进程的上下文中运行。

4.4.5.2 进程的虚拟内存映射

你可能还记得，在 3.5.2 节中，程序通常不会直接使用物理内存地址。[3] 相反，程序会通过虚拟地址访问内存，然后 CPU 和操作系统会协作将这些虚拟地址重新映射到物理地址。我们之前提到，虚拟地址到物理地址的重新映射是通过称为页面的连续地址块进行的，而操作系统使用页表将虚拟页面索引映射到物理页面索引。

每个进程都有自己的虚拟页表。这意味着每个进程都有自己定制的内存视图。这是操作系统提供安全稳定的执行环境的主要方式之一。两个进程无法破坏彼此的内存，因为一个进程拥有的物理页面根本不会映射到另一个进程的地址空间（除非它们明确共享页面）。此外，另一个进程拥有的页面

[3] 用户程序总是按照虚拟内存地址来工作，但内核可以直接使用物理地址来工作。

- the values of all defined *environment variables*;
- the set of all open *file handles* in use by the process;
- the current *working directory* for the process,
- resources for managing *synchronization* and *communication* between processes in the system, such as message queues, pipes, and semaphores;
- one or more *threads*.

A *thread* encapsulates a running instance of a single stream of machine language instructions. By default, a process contains a single thread. But as we'll discuss in depth in Section 4.4.6, more than one thread can be created within a process, allowing more than one instruction stream to run *concurrently*. The kernel schedules all of the threads in the system (from all currently-running processes) to run on the available cores. It uses preemptive multitasking to time-slice between threads when there are more threads than there are cores.

We should stress here that threads are the *fundamental unit of program execution* within an operating system, not processes. A process merely provides an *environment* within which its thread(s) can run, including a virtual memory map and a set of *resources* that are used by and shared between all threads within that process. Whenever a thread is scheduled to run on a core, its process becomes active, and that process's resources and environment become available for use by the thread while it runs. So when we say that a *thread* is running on a core, remember that it is always doing so within the *context* of exactly one process.

4.4.5.2 Virtual Memory Map of a Process

You'll recall from Section 3.5.2 that a program generally never works directly with physical memory addresses.[3] Rather, the program accesses memory in terms of *virtual addresses*, and the CPU and operating system cooperate to *remap* these virtual addresses into physical addresses. We said that the remapping of virtual to physical addresses happens in terms of contiguous blocks of addresses called *pages*, and that a *page table* is used by the OS to map virtual page indices to physical page indices.

Every process has its own virtual page table. This means that every process has its own custom *view* of memory. This is one of the primary ways in which the operating system provides a secure and stable execution environment. Two processes cannot corrupt each other's memory, because the physical pages owned by one process are simply not mapped into the other process's address space (unless they explicitly share pages). Also, pages owned by the

[3] User programs always work in terms of virtual memory addresses, but the kernel can work directly with physical addresses.

内核受到保护，不会受到用户进程无意或故意的破坏，因为它们被映射到称为内核空间的特殊地址范围，该范围只能由在内核模式下运行的代码访问。

进程的虚拟页表有效地定义了其内存映射。内存映射通常包含：

- 从程序的可执行文件中读取的文本、数据和BSS部分；
- 程序使用的任何共享库（DLL、PRX）的视图；
- 每个线程的调用堆栈；
- 称为堆的内存区域，用于动态内存分配；
- 可能与其他进程共享一些内存页面；
- 进程无法访问的内核空间地址范围（但只要执行内核调用即可访问）。

文本、数据和 BSS 部分

当程序首次运行时，内核会在内部创建一个进程并为其分配一个唯一的 PID。然后，它会为该进程设置虚拟页映射，换句话说，它会创建进程的虚拟地址空间。之后，它会根据需要分配物理页面，并通过在进程的页表中添加条目，将这些物理页面映射到虚拟地址空间。

内核通过分配虚拟页并将数据加载到其中，将可执行文件（文本、数据和 BSS 段）读入内存。这使得程序的代码和全局数据在进程的虚拟地址空间中“可见”。可执行文件中的机器码实际上是可重定位的，这意味着其地址被指定为相对偏移量，而不是绝对内存地址。这些相对地址由操作系统修复，这意味着在运行程序之前，它们会被转换回真实（虚拟）地址。（有关可执行文件格式的更多信息，请参见第 3.3.5.1 节。）

调用堆栈

每个正在运行的线程都需要一个调用栈（参见 3.3.5.2 节）。进程首次运行时，内核会为其创建一个默认线程。内核会为该线程的调用栈分配物理内存页，并将其映射到进程的虚拟地址空间，以便线程可以“看到”该栈。栈指针 (SP) 和基指针 (BP) 的值会被初始化为指向空栈的底部。最后，线程开始执行

kernel are protected from inadvertent or deliberate corruption by a user process because they are mapped to a special range of addresses known as *kernel space* which can only be accessed by code running in kernel mode.

A process's virtual page table effectively defines its *memory map*. The memory map typically contains:

- the text, data and BSS sections as read in from the program's executable file;
- a view of any shared libraries (DLLs, PRXs) used by the program;
- a *call stack* for each thread;
- a region of memory called the *heap* for dynamic memory allocation;
- possibly some pages of memory that are *shared* with other processes; and
- a range of *kernel space* addresses which are inaccessible to the process (but become accessible whenever a kernel call executes).

Text, Data and BSS Sections

When a program is first run, the kernel creates a process internally and assigns it a unique PID. It then sets up a virtual page map for the process—in other words, it creates the virtual address space of the process. It then allocates physical pages as necessary and maps them into the virtual address space by adding entries to the process's page table.

The kernel reads the executable file (text, data and BSS sections) into memory by allocating virtual pages and loading the data into them. This allows the program's code and global data to be "visible" within the process's virtual address space. The machine code in an executable file is actually *relocatable*, meaning that its addresses are specified as relative offsets rather than absolute memory addresses. These relative addresses are *fixed up* by the operating system, meaning they are converted back into real (virtual) addresses, prior to running the program. (For more on the format of an executable file, see Section 3.3.5.1.)

Call Stack

Every running thread needs a call stack (see Section 3.3.5.2). When a process is first run, the kernel creates a single default thread for it. The kernel allocates physical memory pages for this thread's call stack, and maps them into the process' virtual address space so the stack can be "seen" by the thread. The values of the stack pointer (SP) and base pointer (BP) are initialized to point to the bottom of the empty stack. Finally, the thread starts executing at the

程序的入口点。（在 C/C++ 中，通常是 main()，在 Windows 下则是 WinMain()。）

堆

进程可以通过 C 语言中的 malloc() 和 free() 函数，或 C++ 中的全局 new 和 delete 函数动态分配内存。这些请求来自一个称为堆的内存区域。内核会根据需要分配物理内存页面，以满足动态分配请求；这些页面会在进程分配内存时映射到进程的虚拟地址空间中，而内容已完全释放的页面则会被取消映射并返回给系统。

共享库

所有重要的程序都依赖于外部库。库可以静态链接到程序中，这意味着库代码的副本会被放入可执行文件本身。大多数操作系统也支持共享库的概念。在这种情况下，程序仅包含对库 API 函数的引用，而不包含库机器码的副本。在 Windows 下，共享库被称为动态链接库 (DLL)。在 PlayStation 4 上，操作系统支持一种称为 PRX 的动态链接库。（有趣的是，PRX 这个名字来自 PlayStation 3，它代表 PPU R elocatable E executable，指的是 PS3 中的主处理器，称为 PPU。）

共享库的工作原理通常如下：当进程首次需要共享库时，操作系统会将该库加载到物理内存中，并将其视图映射到进程的虚拟地址空间中。共享库提供的函数和全局变量的地址会被添加到程序的机器码中，从而允许程序像调用静态链接到可执行文件一样调用它们。

共享库的优势只有在使用相同共享库的第二个进程运行时才会显现。此时，无需加载库代码和全局变量的副本，而是将已加载的物理页面直接映射到新进程的虚拟地址空间中。这不仅节省了内存，还加快了除第一个使用共享库的进程之外所有进程的运行速度。

共享库还有其他好处。例如，如果要更新共享库（例如修复一些 bug），理论上所有使用该共享库的程序都会立即受益（无需重新链接并重新分发给用户）。但实际上，更新共享库可能会

entry point of the program. (In C/C++ this is typically `main()`, or `WinMain()` under Windows.)

Heap

Processes can allocate memory *dynamically* via `malloc()` and `free()` in C, or global `new` and `delete` in C++. These requests come from a region of memory called the *heap*. Physical pages of memory are allocated on demand by the kernel in order to fulfill dynamic allocation requests; these pages are mapped into the virtual address space of the process as memory is allocated by it, and pages whose contents have been completely freed are unmapped and returned to the system.

Shared Libraries

All non-trivial programs depend on external libraries. A library can be *statically* linked into a program, meaning that a *copy* of the libary's code is placed into the executable file itself. Most operating systems also support the concept of *shared libraries*. In this case, the program contains only *references* to the library's API functions, not a copy of the library's machine code. Shared libraries are called *dynamic link libraries* (DLL) under Windows. On the PlayStation 4, the OS supports a kind of dynamically linked library called a PRX. (Interestingly, the name PRX comes from the PlayStation 3, where it stood for **P**PU **R**elocatable **E**xecutable, in reference to the main processor in the PS3 which was called the PPU.)

Shared libraries generally work as follows: The first time a shared library is needed by a process, the OS loads that library into physical memory, and maps a view of it into the process's virtual address space. The addresses of the functions and global variables provided by the shared library are patched into the program's machine code, allowing it to call them as if they had been statically linked into the executable.

The benefit of shared libraries only becomes evident when a second process is run that uses the same shared library. Rather than loading a *copy* of the library's code and global variables, the already-loaded physical pages are simply mapped into the virtual address space of the new process. This saves memory and speeds up the process of running all but the first process that uses a given shared library.

Shared libraries have other benefits, too. For example, a shared library can be updated, say to fix some bugs, and in theory all programs that use that shared library will immediately benefit (without having to be relinked and redistributed to users). That said, in practice updating shared libraries can

无意中会导致使用它们的程序出现兼容性问题。这会导致系统内各个共享库的不同版本激增——这种情况被 Windows 开发人员亲切地称为“DLL 地狱”。为了解决这些问题，Windows 迁移到了一个清单系统，以帮助保证共享库与使用它们的程序之间的兼容性。

内核页面

在大多数操作系统中，进程的地址空间实际上被划分为两大块连续的地址——用户空间和内核空间。例如，在 32 位 Windows 中，用户空间对应的地址范围是从

0x0 到 0x7FFFFFFF（地址空间的低 2 GiB），而内核空间对应的地址范围是 0x80000000 到 0xFFFFFFFF（地址空间的高 2 GiB）。在 64 位 Windows 上，用户空间对应的地址范围是从 0x0 到 0x7FF'FFFFFFFF 这 8 TiB，而从 0xFFFF0800'00000000 到 0xFFFFFFFF'FFFFFFFF 这 248 TiB 的庞大范围则保留给内核使用（尽管实际上并非全部都用到了）。

用户空间通过每个进程独有的虚拟页表进行映射。然而，内核空间使用一个独立的虚拟页表，该表由所有进程共享。这样做的目的是确保系统中的所有进程对内核的内部数据拥有一致的“视图”。

通常情况下，用户进程无法访问内核的页面——如果用户进程尝试访问，就会发生页面错误，程序崩溃。但是，当用户进程进行系统调用时，内核会执行上下文切换（参见第 4.4.6.5 节）。这会将 CPU 置于特权模式，允许内核访问内核空间地址范围（以及当前进程的虚拟页面）。内核以特权模式运行其代码，根据需要更新其内部数据结构，最终将 CPU 重新置于用户模式并将控制权返回给用户程序。有关 Windows 下用户空间和内核空间内存映射工作原理的更多详细信息，请在 https://docs.microsoft.com 上搜索“虚拟地址空间”。

值得注意的是，最近发现的“Meltdown”和“Spectre”漏洞利用了CPU的乱序执行逻辑和推测执行逻辑，诱使CPU访问通常受到保护的内存页中的数据，而这些内存页通常不会受到用户模式进程的攻击。有关这些漏洞的更多信息以及操作系统如何防范这些漏洞，请参阅https://meltdownattack.com/。

inadvertently cause compatibility problems amongst the programs that use them. This leads to a proliferation of different versions of each shared library within the system—a situation affectionately known as “DLL hell” amongst Windows developers. To work around these problems, Windows moved to a system of manifests that help to guarantee compatibility between shared libraries and the programs that use them.

Kernel Pages

On most operating systems, the address space of a process is actually divided into two large contiguous blocks—user space and kernel space. For example, on 32-bit Windows, user space corresponds to the address range from address 0x0 through 0x7FFFFFFF (the lower 2 GiB of the address space), while kernel space corresponds to addresses between 0x80000000 and 0xFFFFFFFF (the upper 2 GiB of the space). On 64-bit Windows, user space corresponds to the 8 TiB range of addresses from 0x0 through 0x7FF’FFFFFFFF, and the gigantic 248 TiB range from 0xFFFF0800’00000000 through 0xFFFFFFFF’FFFFFFFF is reserved for use by the kernel (although not all of it is actually used).

User space is mapped through a virtual page table that is unique to each process. However, kernel space uses a separate virtual page table that is shared between all processes. This is done so that all processes in the system have a consistent “view” of the kernel’s internal data.

Normally, user processes are prevented from accessing the kernel’s pages—if they try to do so, a page fault will occur and the program will crash. However, when a user process makes a system call, a *context switch* (see Section 4.4.6.5) is performed into the kernel. This puts the CPU into privileged mode, allowing the kernel to access the kernel space address ranges (as well as the virtual pages of the current process). The kernel runs its code in privileged mode, updates its internal data structures as necessary, and finally puts the CPU back into user mode and returns control to the user program. For more details on how user and kernel space memory mapping works under Windows, search for “virtual address spaces” on https://docs.microsoft.com.

It’s interesting (and a bit frightening!) to note that the recently-discovered “Meltdown” and “Spectre” exploits make use of a CPU’s *out-of-order* and *speculative* execution logic (respectively) to trick it into accessing data located in memory pages that are normally protected from a user-mode process. For more on these exploits and how operating systems are protecting themselves against them, see https://meltdownattack.com/.

图 4.23. 进程的内存映射在 32 位 Windows 下的样子。

进程内存映射示例

图 4.23 描绘了进程在 32 位 Windows 下的内存映射。进程的所有虚拟页面都被映射到用户空间——地址空间的低 2 GiB。可执行文件的文本段、数据段和 BSS 段被映射到低内存地址，然后是较高范围的堆，最后是任何共享内存页面。调用堆栈被映射到用户地址空间的高端。最后，操作系统的内核页面被映射到地址空间的高 2 GiB。

内存映射中每个区域的实际地址是不可预测的。部分原因是每个程序的段大小不同，因此会映射到不同的地址范围。此外，由于一种称为地址空间布局随机化 (ASLR) 的安全措施，地址的数值实际上会在同一个可执行程序运行之间发生变化。

Figure 4.23. A process's memory map as it might look under 32-bit Windows.

Example Process Memory Map

Figure 4.23 depicts the memory map of a process as it might look under 32-bit Windows. All of the process's virtual pages are mapped into user space—the lower 2 GiB of the address space. The executable files' text, data and BSS segments are mapped at a low memory address, followed by the heap in a higher range, followed by any shared memory pages. The call stack is mapped at the high end of the user address space. Finally, the operating system's kernel pages are mapped into the upper 2 GiB of the address space.

The actual addresses for each of the regions in the memory map aren't predictable. This is partly because each program's segments are of different sizes and hence will map to different address ranges. Also, the numeric values of the addresses actually *change* between runs of the same executable program, thanks to a security measure known as *address space layout randomization* (ASLR).

4.4.6 线程

线程封装了单个机器语言指令流的运行实例。进程中的每个线程都包含以下部分：

- 线程 ID (TID) 在其进程内是唯一的，但在整个操作系统中可能是或可能不是唯一的；
- 线程的调用堆栈——包含所有当前正在执行的函数的堆栈帧的连续内存块；
- 所有专用和通用寄存器 4 的值，包括指向线程指令流中当前指令的指令指针 (IP)、定义当前函数堆栈框架的基指针 (BP) 和堆栈指针 (SP)；
- 与每个线程关联的通用内存块，称为线程本地存储 (TLS)。

默认情况下，一个进程包含一个主线程，因此执行单个指令流。该线程从程序的入口点（通常是 main() 函数）开始执行。然而，所有现代操作系统都能够在单个进程的上下文中执行多个并发指令流。

您可以将线程视为操作系统中的基本执行单元。线程提供执行指令流所需的最小资源——调用堆栈和一组寄存器。进程仅提供一个或多个线程执行的环境。如图 4.24 所示。

4.4.6.1 线程库

所有支持多线程的操作系统都提供了一组用于创建和操作线程的系统调用。此外，还有一些可移植的线程库可用，其中最著名的是 IEEE POSIX 1003.1c 标准线程库 (pthread) 以及 C11 和 C++11 标准线程库。索尼 PlayStation 4 SDK 提供了一组以 sce 为前缀的线程函数，它们几乎直接映射到 POSIX 线程 API。

各种线程 API 在细节上有所不同，但它们都支持以下基本操作：

1. 创建。生成新线程的函数或类构造函数。

4 从技术上讲，线程的执行上下文仅包含在用户模式下可见的寄存器的值；它不包括某些特权模式寄存器的值。

4.4.6 Threads

A *thread* encapsulates a running instance of a single stream of machine language instructions. Each thread within a process is comprised of:

- a *thread id (TID)* which is unique within its process, but may or may not be unique across the entire operating system;
- the thread's *call stack*—a contiguous block of memory containing the stack frames of all currently-executing functions;
- the values of all special- and general-purpose *registers*[4] including the *instruction pointer* (IP), which points at the current instruction in the thread's instruction stream, the *base pointer* (BP) and *stack pointer* (SP) which define the current function's stack frame;
- a block of general-purpose memory associated with each thread, known as *thread local storage* (TLS).

By default, a process contains a single *main thread* and hence executes a single instruction stream. This thread begins executing at the entry point of the program—typically the `main()` function. However, all modern operating systems are capable of executing more than one *concurrent* instruction stream within the context of a single process.

You can think of a thread as the *fundamental unit of execution* within the operating system. A thread provides the minimum resources required to execute an instruction stream—a call stack and a set of registers. The process merely provides the *environment* in which one or more threads execute. This is illustrated in Figure 4.24.

4.4.6.1 Thread Libraries

All operating systems that support multithreading provide a collection of system calls for creating and manipulating threads. A few portable thread libraries are also available, the best known of which are the IEEE POSIX 1003.1c standard thread library (pthread) and the C11 and C++11 standard thread libraries. The Sony PlayStation 4 SDK provides a set of thread functions prefixed with `sce` that map pretty much directly to the POSIX thread API.

The various thread APIs differ in their details, but all of them support the following basic operations:

1. *Create*. A function or class constructor that spawns a new thread.

[4] Technically a thread's execution context only encompasses the values of registers that are visible in *user mode*; it excludes the values of certain privileged-mode registers.

图 4.24 进程封装了运行一个或多个线程所需的资源。每个线程都封装了一个由 CPU 寄存器内容和调用堆栈组成的执行上下文。

2. Terminate 。终止调用线程的函数。

3. 请求退出。允许一个线程请求另一个线程退出的函数。

4. 睡眠。使当前线程睡眠指定时间长度的函数。

5. Yield 。该函数用于放弃线程剩余的时间片，以便其他线程有机会运行。

6. Join 。该函数使调用线程进入睡眠状态，直到另一个线程或线程组终止。

4.4.6.2 线程创建和终止

当可执行文件运行时，操作系统自动创建的用于封装该文件的进程包含一个线程，该线程的执行从程序的入口点开始——在 C/C++ 中，入口点是特殊函数 main() 。如果需要，这个“主线程”可以通过调用操作系统特定的函数（例如 pthread_create() （POSIX 线程）、CreateThread() （Windows 线程））或实例化一个

Figure 4.24. A process encapsulates the resources required to run one or more threads. Each thread encapsulates an *execution context* comprised of the contents of the CPU's registers and a call stack.

2. *Terminate*. A function that terminates the calling thread.

3. *Request to exit*. A function that allows one thread to request another thread to exit.

4. *Sleep*. A function that puts the current thread to sleep for a specified length of time.

5. *Yield*. A function that yields the remainder of the thread's time slice so other threads can get a chance to run.

6. *Join*. A function that puts the calling thread to sleep until another thread or group of threads has terminated.

4.4.6.2 Thread Creation and Termination

When an executable file is run, the process created by the OS to encapsulate it automatically contains a single thread, and execution of this thread begins at the *entry point* of the program—in C/C++ this is the special function `main()`. This "main thread" can spawn new threads if desired, by making calls to an operating system specific function such as `pthread_create()` (POSIX threads), `CreateThread()` (Windows), or by instantiating an instance of a

线程类，例如 std::thread (C++11)。新线程从调用者提供的入口点函数开始执行。
线程一旦创建，就会一直存在，直到终止。线程的执行可以通过多种方式终止：

- 它可以通过从其入口点函数返回来“自然”结束。（在主线程的特殊情况下，从 main() 返回不仅会结束线程，还会结束整个进程。）

- 它可以调用诸如 pthread_exit() 之类的函数来在从其入口点函数返回之前明确终止其执行。

- 它可能被其他线程从外部终止。在这种情况下，外部线程会发出取消相关线程的请求，但该线程可能不会立即响应该请求，或者可能完全忽略该请求。线程的可取消性在创建时确定。

- 由于进程已结束，进程可能会被强制终止。（进程终止的条件包括：主线程从 main() 入口函数返回；任何线程调用 exit() 显式终止进程；或者外部参与者终止进程。）

4.4.6.3 加入线程

一个线程通常会创建一个或多个子线程，执行一些有用的工作，然后等待子线程完成工作后再继续执行。例如，假设主线程要执行 1000 次计算，并且我们进一步假设该程序运行在四核机器上。最有效的方法是将工作分成四个大小相等的块，并创建四个线程并行处理。计算完成后，假设主线程要对结果执行校验和。最终的代码可能如下所示：

```
ComputationResult g_aResult[1000];

void Compute(void* arg)
{
    uintptr_t startIndex = (uintptr_t)arg;
    uintptr_t endIndex = startIndex + 250;
    for (uintptr_t i = startIndex; i < endIndex; ++i)
    {
        g_aResult[i] = ComputeOneResult(...);
```


thread class such as `std::thread` (C++11). The new thread begins execution at an entry point function whose address is provided by the caller.

Once created, a thread will continue to exist until it terminates. The execution of a thread can terminate in a number of ways:

- It can end “naturally” by *returning* from its entry point function. (In the special case of the main thread, returning from `main()` not only ends the thread, but also ends the entire process.)

- It can call a function such as `pthread_exit()` to *explicitly* terminate its execution *before* having returned from its entry point function.

- It can be *killed* externally by another thread. In this case, the external thread makes a *request* to cancel the thread in question, but the thread may not respond immediately to the request, or it may ignore the request entirely. The cancelability of a thread is determined when that thread is created.

- It can be forcibly killed because its process has ended. (A process terminates when the main thread returns from the `main()` entry point function, when any thread calls `exit()` to kill the process explicitly, or when an external actor kills the process.)

4.4.6.3 Joining Threads

It’s common for one thread to spawn one or more child threads, do some useful work of its own, and then wait for the child threads to be done with their work before continuing. For example, let’s say the main thread wants to perform 1000 computations, and let’s assume further that this program is running on a quad-core machine. The most efficient approach would be to divide the work into four equally-sized chunks, and spawn four threads to do the processing in parallel. Once the computations are complete, let’s assume that the main thread wants to perform a checksum on the results. The resulting code might look something like this:

```
ComputationResult g_aResult[1000];

void Compute(void* arg)
{
    uintptr_t startIndex = (uintptr_t)arg;
    uintptr_t endIndex = startIndex + 250;
    for (uintptr_t i = startIndex; i < endIndex; ++i)
    {
        g_aResult[i] = ComputeOneResult(...);
```

```
    }
}

void main()
{
    pthread_t  tid[4];

    for (int i = 0; i < 4; ++i)
    {
        const uintptr_t startIndex = i * 250;
        pthread_create(&tid[i], nullptr,
                       Compute, (void*)startIndex);
    }

    // perhaps do some other useful work...

    // wait for computations to be done
    for (int i = 0; i < 4; ++i)
    {
        pthread_join(&tid[i], nullptr);
    }

    // all threads are done, so we can do our checksum
    unsigned checksum = Sha1(g_aResult,
        1000*sizeof(ComputationResult));

    // ...
}
```

4.4.6.4 轮询、阻塞和让步

通常，线程会一直运行直到终止。但有时，正在运行的线程需要等待某些未来事件的发生。例如，线程可能需要等待某个耗时操作完成，或者等待某些资源可用。在这种情况下，我们有三种选择：

1. 线程可以轮询，
2. 它可以阻止，或者
3. 轮询时可以产生。

轮询

轮询涉及一个线程处于紧密循环中，等待条件满足


```
    }
}

void main()
{
    pthread_t  tid[4];

    for (int i = 0; i < 4; ++i)
    {
        const uintptr_t startIndex = i * 250;
        pthread_create(&tid[i], nullptr,
                       Compute, (void*)startIndex);
    }

    // perhaps do some other useful work...

    // wait for computations to be done
    for (int i = 0; i < 4; ++i)
    {
        pthread_join(&tid[i], nullptr);
    }

    // all threads are done, so we can do our checksum
    unsigned checksum = Sha1(g_aResult,
        1000*sizeof(ComputationResult));

    // ...
}
```

4.4.6.4 Polling, Blocking and Yielding

Normallly a thread runs until it terminates. But sometimes a running thread needs to wait for some future event to occur. For example, a thread might need to wait for a time-consuming operation to complete, or for some resource to become available. In such a situation, we have three options:

1. The thread can *poll*,
2. it can *block*, or
3. it can *yield* while polling.

Polling

Polling involves a thread sitting in a tight loop, waiting for a condition to be-

梦想成真。这有点像旅途中坐在后座的孩子，不停地问："我们到了吗？我们到了吗？"举个例子：

```
// wait for condition to become true
while (!CheckCondition())
{
    // twiddle thumbs
}

// the condition is now true and we can continue...
```

显然，这种方法虽然简单，但可能会不必要地消耗 CPU 周期。这种方法有时被称为"自旋等待"或"忙等待"。

阻塞

如果我们预计线程需要等待相对较长的时间才能使某个条件成立，那么忙等待不是一个好的选择。理想情况下，我们应该让线程进入睡眠状态，这样就不会浪费 CPU 资源，并依靠内核在未来某个时间条件成立时将其唤醒。这称为阻塞线程。

线程通过执行一种称为阻塞函数的特殊操作系统调用来阻塞。如果阻塞函数被调用时条件已经为真，则该函数实际上不会阻塞——它会立即返回。但如果条件为假，内核会让线程进入睡眠状态，并将线程及其等待的条件添加到一个表中。之后，当条件变为真时，内核会使用这个内部表来识别并唤醒任何正在等待该条件的线程。

各种各样的操作系统函数都会阻塞。以下是一些示例：

• 打开文件。大多数打开文件的函数（例如 fopen()）都会阻塞调用线程，直到文件真正打开（这可能需要数百甚至数千个周期）。某些函数（例如 Linux 下的 open()）提供了非阻塞选项（O_NONBLOCK）来支持异步文件 I/O。

• 显式休眠。某些函数会显式地让调用线程休眠指定的时间长度。相关变体包括 usleep() (Linux)、Sleep() (Windows)、std::this_thread::sleep_until() (C++11 标准库) 和 pthread_sleep() (POSIX 线程)。

• 与另一个线程连接。诸如 pthread_join() 之类的函数会阻塞调用线程，直到被等待的线程终止。

come true. It's a bit like kids in the back seat on a road trip, repeatedly asking, "Are we there yet? Are we there yet?" Here's an example:

```
// wait for condition to become true
while (!CheckCondition())
{
    // twiddle thumbs
}

// the condition is now true and we can continue...
```

It should be obvious that this approach, while simple, has the potential of burning CPU cycles unnecessarily. This approach is sometimes called a *spin-wait* or a *busy-wait*.

Blocking

If we expect our thread to wait for a relatively long period of time for a condition to become true, busy-waiting is not a good option. Ideally we'd like to put our thread to sleep so that it doesn't waste CPU resources, and rely on the kernel to wake it back up when the condition becomes true at some future time. This is called *blocking* the thread.

A thread blocks by making a special kind of operating system call known as a *blocking function*. If the condition is already true at the moment a blocking function is called, the function won't actually block—it will simply return immediately. But if the condition is false, the kernel will put the thread to sleep, and add the thread and the condition on which it is waiting into a table. Later, when the condition becomes true, the kernel uses this internal table to identify and wake up any threads that are waiting on that condition.

There are all sorts of OS functions that block. Here are a few examples:

- *Opening a file*. Most functions that open a file such as `fopen()` will block the calling thread until the file has actually been opened (which may take hundreds or even thousands of cycles). Some functions, like `open()` under Linux, offer a non-blocking option (`O_NONBLOCK`) to support asynchronous file I/O.
- *Explicit sleeping*. Some functions explicitly put the calling thread to sleep for a specified length of time. Variants include `usleep()` (Linux), `Sleep()` (Windows) `std::this_thread::sleep_until()` (C++11 standard library) and `pthread_sleep()` (POSIX threads).
- *Joining with another thread*. A function such as `pthread_join()` blocks the calling thread until the thread being waited on has terminated.

• 等待互斥锁。类似 pthread_mutex_wait() 的函数会尝试通过操作系统对象（称为互斥锁）获取资源的独占锁（参见 4.6 节）。如果没有其他线程持有该锁，该函数会将锁授予调用线程并立即返回；否则，调用线程将进入休眠状态，直到获取到锁为止。

操作系统调用并非唯一可能阻塞的函数。任何最终调用阻塞操作系统函数的用户空间函数本身也被视为阻塞函数。最好为此类函数添加文档，以便使用它的程序员知道它有可能阻塞。

屈服

这种技术介于轮询和阻塞之间。线程循环轮询条件，但在每次迭代中，它会通过调用 pthread_yield() (POSIX)、Sleep(0) 或 SwitchToThread() (Windows) 或等效的系统调用来放弃剩余的时间片。

以下是一个例子：

```
// wait for condition to become true
while (!CheckCondition())
{
    // yield the remainder of my time slice
    pthread_yield(nullptr);
}

// the condition is now true and we can continue...
```

与纯粹的忙等待循环相比，这种方法往往会减少浪费的周期并降低功耗。

放弃 CPU 仍然需要内核调用，因此开销相当大。一些 CPU 提供了轻量级的“暂停”指令。（例如，在支持 SSE2 的 Intel x86 ISA 上，_mm_pause() 内部函数会发出这样的指令。）这类指令通过简单地等待 CPU 的指令流水线清空后再允许继续执行，从而降低了忙等待循环的功耗：

```
// wait for condition to become true
while (!CheckCondition())
{
    // Intel SSE2 only:
    // reduce power consumption by pausing for ~40 cycles
    _mm_pause();
```


- *Waiting for a mutex lock*. Functions like `pthread_mutex_wait()` attempt to obtain an exclusive lock on a resource via an operating system object known as a *mutex* (see Section 4.6). If no other thread holds the lock, the function grants the lock to the calling thread and returns immediately; otherwise, the calling thread is put to sleep until the lock can be obtained.

Operating system calls aren't the only functions that can block. Any user-space function that ultimately calls a blocking OS function is itself considered to be a blocking function. It's a good idea to document such a function, so that the programmers who use it will know that it has the potential to block.

Yielding

This technique falls part-way between polling and blocking. The thread polls the condition in a loop, but on each iteration it relinquishes the remainder of its time slice by calling `pthread_yield()` (POSIX), `Sleep(0)` or `SwitchToThread()` (Windows), or an equivalent system call.

Here's an example:

```
// wait for condition to become true
while (!CheckCondition())
{
    // yield the remainder of my time slice
    pthread_yield(nullptr);
}

// the condition is now true and we can continue...
```

This approach tends to result in fewer wasted cycles and better power consumption than a pure busy-wait loop.

Yielding the CPU still involves a kernel call, and is therefore quite expensive. Some CPUs provide a lightweight "pause" instruction. (For example, on an Intel x86 ISA with SSE2, the `_mm_pause()` intrinsic emits such an instruction.) This kind of instruction reduces the power consumption of a busy-wait loop by simply waiting for the CPU's instruction pipeline to empty out before allowing execution to continue:

```
// wait for condition to become true
while (!CheckCondition())
{
    // Intel SSE2 only:
    // reduce power consumption by pausing for ~40 cycles
    _mm_pause();
```

```
}

// the condition is now true and we can continue...
```

有关如何以及为何在忙等待循环中使用暂停指令的深入讨论，请参阅 https://software.intel.com/en-us/comment/1134767 和 http://software.intel.com/en-us/forums/topic/309231。

4.4.6.5 上下文切换

内核维护的每个线程都处于以下三种状态之一：5

- 正在运行。线程正在核心上主动运行。
- 可运行。线程能够运行，但正在等待接收核心上的时间片。
- 阻塞。线程处于睡眠状态，等待某些条件变为真。

每当内核导致线程从其中一个状态转换到另一个状态时，就会发生上下文切换。
上下文切换总是在 CPU 的特权模式下进行——响应驱动抢占式多任务处理的硬件中断（即在“运行”和“就绪”状态之间切换）；响应正在运行的线程发出的显式阻塞内核调用（即从“运行”或“就绪”状态切换到“阻塞”）；或者响应等待条件变为真，从而“唤醒”正在休眠的线程（即从“阻塞”状态切换到“就绪”状态）。内核的线程状态机如图 4.25 所示。

当线程处于“运行”状态时，它正在主动使用 CPU 核心。核心的寄存器包含与该线程执行相关的信息，例如其指令指针 (IP)、堆栈和基址指针（SP 和 BP），以及各种通用寄存器 (GPR) 的内容。该线程还维护一个调用堆栈，该堆栈存储当前正在运行的函数以及最终调用该函数的整个函数堆栈的局部变量和返回地址。这些信息统称为线程的执行上下文。

每当线程从“运行”状态转换为“可运行”或“阻塞”状态时，CPU 寄存器的内容都会保存到内核为该线程保留的内存块中。之后，当“可运行”线程转换回“运行”状态时，内核会将该线程保存的寄存器内容重新填充到 CPU 寄存器中。

5 一些操作系统利用了附加状态，但这些状态是实现细节，我们可以安全地忽略它们。


```
}

// the condition is now true and we can continue...
```

See https://software.intel.com/en-us/comment/1134767 and http://software.intel.com/en-us/forums/topic/309231 for an in-depth discussion of how and why to use a pause instruction in a busy-wait loop.

4.4.6.5 Context Switching

Every thread maintained by the kernel exists in one of three states:[5]

- *Running*. The thread is actively running on a core.
- *Runnable*. The thread is able to run, but it is waiting to receive a time slice on a core.
- *Blocked*. The thread is asleep, waiting for some condition to become true.

A *context switch* occurs whenever the kernel causes a thread to transition from one of these states to another.

A context switch always happens in privileged mode on the CPU—in response to the hardware interrupt that drives preemptive multitasking (i.e., transitions between Running and Runnable), in response to an explicit blocking kernel call made by a running thread (i.e., a transition from Running or Runnable to Blocked), or in response to a waited-on condition becoming true, thus “waking” a sleeping thread (i.e., transitioning it from Blocked to Runnable). The kernel’s thread state machine is illustrated in Figure 4.25.

When a thread is in the Running state, it is actively making use of a CPU core. The core’s registers contain information pertinent to the execution of that thread, such as its instruction pointer (IP), stack and base pointers (SP and BP), and the contents of various general-purpose registers (GPRs). The thread also maintains a call stack, which stores local variables and return addresses for the currently-running function and the entire stack of functions that ultimately called it. Together, this information is known as the thread’s *execution context*.

Whenever a thread transitions away from the Running state to either Runnable or Blocked, the contents of the CPU’s registers are saved to a memory block that has been reserved for the thread by the kernel. Later, when a Runnable thread transitions back to the Running state, the kernel repopulates the CPU’s registers with that thread’s saved register contents.

[5]Some operating systems make use of additional states, but such states are implementation details that we can safely ignore for our purposes.

图 4.25。每个线程都可以处于以下三种状态之一：正在运行、可运行或阻塞。

这里需要注意的是，在上下文切换期间，线程的调用堆栈无需显式保存或恢复。这是因为每个线程的调用堆栈已经驻留在其进程虚拟内存映射中的不同区域中。保存和恢复 CPU 寄存器内容的操作包括保存和恢复堆栈和基指针（SP 和 BP），从而有效地“免费”地保存和恢复线程的调用堆栈。

在上下文切换期间，如果传入线程与传出线程驻留在不同的进程中，则内核还需要保存传出进程的虚拟内存映射状态，并设置传入进程的虚拟内存映射。您可能还记得 3.5.2 节中提到，虚拟内存映射由虚拟页表定义。因此，保存和恢复虚拟内存映射的操作涉及保存和恢复指向该页表的指针，该指针通常维护在一个特殊的特权 CPU 寄存器中。每当发生进程间上下文切换时，还必须刷新转换后备缓冲区 (TLB)（参见 3.5.2.4 节）。这些额外的步骤使得进程间上下文切换比单个进程内线程间的上下文切换更昂贵。

4.4.6.6 线程优先级和亲和性

在大多数情况下，内核负责调度线程，使其在机器的可用核心上运行。然而，程序员有两种方法

Figure 4.25. Every thread can be in one of three states: Running, Runnable or Blocked.

We should note here that a thread's call stack need not be saved or restored explicitly during a context switch. This is because each thread's call stack already resides in a distinct region within its process' virtual memory map. The act of saving and restoring the contents of the CPU's registers includes saving and restoring the stack and base pointers (SP and BP), and thereby effectively saves and restores the thread's call stack "for free."

During a context switch, if the incoming thread resides in a different process from that of the outgoing thread, the kernel also needs to save off the state of the outgoing process' virtual memory map, and set up the virtual memory map of the incoming process. You'll recall from Section 3.5.2 that a virtual memory map is defined by a virtual page table. The act of saving and restoring a virtual memory map therefore involves saving and restoring a pointer to this page table, which is usually maintained in a special privileged CPU register. The translation lookaside buffer (TLB) must also be flushed whenever an inter-process context switch occurs (see Section 3.5.2.4). These additional steps make context switching between processes more expensive than context switching between threads within a single process.

4.4.6.6 Thread Priorities and Affinity

For the most part, the kernel handles the job of scheduling threads to run on the available cores in the machine. However, programmers do have two ways

影响线程的调度方式：优先级和亲和力。
线程的优先级决定了它相对于系统中其他 Runnable 线程的调度方式。高优先级线程通常优先于低优先级线程。不同的操作系统提供不同数量的优先级。例如，Windows 线程可以属于六个优先级类别之一，每个类别中有七个不同的优先级。这两个值组合起来，总共产生 32 个不同的“基本优先级”，用于调度线程。

最简单的线程调度规则是：只要存在至少一个高优先级的 Runnable 线程，就不会调度任何低优先级的线程运行。这种方法背后的原理是，系统中的大多数线程都会以某个默认优先级创建，从而公平地共享处理资源。但偶尔也会有一个高优先级的线程变为 Runnable 状态。当它变为 Runnable 状态时，它会尽可能立即运行，并希望在相对较短的时间内退出，从而将控制权交还给所有低优先级的线程。

这种简单的基于优先级的调度算法可能会导致少数高优先级线程持续运行，从而阻止任何低优先级线程运行。这种情况被称为“饥饿”。一些操作系统试图通过在简单调度规则中引入例外来缓解“饥饿”带来的负面影响，这些例外旨在为处于饥饿状态的低优先级线程提供至少一些 CPU 时间。

程序员控制线程调度的另一种方式是通过线程的亲和性。此设置要求内核将线程锁定到特定核心，或者在调度线程时至少优先选择一个或多个核心。

4.4.6.7 线程本地存储

我们说过，进程内的所有线程共享进程的资源，包括其虚拟内存空间。这条规则有一个例外——每个线程都被赋予一个私有内存块，称为线程本地存储 (TLS)。这允许线程跟踪不应与其他进程共享的数据。例如，每个线程可能维护一个私有内存分配器。我们可以将 TLS 内存块视为线程执行上下文的一部分。

实际上，TLS 内存块通常对进程内的所有线程都是可见的。它们通常不像操作系统虚拟内存页那样受到保护。相反，操作系统会为每个线程分配一个自己的 TLS 内存块，所有 TLS 内存块都映射到进程虚拟地址空间的不同数字地址，并提供一个系统调用，允许任何一个线程获取

to affect how threads are scheduled: *priority* and *affinity*.

A thread's priority controls how it is scheduled relative to other Runnable threads in the system. Higher-priority threads generally take precedence over lower-priority threads. Different operating systems offer different numbers of priority levels. For example, Windows threads can belong to one of six priority classes, and there are seven distinct priority levels within each class. These two values are combined to produce a total of 32 distinct "base priorities" which are used when scheduling threads.

The simplest thread scheduling rule is this: As long as at least one higher-priority Runnable thread exists, no lower-priority threads will be scheduled to run. The idea behind this approach is that most threads in the system will be created at some default priority level, and hence will share the processing resources fairly. But once in a while, a higher-priority thread can become Runnable. When it does, it runs as close to immediately as possible, hopefully exits after a relatively short period of time, and thereby returns control to all of the lower-priority threads.

Such a simple priority-based scheduling algorithm can lead to a situation in which a small number of high-priority threads run continually, thereby preventing any lower-priority threads from running. This is known as *starvation*. Some operating systems attempt to mitigate the ill effects of starvation by introducing exceptions to the simple scheduling rule that aim to give at least some CPU time to starving lower-priority threads.

Another way in which programmers can control thread scheduling is via a thread's *affinity*. This setting requests that the kernel either lock a thread to a particular core, or that it should at least *prefer* one or more cores over the others when scheduling the thread.

4.4.6.7 Thread Local Storage

We said that all threads within a process share the process's resources, including its virtual memory space. There is one exception to this rule—each thread is given a private memory block known as *thread local storage* (TLS). This allows threads to keep track of data that shouldn't be shared with other processes. For example, each thread might maintain a private memory allocator. We can think of the TLS memory block as being a part of the thread's execution context.

In practice, TLS memory blocks are usually visible to all threads within a process. They're typically not protected, the way operating system virtual memory pages are. Instead, the OS grants each thread its own TLS block, all mapped into the process's virtual address space at different numerical addresses, and a system call is provided that allows any one thread to obtain

图 4.26。Visual Studio 中的线程窗口是调试多线程程序的主要界面。

其私有 TLS 块的地址。

4.4.6.8 线程调试

如今，所有优秀的调试器都提供了用于调试多线程应用程序的工具。在 Microsoft Visual Studio 中，“线程”窗口是实现此目的的核心工具。每当您中断进入调试器时，此窗口都会列出应用程序中当前存在的所有线程。双击某个线程会使其执行上下文在调试器中处于活动状态。一旦线程的上下文被激活，您就可以通过“调用堆栈”窗口上下移动其调用堆栈，并通过“监视”窗口查看每个函数范围内的局部变量。即使线程处于“可运行”或“阻塞”状态，此功能也有效。Visual Studio 线程窗口如图 4.26 所示。

4.4.7 纤维

在抢占式多任务处理中，线程调度由内核自动处理。这通常很方便，但有时程序员希望能够控制程序中工作负载的调度。例如，在为游戏引擎实现作业系统（将在 8.6.4 节中讨论）时，我们可能希望允许作业明确地将 CPU 让给其他作业，而不必担心在作业运行时被抢占。换句话说，有时我们希望使用合作式多任务处理，而不是抢占式多任务处理。

Figure 4.26. The Threads window in Visual Studio is the primary interface for debugging multithreaded programs.

the address of its private TLS block.

4.4.6.8 Thread Debugging

All good debuggers nowadays provide tools for debugging multithreaded applications. In Microsoft Visual Studio, the *Threads Window* is the central tool for this purpose. Whenever you break into the debugger, this window lists all the threads currently in existence within the application. Double-clicking on a thread makes its execution context active within the debugger. Once a thread's context has been activated, you can walk up and down its call stack via the Call Stack window, and view local variables within each function's scope via the Watch windows. This works even if the thread is in its Runnable or Blocked state. The Visual Studio Threads window is shown in Figure 4.26.

4.4.7 Fibers

In preemptive multitasking, thread scheduling is handled automatically by the kernel. This is often convenient, but sometimes programmers find it desirable to have control over the scheduling of workloads in their programs. For example, when implementing a *job system* for a game engine (discussed in Section 8.6.4), we might want to allow jobs to explicitly yield the CPU to other jobs, without worrying about the possibility of preemption "pulling the rug out" from under our jobs as they run. In other words, sometimes we want to use *cooperative* rather than preemptive multitasking.

一些操作系统提供了这样一种协作式多任务机制：它们被称为纤程 (fiber)。纤程与线程非常相似，因为它代表了机器语言指令流的运行实例。纤程与线程一样，具有调用堆栈和寄存器状态（执行上下文）。然而，最大的区别在于，纤程从不由内核直接调度。相反，纤程在线程上下文中运行，并由彼此协作调度。

在本节中，我们将专门讨论 Windows 纤程。其他一些操作系统，例如索尼的 PlayStation 4 SDK，也提供了非常类似的纤程 API。

4.4.7.1 纤维的产生和破坏

如何将基于线程的进程转换为基于纤程的进程？每个进程在首次运行时都以单个线程开始；因此默认情况下进程是基于线程的。当线程调用函数 ConvertThreadTo Fiber() 时，会在调用线程的上下文中创建一个新的纤程。这会“引导”进程，以便它可以创建和调度更多纤程。其他纤程是通过调用 CreateFiber() 并向其传递将作为其入口点的函数地址来创建的。任何正在运行的纤程都可以通过调用 SwitchTo Fiber() 来协作调度不同的纤程在其线程内运行。当不再需要纤程时，可以通过调用

`DeleteFiber()`.

4.4.7.2 光纤状态

纤程可以处于两种状态之一：活动 (Active) 或非活动 (Inactive)。当纤程处于活动状态时，它会被分配给一个线程，并代表该线程执行。当纤程处于非活动状态时，它会处于空闲状态，不消耗任何线程的资源，只是等待被激活。Windows 将活动纤程称为给定线程的“选定”纤程。

活动 Fiber 可以通过调用 SwitchToFiber() 来停用自身并激活另一条 Fiber。这是 Fiber 在活动和非活动状态之间切换的唯一方法。

Active 纤程是否正在 CPU 核心上执行取决于其所在线程的状态。当 Active 纤程的线程处于 Running 状态时，该纤程的机器语言指令正在核心上执行。当 Active 纤程的线程处于 Runnable 或 Blocked 状态时，其指令当然无法执行，因为整个线程处于等待状态，要么等待在核心上被调度，要么等待某个条件成立。

Some operating systems provide just such a cooperative multitasking mechanism: They are known as *fibers*. A fiber is a lot like a thread, in that it represents a running instance of a stream of machine language instructions. A fiber has a call stack and register state (an execution context), just like a thread. However, the big difference is that a fiber is never scheduled directly by the kernel. Instead, fibers run within the context of a thread, and are scheduled cooperatively, by each other.

In this section, we'll talk about Windows fibers specifically. Some other operating systems, such as Sony's PlayStation 4 SDK, provide very similar fiber APIs.

4.4.7.1 Fiber Creation and Destruction

How do we convert a thread-based process into a fiber-based one? Every process starts with a single thread when it first runs; hence processes are thread-based by default. When a thread calls the function `ConvertThreadTo Fiber()`, a new fiber is created within the context of the calling thread. This "bootstraps" the process so that it can create and schedule more fibers. Other fibers are created by calling `CreateFiber()` and passing it the address of a function that will serve as its entry point. Any running fiber can cooperatively schedule a different fiber to run within its thread by calling `SwitchTo Fiber()`. When a fiber is no longer needed, it can be destroyed by calling `DeleteFiber()`.

4.4.7.2 Fiber States

A fiber can be in one of two states: Active or Inactive. When a fiber is in its Active state, it is assigned to a thread, and executes on its behalf. When a fiber is in its Inactive state, it is sitting on the sidelines, not consuming the resources of any thread, just waiting to be activated. Windows calls an Active fiber the "selected" fiber for a given thread.

An Active fiber can deactivate itself and make another fiber active by calling `SwitchToFiber()`. This is the only way that fibers can switch between the Active and Inactive states.

Whether or not an Active fiber is actively executing on a CPU core is determined by the state of its enclosing thread. When an Active fiber's thread is in the Running state, that fiber's machine language instructions are being executed on a core. When an Active fiber's thread is in the Runnable or Blocked state, its instructions of course cannot execute, because the entire thread is sitting on the sidelines, either waiting to be scheduled on a core or waiting for a condition to become true.

重要的是要理解，纤程本身并不像线程那样具有阻塞状态。换句话说，无法让纤程在等待某个条件时进入睡眠状态。只有它的线程才能进入睡眠状态。由于这个限制，每当纤程需要等待某个条件变为真时，它要么忙等待，要么调用 SwitchToFiber 以便在等待期间将控制权交给另一个纤程。在纤程内部进行阻塞的操作系统调用通常是大忌。这样做会使纤程的封闭线程进入睡眠状态，从而阻止该纤程执行任何操作——包括调度其他纤程协同运行。

4.4.7.3 光纤迁移

纤程可以从一个线程迁移到另一个线程，但必须经过其 Inactive 状态。例如，假设在线程 A 上下文中运行的纤程 F。纤程 F 调用 SwitchToFiber(G) 来激活线程 A 中名为 G 的另一个纤程。这会使纤程 F 进入 Inactive 状态（意味着它不再与任何线程关联）。现在假设另一个名为 B 的线程正在运行纤程 H。如果纤程 H 调用 SwitchToFiber(F)，则纤程 F 已有效地从线程 A 迁移到线程 B。

4.4.7.4 使用纤程进行调试

由于纤程由操作系统提供，因此调试工具和性能分析工具应该能够“看到”它们，就像它们能够“看到”线程一样。例如，在 PS4 上使用 SN Systems 的 Visual Studio Clang 调试器插件进行调试时，纤程会自动显示在“线程”窗口中，就像它们是线程一样。您可以双击纤程以在“监视”和“调用堆栈”窗口中将其激活，然后像平常处理线程一样上下移动其调用堆栈。

如果您正在考虑在游戏引擎中使用纤程，那么在投入大量时间和精力进行基于纤程的设计之前，最好先检查一下调试器在目标平台上的功能。如果您的调试器和/或目标平台没有提供良好的纤程调试工具，那么这可能会成为交易的障碍。

4.4.7.5 关于纤维的进一步阅读

您可以在此处阅读有关 Windows 光纤的更多信息：https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx。

It's important to understand that fibers don't themselves have a Blocked state, the way threads do. In other words, it's not possible to put a fiber to sleep waiting on a condition. Only its thread can be put to sleep. Because of this restriction, whenever a fiber needs to wait for a condition to become true, it either busy-waits or it calls `SwitchToFiber` in order to yield control to another fiber while it waits. Making a blocking OS call from within a fiber is usually a pretty big no-no. Doing so would put the fiber's enclosing thread to sleep, thereby preventing that fiber from doing anything—including scheduling other fibers to run cooperatively.

4.4.7.3 Fiber Migration

A fiber can migrate from thread to thread, but only by passing through its Inactive state. As an example, consider a fiber F that is running within the context of thread A. Fiber F calls `SwitchToFiber(G)` to activate a different fiber named G inside thread A. This puts fiber F into its Inactive state (meaning it is no longer associated with any thread). Now let's assume that another thread named B is running fiber H. If fiber H calls `SwitchToFiber(F)`, then fiber F has effectively migrated from thread A to thread B.

4.4.7.4 Debugging with Fibers

Because fibers are provided by the OS, debugging tools and profiling tools should be able to "see" them, just the way they can "see" threads. For example, when debugging on the PS4 using SN Systems' Visual Studio debugger plug-in for Clang, fibers automatically show up in the Threads window as if they were threads. You can double-click a fiber to activate it within the Watch and Call Stack windows, and then walk up and down its call stack just as you normally would with a thread.

If you're considering using fibers in your game engine, it's a good idea to check out your debugger's capabilities on your target platform before you commit a lot of time and effort to a fiber-based design. If your debugger and/or target platform doesn't provide good tools for debugging fibers, that could be a deal breaker.

4.4.7.5 Further Reading on Fibers

You can read more about Windows fibers here: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx.

4.4.8 用户级线程和协程

线程和纤程都倾向于相当“重量级”，因为这些功能是由内核提供的。这意味着，大多数用于操作线程或纤程的函数都需要切换到内核空间的上下文——这可不是什么低效的操作。但是，有一些比线程和纤程更轻量级的替代方案。这些机制允许程序员以多个独立的控制流的形式进行编码，每个控制流都有自己的执行上下文，但无需进行内核调用的高昂成本。这些功能统称为用户级线程。

用户级线程完全在用户空间实现。内核对它们一无所知。每个用户级线程都由一个普通的数据结构表示，该结构跟踪线程的 ID（可能是一个人类可读的名称）以及执行上下文信息（CPU 寄存器和调用堆栈的内容）。用户级线程库提供了用于创建和销毁线程以及在线程之间切换的 API 函数。每个用户级线程都在操作系统提供的“真实”线程或纤程的上下文中运行。

实现用户级线程库的诀窍在于如何实现上下文切换。仔细想想，上下文切换主要就是交换 CPU 寄存器的内容。毕竟，寄存器包含了描述线程执行上下文所需的所有信息——包括指令指针和调用堆栈。因此，通过编写一些巧妙的汇编语言代码，就可以实现上下文切换。一旦有了上下文切换，用户级线程库的其余部分就只不过是数据管理了。

C 和 C++ 对用户级线程的支持并不完善，但确实存在一些可移植和不可移植的解决方案。POSIX 通过其 ucontext.h 头文件 (https://en.wikipedia.org/wiki/Setcontext) 提供了一组用于管理轻量级线程执行上下文的函数，但此 API 现已弃用。C++ Boost 库提供了一个可移植的用户级线程库。（在 http://www.boost.org/ 上搜索“context”可查看此库的文档。）

4.4.8.1 协程

协程是一种特殊的用户级线程，对于编写本质上异步的程序（例如 Web 服务器和游戏）非常有用！协程是子程序概念的泛化。子程序只能通过将控制权返回给调用者来退出，而协程也可以通过让位于另一个协程来退出。当一个协程让位于另一个协程时，它的执行

4.4.8 User-Level Threads and Coroutines

Both threads and fibers tend to be rather “heavy weight” because these facilities are provided by the kernel. This implies that most functions that you’d call to manipulate threads or fibers involve a context switch into kernel space—not a cheap operation. But there are lighter-weight alternatives to threads and fibers. These mechanisms allow programmers to code in terms of multiple independent flows of control, each with its own execution context, but without the high cost of making kernel calls. Collectively, these facilities are known as *user-level threads*.

User-level threads are implemented entirely in user space. The kernel knows nothing about them. Each user-level thread is represented by an ordinary data structure that keeps track of the thread’s id, possibly a human-readable name, and execution context information (the contents of CPU registers and a call stack). A user-level thread library provides API functions for creating and destroying threads, and context switching between them. Each user-level thread runs within the context of a “real” thread or fiber that has been provided by the operating system.

The trick to implementing a user-level thread library is figuring out how to implement a context switch. If you think about it, a context switch mostly boils down to swapping the contents of the CPU’s registers. After all, the registers contain all of the information needed to describe a thread’s execution context—including the instruction pointer and the call stack. So by writing some clever assembly language code, it’s possible to implement a context switch. And once you have a context switch, the rest of your user-level thread library is nothing more than data management.

User-level threads aren’t supported very well in C and C++, but some portable and non-portable solutions do exist. POSIX provided a collection of functions for managing lightweight thread execution contexts via its `ucontext.h` header file (https://en.wikipedia.org/wiki/Setcontext), but this API has since been deprecated. The C++ Boost library provides a portable user-level thread library. (Search for “context” on http://www.boost.org/ for documentation on this library.)

4.4.8.1 Coroutines

Coroutines are a particular type of user-level thread that can prove very useful for writing inherently asynchronous programs, like web servers—and games! A coroutine is a generalization of the concept of a subroutine. Whereas a subroutine can only exit by returning control to its caller, a coroutine can also exit by *yielding* to another coroutine. When a coroutine yields, its execution

上下文保存在内存中。下次调用该协程时（由其他协程让出），它会从上次中断的地方继续执行。

子程序以分层方式相互调用。子程序 A 调用 B，B 调用 C，C 返回 B，B 返回 A。但协程之间是对称调用的。协程 A 可以让位于 B，B 又可以让位于 A，如此循环往复。这种来回调用模式不会导致调用栈无限加深，因为每个协程都维护着自己的私有执行上下文（调用栈和寄存器内容）。因此，从协程 A 让位于协程 B 更像是线程间的上下文切换，而不是函数调用。但由于协程是用用户级线程实现的，因此这些上下文切换非常高效。

下面是一个系统的伪代码示例，其中一个协程不断生成由另一个协程使用的数据：

```
Queue g_queue;

coroutine void Produce()
{
    while (true)
    {
        while (!g_queue.IsFull())
        {
            CreateItemAndAddToQueue(g_queue);
        }
        YieldToCoroutine(Consume);

        // continues from here on next yield...
    }
}

coroutine void Consume()
{
    while (true)
    {
        while (!g_queue.IsEmpty())
        {
            ConsumeItemFromQueue(g_queue);
        }
        YieldToCoroutine(Produce);

        // continues from here on next yield...
    }
}
```

协程通常由 Ruby、Lua 等高级语言提供

context is maintained in memory. The next time the coroutine is *called* (by being yielded *to* by some other coroutine) it continues from where it left off.

Subroutines call each other in a heirarchical fashion. Subroutine A calls B, which calls C, which returns to B, which returns to A. But coroutines call each other symmetrically. Coroutine A can yield to B, which can yield to A, ad infinitum. This back and forth calling pattern doesn't lead to an infinitely deepening call stack, because each coroutine maintains its own private execution context (call stack and register contents). Thus yielding from coroutine A to coroutine B acts more like a context switch between threads than like a function call. But because coroutines are implemented with *user-level* threads, these context switches are very efficient.

Here's a pseudocode example of a system in which one coroutine continually produces data that is consumed by another coroutine:

```
Queue g_queue;

coroutine void Produce()
{
    while (true)
    {
        while (!g_queue.IsFull())
        {
            CreateItemAndAddToQueue(g_queue);
        }
        YieldToCoroutine(Consume);

        // continues from here on next yield...
    }
}

coroutine void Consume()
{
    while (true)
    {
        while (!g_queue.IsEmpty())
        {
            ConsumeItemFromQueue(g_queue);
        }
        YieldToCoroutine(Produce);

        // continues from here on next yield...
    }
}
```

Coroutines are most often provided by high-level languages like Ruby, Lua

以及谷歌的 Go。C 或 C++ 也可以使用协程。C++ Boost 库提供了可靠的协程实现，但 Boost 需要编译和链接相当庞大的代码库。如果您想要更精简的代码，不妨尝试编写自己的协程库。Malte Skarupke 的以下博客文章演示了这样做并不像您最初想象的那么繁重：https://probablydance.com/2013/02/20/handmade-coroutines-for-windows/。

4.4.8.2 内核线程与用户线程

“内核线程”一词有两种截然不同的含义，这在您深入了解多线程时可能会造成很大的困惑。因此，让我们来揭开这个术语的神秘面纱。这两种定义如下：

1. 在 Linux 上，“内核线程”是一种由内核自行创建供内部使用的特殊线程，仅在 CPU 处于特权模式时运行。内核还会创建供进程使用的线程（通过 pthread 或 C++11 的 std::thread 等 API）。这些线程在进程上下文中的用户空间中运行。从这个意义上讲，任何在特权模式下运行的线程都是内核线程，任何在用户模式下运行的线程（在单线程或多线程进程的上下文中）都是“用户线程”。

2. 术语“内核线程”也可用于指代任何内核已知并由内核调度的线程。根据此定义，内核线程可以在内核空间或用户空间中执行，而术语“用户线程”仅适用于完全由用户空间程序管理的控制流，内核完全不参与，例如协程。

根据定义 #2，纤程模糊了“内核线程”和“用户线程”之间的界限。一方面，内核能够感知纤程的存在，并为每个纤程维护一个单独的调用栈。另一方面，纤程不由内核调度——只有当另一个纤程或线程通过 SwitchToFiber() 之类的调用明确地将控制权移交给它时，它才能运行。

4.4.9 关于进程和线程的进一步阅读

在前面的章节中，我们已经介绍了进程、线程和纤程的基础知识，但实际上我们只是触及了皮毛。有关更多信息，请访问以下网站：

and Google's Go. It's also possible to use coroutines in C or C++. The C++ Boost library provides a solid implementation of coroutines, but Boost requires you to compile and link against a pretty huge codebase. If you want something leaner, you may want to try rolling your own coroutine library. The following blog post by Malte Skarupke demonstrates that doing so isn't quite as onerous a task as you might at first imagine: https://probablydance.com/2013/02/20/handmade-coroutines-for-windows/.

4.4.8.2 Kernel Threads versus User Threads

The term "kernel thread" has two very different meanings, and this can become a major source of confusion as you read more about multithreading. So let's demystify the term. The two definitions are as follows:

1. On Linux, a "kernel thread" is a special kind of thread created for internal use by the kernel itself, which runs only while the CPU is in privileged mode. The kernel also creates threads for use by processes (via an API such as pthread or C++11's `std::thread`). These threads run in user space within the context of a process. In this sense of the term, any thread that runs in privileged mode is a kernel thread, and any thread that runs in user mode (in the context of a single-threaded or multithreaded process) is a "user thread."

2. The term "kernel thread" can also be used to refer to any thread that is *known to* and *scheduled by* the kernel. Using this definition, a kernel thread can execute in either kernel space or user space, and the term "user thread" only applies to a flow of control that is managed entirely by a user-space program *without* the kernel being involved at all, such as a coroutine.

Using definition #2, a fiber blurs the line between "kernel thread" and "user thread." On the one hand, the kernel is aware of fibers and maintains a separate call stack for each one. On the other hand, a fiber is not scheduled by the kernel—it can run only when another fiber or thread explicitly hands control to it via a call such as `SwitchToFiber()`.

4.4.9 Further Reading on Processes and Threads

We've covered the basics of processes, threads and fibers in the preceding sections, but really we've only just scratched the surface. For more information, check out some of the following websites:

• 有关线程的介绍，请参阅https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html。

• 完整的 pthread API 文档可在线获取；只需搜索“pthread 文档”。

• 有关 Windows 线程 API 的文档，请在 https://msdn.microsoft.com/ 上搜索“进程和线程函数”。

• 有关线程调度的更多信息，请在线搜索 Nikita Ishkov 的“Linux 进程调度完整指南”。

• 要详细了解 Go 的协程实现（称为“goroutines”），请观看 Rob Pike 的演示：https://www.youtube.com/watch?v=f6kdp27TYZs。

4.5 并发编程简介

市面上有各种各样的显式并行计算硬件，但作为程序员，我们该如何利用它们呢？答案在于并发编程技术。在并发软件中，工作负载被分解成两个或多个可以半独立运行的控制流。正如我们在 4.1 节中看到的，一个系统要想符合并发的条件，必须包含多个共享数据的读取器和/或写入器。

Rob Pike 是 Google 公司的杰出工程师，专注于分布式并发系统和编程语言的研究。他将并发定义为“独立执行计算的组合”。该定义强调了这样一种观点：并发系统中的多个控制流通常是半独立运行的，但它们的计算是通过共享数据并以各种方式同步操作而组成的。

并发可以有多种形式。例如：

• 在 Linux 或 Windows 下运行的管道命令链，例如
`cat render.cpp | grep "light",`

• 由多个线程组成的单个进程，这些线程共享虚拟内存空间并对公共数据集进行操作，

• 一个线程组由在 GPU 上运行的数千个线程组成，所有线程协作渲染场景，

• 多人视频游戏，在多台 PC 或游戏机上运行的客户端之间共享共同的游戏状态。

- For an introduction to threads, see https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html.
- The full pthread API docs are available online; just search for “pthread documentation.”
- For documentation on the Windows thread API, seach for “Process and Thread Functions” on https://msdn.microsoft.com/.
- For more information on thread scheduling, search for Nikita Ishkov’s “A Complete Guide to Linux Process Scheduling” online.
- For a great introduction to Go’s implementation of coroutines (which are known as “goroutines”), watch this presentation by Rob Pike: https://www.youtube.com/watch?v=f6kdp27TYZs.

4.5 Introduction to Concurrent Programming

There’s a wide range of explicitly parallel computing hardware out there, but how can we take advantage of it as programmers? The answer lies in *concurrent programming* techniques. In concurrent software, a workload is broken down into two or more flows of control that can run semi-independently. As we saw in Section 4.1, in order for a system to qualify as concurrent, it must involve multiple readers and/or multiple writers of *shared* data.

Rob Pike, a Distinguished Engineer at Google Inc. who specializes in distributed and concurrent systems and programming languages, defines concurrency as “the composition of independently executing computations.” This definition underscores the idea that the multiple flows of control in a concurrent system normally operate semi-independently, but their computations are *composed* by *sharing data* and by *synchronizing* their operations in various ways.

Concurrency can take many forms. Some examples include:

- a piped chain of commands running under Linux or Windows, such as `cat render.cpp | grep "light",`
- a single process comprised of multiple threads that share a virtual memory space and operate on a common dataset,
- a thread group comprised of thousands of threads running on a GPU, all cooperating to render a scene,
- a multiplayer video game, sharing a common game state between clients running on multiple PCs or game consoles.

4.5.1 为什么要编写并发软件?

人们有时会编写并发程序，因为多个半独立控制流的模型比单一控制流设计更能解决问题。即使当前问题可能更适合顺序设计，也可能选择并发设计来充分利用多核计算平台。

4.5.2 并发编程模型

为了使并发程序中的各个线程能够协作，它们需要共享数据，并同步它们的活动。换句话说，它们需要通信。并发线程可以通过两种基本方式进行通信:

- 消息传递。在这种通信模式下，并发线程彼此之间传递消息，以共享数据并同步其活动。消息可以通过网络发送，使用管道在进程之间传递，或通过发送方和接收方均可访问的内存中的消息队列传输。这种方法既适用于在单台计算机上运行的线程（无论是在单个进程内还是跨多个进程），也适用于在物理上不同的计算机上运行的进程内的线程（例如，计算机集群或遍布全球的计算机网格）。

- 共享内存。在这种通信模式下，两个或多个线程被授予访问同一块物理内存的权限，因此可以直接操作该内存区域中的任何数据对象。只有当所有线程都在一台计算机上运行，□□并且该计算机拥有一组可被所有 CPU 核心“看到”的物理 RAM 时，才能直接访问共享内存。单个进程内的线程始终共享一个虚拟地址空间，因此它们可以“免费”共享内存。不同进程内的线程也可以通过将某些物理内存页面映射到所有进程的虚拟地址空间来共享内存。

有趣的是，物理上独立的计算机之间共享内存的假象可以在消息传递系统之上实现——这种技术被称为分布式共享内存。同样，消息传递机制也可以在共享内存架构之上实现，方法是在共享内存池中实现一个消息队列。

4.5.1 Why Write Concurrent Software?

Concurrent programs are sometimes written because a model of multiple semi-independent flows of control simply matches the problem better than a single flow-of-control design. A concurrent design might also be chosen to best make use of a multicore computing platform, even if the problem at hand might be more naturally suited to a sequential design.

4.5.2 Concurrent Programming Models

In order for the various threads within a concurrent program to cooperate, they need to *share data*, and they need to *synchronize* their activities. In other words, they need to *communicate*. There are two basic ways in which concurrent threads can communicate:

- *Message passing*. In this communication mode, concurrent threads pass *messages* between one another in order to share data and synchronize their activities. The messages might be sent across a network, passed between processes using a pipe, or transmitted via a message queue in memory that is accessible to both sender and receiver. This approach works both for threads running on a single computer (either within a single process or across multiple processes), and for threads within processes running on physically distinct computers (e.g., a computer cluster or a grid of machines spread across the globe).

- *Shared memory*. In this communication mode, two or more threads are granted access to the same block of physical memory, and can therefore operate directly on any data objects residing in that memory area. Direct access to shared memory only works when all threads are running on a single computer with a bank of physical RAM that can be "seen" by all CPU cores. Threads within a single process always share a virtual address space, so they can share memory "for free." Threads within different processes can also share memory by mapping certain physical memory pages into all of the processes' virtual address spaces.

It's interesting to note that the *illusion* of shared memory between physically separate computers can be implemented on top of a message-passing system—this technique is known as *distributed shared memory*. Likewise, a message-passing mechanism can be implemented on top of a shared memory architecture, by implementing a message queue that resides in the shared memory pool.

每种方法都有其优缺点。物理共享内存是共享大量数据最有效的方式，因为这些数据在线程间传输时无需复制。另一方面，正如我们将在 4.5.3 和 4.7 节中看到的那样，任何类型的资源（内存或其他资源）的共享都会带来一系列同步问题，这些问题往往难以推理，而且很难以保证程序正确性的方式来解释。消息传递设计往往会减轻（但不会消除）这些问题的影响。

本书将主要关注共享内存并发。这样做有两个原因：首先，作为游戏程序员，你最有可能遇到这种并发，因为游戏引擎通常实现为单进程多线程程序。（联网多人游戏是一个明显的例外，因为它们大量使用消息传递。）其次，共享内存并发是一个比较难理解的主题。一旦你理解了共享内存环境中的并发，消息传递技术应该会相对容易学习。

4.5.3 竞争条件

竞争条件是指程序行为依赖于时间的任何情况。换句话说，在存在竞争条件的情况下，由于各个控制流执行其任务所需的时间长度不同，当系统中发生的事件的相对顺序发生变化时，程序的行为可能会发生变化。

4.5.3.1 关键竞赛

有时竞争条件是无害的——程序的行为可能会根据时间发生一些变化，但竞争不会造成不良影响。另一方面，严重竞争是一种有可能导致程序行为错误的竞争条件。

对于没有经验的程序员来说，关键竞争导致的 bug 类型通常看起来“很奇怪”，甚至“不可能”。例如：

- 间歇性或看似随机的错误或崩溃，
- 不正确的结果，
- 数据结构陷入损坏状态，
- 当您切换到调试版本时，错误会神奇地消失，

Each approach has its pros and cons. Physically-shared memory is the most efficient way to share a large amount of data, because that data doesn't have to be copied for transmission between threads. On the other hand, as we'll see in Sections 4.5.3 and 4.7, the *sharing* of resources of any kind (memory or other resources) brings with it a host of synchronization problems that tend to be difficult to reason about, and are very tricky to account for in a manner that guarantees correctness of the program. A message-passing design tends to lessen the impacts of (but not eliminate) these kinds of problems.

In this book, we'll focus primarily on shared memory concurrency. We do this for two reasons: First, this is the kind of concurrency you're most likely to encounter as a game progammer, because game engines are typically implemented as single-process multithreaded programs. (Networked multiplayer games are a notable exception to this rule, since they make heavy use of message passing.) Second, shared memory concurrency is a more difficult topic to get your head around. Once you understand concurrency within a shared memory environment, message passing techniques should prove relatively easy to learn.

4.5.3 Race Conditions

A *race condition* is defined as any situation in which the behavior of a program is dependent on timing. In other words, in the presence of a race condition, the behavior of the program can *change* when the relative sequence of events occurring across the system changes, due to variability in the lengths of time taken by the various flows of control to perform their tasks.

4.5.3.1 Critical Races

Sometimes race conditions are innocuous—the behavior of the program may change somewhat depending on timing, but the race causes no ill effects. On the other hand, a *critical race* is a race condition that has the potential to cause *incorrect* program behavior.

The kinds of bugs caused by critical races often seem “strange” or even “impossible” to programmers who aren't experienced with them. Examples include:

- intermittent or seemingly random bugs or crashes,
- incorrect results,
- data structures that get into corrupted states,
- bugs that magically disappear when you switch to a debug build,

- 存在一段时间的 bug，然后消失几天，然后再次出现（通常是在 E3 前一天晚上！），
- 当将日志记录（又称“ printf() 调试”）添加到程序中以尝试发现问题的根源时，错误就会消失。

程序员通常将这类问题称为 Heisenbug。

4.5.3.2 数据竞争

数据争用是一种严重的竞争条件，指的是两个或多个控制流在读取和/或写入共享数据块时相互干扰，从而导致数据损坏。数据争用是并发编程的核心问题。编写并发程序的关键在于消除数据争用，具体方法是：仔细控制对共享数据的访问；或者将共享数据替换为私有的独立数据副本（从而将并发问题转化为顺序编程问题）。

为了更好地理解数据竞争，请考虑以下简单的 C/C++ 代码片段：

```
int g_count = 0;

inline void IncrementCount()
{
    ++g_count;
}
```

如果您为 Intel x86 CPU 编译此代码并查看反汇编，它将看起来像这样：

```
mov   eax,[g_count]  ; read g_count into register EAX
inc   eax            ; increment the value
mov   [g_count],eax  ; write EAX back into g_count
```

这是读取-修改-写入 (RMW) 操作的一个示例。
现在假设两个线程 A 和 B 同时调用 Increment Count() 函数（并行或通过抢占式多线程）。在正常操作下，如果每个线程只调用一次该函数，我们预期 g_count 的最终值为 2，因为要么线程 A 先递增 g_count，然后线程 B 也递增它，要么反之亦然。如表 4.1 所示。

接下来，我们考虑一下两个线程在单核机器上运行抢占式多任务的情况。假设线程 A 先运行，

- bugs that are around for a while, and then go away for a few days, only to return again (usually the night before E3!),
- bugs that go away when logging (a.k.a., “`printf()` debugging”) is added to the program in an attempt to discover the source of the problem.

Programmers often call these kinds of issues Heisenbugs.

4.5.3.2 Data Races

A *data race* is a critical race condition in which two or more flows of control intefere with one another while reading and/or writing a block of shared data, resulting in data corruption. Data races are *the* central problem of concurrent programming. Writing concurrent programs always boils down to eliminating data races, either by carefully controlling access to shared data, or by replacing shared data with private, independent copies of data (thereby transforming a concurrency problem into a sequential programming problem).

To better understand data races, consider the following simple snippet of C/C++ code:

```
int g_count = 0;

inline void IncrementCount()
{
    ++g_count;
}
```

If you compiled this code for an Intel x86 CPU and viewed the disassembly, it would look something like this:

```
mov   eax,[g_count]  ; read g_count into register EAX
inc   eax            ; increment the value
mov   [g_count],eax  ; write EAX back into g_count
```

This is an example of a *read-modify-write* (RMW) operation.

Now imagine that two threads A and B were both to call the `Increment Count()` function concurrently (either in parallel or via preemptive multithreading). Under normal operation, if each thread called the function exactly once, we'd expect the final value of `g_count` to be 2, because either thread A increments `g_count` and then thread B increments it, or vice-versa. This is illustrated in Table 4.1.

Next let's consider the case of our two threads running on a single-core machine with preemptive multitasking. Let's say that thread A runs first, and

Thread A		Thread B		Value of
Action	EAX	Action	EAX	g_count
	?		?	0
Read	0		?	0
Increment	1		?	0
Write	1		?	1
	1	Read	1	1
	1	Increment	2	1
	1	Write	2	2

表 4.1 一个简单的双线程并发程序的正确操作示例。首先，线程 A 读取共享变量的内容，增加其值，然后将结果写回共享变量。稍后，线程 B 执行相同的步骤。共享变量的最终值与预期一致，为 2。

Thread A		Thread B		Value of
Action	EAX	Action	EAX	g_count
	?		?	0
Read	0		?	0
	0	Read	0	0
	0	Increment	1	0
	0	Write	1	1
Increment	1		1	1
Write	1		1	1

表 4.2 竞争条件示例。线程 A 读取共享变量的值，但随后被线程 B 抢占，后者也读取了（相同的）值。当两个线程都完成值递增并将结果写回共享变量时，全局变量的值却变成了错误的 1，而不是预期的 2。

刚执行完第一条 mov 指令，就发生了到线程 B 的上下文切换。线程 A 并没有执行其 inc 指令，而是执行了线程 B 的第一条 mov 指令。一段时间后，线程 B 的时间片用完，内核上下文切换回线程 A，线程 A 从中断处继续执行 inc 指令。表 4.2 说明了发生的情况。提示：情况不太妙！g_count 的最终值不再是应有的 2。

Thread A		Thread B		Value of
Action	EAX	Action	EAX	g_count
	?		?	0
Read	0		?	0
Increment	1		?	0
Write	1		?	1
	1	Read	1	1
	1	Increment	2	1
	1	Write	2	2

Table 4.1. Example of correct operation of a simple two-threaded concurrent program. First thread A reads the contents of a shared variable, increments the value, and writes the results back into the shared variable. At a later time, thread B does the same steps. The final value of the shared variable is 2 as expected.

Thread A		Thread B		Value of
Action	EAX	Action	EAX	g_count
	?		?	0
Read	0		?	0
	0	Read	0	0
	0	Increment	1	0
	0	Write	1	1
Increment	1		1	1
Write	1		1	1

Table 4.2. Example of a race condition. Thread A reads the value of the shared variable, but then is preempted by thread B, which also reads the (same) value. By the time both threads have incremented the value and written their results back to the shared variable, the global variable contains the incorrect value 1 instead of the exected value of 2.

has just finished executing the first `mov` instruction when a context switch to thread B occurs. Instead of thread A executing its `inc` instruction, thread B runs its first `mov` instruction. After some time thread B's quantum expires, and the kernel context switches back to thread A, which continues where it left off and executes the `inc` instruction. Table 4.2 illustrates what happens. Hint: It's not good! The final value of `g_count` is no longer 2 as it should be.

图 4.27。读取-修改-写入操作中可能发生数据争用的三种方式。上图：两个线程在单个 CPU 核心上争用。中图：两个线程在两个独立核心上重叠，且偏移一条指令。下图：两个线程在两个核心上完美同步重叠。

如果我们在并行硬件上运行两个线程，可能会出现类似的错误，尽管原因略有不同。与单核情况一样，我们可能会很幸运：两个读取-修改-写入操作可能根本不重叠，结果将是正确的。但是，如果两个读取-修改-写入操作重叠（彼此偏移或完全同步），则两个线程最终可能会将相同的 g_count 值加载到各自的 EAX 寄存器中。两者都会增加该值，并将其写入内存。一个线程会覆盖另一个线程的结果，但这并不重要——因为它们都加载了相同的初始值，所以 g_count 的最终值最终会不正确，就像在单核场景中一样。图 4.27 展示了三种数据争用场景（抢占、偏移重叠和完全同步）。

4.5.4 关键操作和原子性

每当一个操作被另一个操作中断时，就有可能出现数据竞争错误。然而，并非所有中断都会导致错误。例如，如果一个线程正在对一块数据执行操作，而该数据块只能由该线程“看到”，则不会发生数据竞争。这样的操作可以随时被任何其他操作中断，而不会产生任何后果。

Figure 4.27. Three ways in which a data race can occur within a read-modify-write operation. Top: Two threads racing on a single CPU core. Middle: Two threads overlapping on two separate cores and offset by one instruction. Bottom: Two threads overlapping in perfect synchronization on two cores.

If we run our two threads on parallel hardware, a similar bug can occur, although for a slightly different reason. As in the single-core case, we might get lucky: The two read-modify-write operations might not overlap at all, and the result will be correct. However, if the two read-modify-write operations overlap, either offset from one another or in perfect synchronization, both threads can end up loading the same value of `g_count` into their respective EAX registers. Both will increment the value, and both will write it to memory. One thread will overwrite the results of the other, but it doesn't really matter—because they both loaded the same initial value, the final value of `g_count` will end up being incorrect, just as it was in the single-core scenario. The three data race scenarios (preemption, offset overlap, and perfect synchronization) are illustrated in Figure 4.27.

4.5.4 Critical Operations and Atomicity

Whenever one operation is interrupted by another, we have the potential for a data race bug. However, not all interruptions actually cause bugs. For example, if a thread is performing an operation on a chunk of data that can only be "seen" by that one thread, no data race can occur. Such an operation can be interrupted at any moment by any other operation without consequence.

图 4.28。由于算法中每个步骤的执行时间有限，因此在多线程程序中，很难回答各步骤的相对顺序问题。例如，操作 B 是在操作 C 之前还是之后发生？

同样，如果一个数据对象的操作被另一个数据对象的操作中断，这两个操作之间也不会互相干扰，因此也就不可能出现数据竞争错误。只有当一个共享对象上的操作被另一个共享对象上的操作中断时，才会发生数据竞争错误。因此，我们只能针对特定的共享数据对象来讨论数据竞争问题。

我们用“关键操作”来指代任何可能读取或修改特定共享对象的操作。为了保证共享对象不存在数据竞争错误，我们必须确保其关键操作之间不能相互干扰。当关键操作以这种方式不可中断时，它被称为原子操作。或者，我们可以说这样的操作具有原子性。

4.5.4.1 调用和响应

我们初学编程时，通常会被告知，算法中每一步的执行时间与算法的正确性无关——重要的是这些步骤是否按正确的顺序执行。这个简单的模型对于顺序（单线程）程序来说非常有效。但在多线程环境下，当一组操作的执行时间有限时，我们无法定义它们的执行顺序。图 4.28 演示了这一思路。

在并发系统中，定义“顺序”概念的唯一方法是将我们自己限制在瞬时事件的讨论上。给定任何一对瞬时事件，只有三种可能性：事件 A 发生在事件 B 之前，事件 A 发生在事件 B 之后，或者这两个事件同时发生。（完全同时发生的事件很少见，但它们可能发生在多核计算机中，其中部分或所有核心共享同步时钟。）任何具有有限持续时间的操作都可以分解为两个瞬时事件——它的调用（操作开始的时刻）和

6 仅当两个对象位于不同的缓存行上时，这才是严格正确的。

Figure 4.28. Because each step in an algorithm takes a finite amount of time to be performed, it becomes difficult to answer questions about the relative ordering of the steps in a multithreaded program. For example, does operation B happen before or after operation C?

Likewise, if an operation on one data object is interrupted by an operation on a different object, there's no way for those two operations to interefere with one another,[6] and hence no data race bugs are possible. Data race bugs only occur when an operation on a shared object is interrupted by another operation on that *same* object. So we can only talk meaningfully about data races in relation to a particular shared data object.

Let's use the term *critical operation* to refer to any operation that can possibly read or mutate one particular shared object. To guarantee that the shared object is free from data race bugs, we must ensure that none of its critical operations can interrupt one another. When a critical operation is made uninterruptable in this manner, it is called an *atomic operation*. Alternatively, we can say that such an operation has the property of *atomicity*.

4.5.4.1 Invocation and Response

When we first learn to program, we're usually taught that the *time* required to perform each step in an algorithm is not relevant to the correctness of the algorithm—all that matters is that the steps are performed in the proper *order*. This simple model works well for sequential (single-threaded) programs. But in the presence of multiple threads, it's not possible to define the order of a set of operations when those operations each have a finite duration. This idea is illustrated in Figure 4.28.

In a concurrent system, the only way to define the notion of *order* is to restrict ourselves to talking about *instantaneous events*. Given any pair of instantaneous events, there are only three possibilities: event A happens before event B, event A happens after event B, or the two events are simultaneous. (Perfectly simultaneous events are rare, but they can occur in a multicore computer in which some or all cores share a synchronized clock.)

Any operation with a finite duration can be broken down into two instantaneous events—its *invocation* (the moment at which the operation begins) and

[6] This is only strictly true if the two objects reside on different cache lines.

图 4.29 任何包含关键操作的代码片段都可以分为三个部分。关键操作本身的上界是其调用，下界是其响应。

图 4.30 另一个例子，这次是汇编语言，代码片段被分成三个部分，由关键操作的调用和响应限定。

它的响应（即被认为完成的时刻）。当我们查看任何包含对某个共享数据对象的关键操作的代码片段时，我们可以将其分为三个部分，瞬时调用和响应事件标记了它们之间的界限。请注意，我们这里讨论的是事件在源代码中发生的顺序——这被称为程序顺序。

- 前导部分：按程序顺序，在关键操作调用之前发生的所有代码。
- 关键部分：包□□含关键操作本身的代码。
- 后同步部分：按照程序顺序，在关键操作响应之后发生的所有代码。

图 4.29 和 4.30 说明了将代码块划分为三个部分的概念。

Figure 4.29. Any code snippet that includes a critical operation can be partitioned into three sections. The critical operation itself is bounded above by its invocation, and below by its response.

Figure 4.30. Another example, this time in assembly language, of a code snippet partitioned into three sections, bounded by the invocation and response of a critical operation.

its *response* (the moment at which it is considered to be complete). When we look at any code snippet that includes a critical operation on some shared data object, we can thus divide it into three sections, with the instantaneous invocation and response events demarking the boundaries between them. Note that we're talking here about the order in which events occur *as they were written* in the source code—this is known as *program order*.

- *Preamble section:* All code that occurs before the critical operation's invocation, in program order.
- *Critical section:* The code that comprises the critical operation itself.
- *Postamble section:* All code that occurs after the critical operation's response, in program order.

This notion of partitioning a block of code into three sections is illustrated in Figures 4.29 and 4.30.

4.5.4.2 原子性定义

正如我们在 4.5.3.2 节中看到的，当一个关键操作被同一共享对象上的另一个关键操作打断时，可能会发生数据竞争错误。这种情况可能发生：

- 当一个线程在单个核心上抢占另一个线程时，或者
- 当两个或多个关键操作跨多个核心重叠时。

从调用和响应的角度思考，我们可以更精确地定义中断的一般概念：每当一个操作的调用和/或响应发生在另一个操作的调用和响应之间时，就会发生中断。但正如我们所说，并非所有中断都会导致数据争用错误。特定共享对象上的关键操作，只有当其调用和响应被同一对象上的另一个关键操作中断时，才会受到数据争用的影响。因此，我们可以将关键操作的原子性定义如下：

> 如果某个关键操作的调用和响应没有被同一对象上的另一个关键操作打断，则可以说该关键操作已经原子执行。

这里需要强调的是，关键操作被其他非关键操作或影响其他不相关数据对象的关键操作打断是完全正常的。只有当同一对象上的两个关键操作相互打断时，才会发生数据竞争错误。图 4.31 展示了各种情况——其中一种情况是关键操作以原子方式成功执行，另外三种情况是关键操作未能成功执行。

如果我们能让一个关键操作从系统中所有其他线程的角度来看，看起来像是瞬间发生的，那么我们就能保证该操作将以原子方式执行。换句话说，操作的调用和响应必须看起来像是同时发生的，或者关键操作本身的持续时间为零。这样，其他关键操作的调用或响应就不可能“潜入”到该操作的调用和响应之间。

4.5.4.3 使操作原子化

但是，如何将关键操作转换为原子操作呢？最简单、最可靠的方法是使用一个叫做互斥锁（mutex）的特殊对象。互斥锁是操作系统提供的一个对象，其作用类似于挂锁，可以由线程锁定和解锁。给定两个关键操作

4.5.4.2 Atomicity Defined

As we saw in Section 4.5.3.2, a data race bug can occur when a critical operation is interrupted by another critical operation on the same shared object. This can happen:

- when one thread preempts another on a single core, or
- when two or more critical operations overlap across multiple cores.

Thinking in terms of invocation and response, we can pin down the general notion of interruption a bit more precisely: An *interruption* occurs whenever the invocation and/or response of one operation occurs *between* the invocation and response of another operation. But as we've said, not all interruptions lead to data race bugs. A critical operation on a particular shared object can only be affected by a data race if its invocation and response are interrupted by another critical operation on that *same* object. Therefore, we can define the *atomicity* of a critical operation as follows:

> A critical operation can be said to have executed *atomically* if its invocation and response are not interrupted by another critical operation on that same object.

We should stress here that it's perfectly fine for a critical operation to be interrupted by other noncritical operations, or by critical operations affecting other unrelated data objects. Only when two critical operations on the *same* object interrupt one another does a data race bug occur. Figure 4.31 illustrates various cases—one in which a critical operation succeeds in executing atomically, and three in which it does not.

We can guarantee that a critical operation will be executed atomically if we can make it appear, from the point of view of all other threads in the system, to have occurred *instantaneously*. In other words, it must appear as though the invocation and response of the operation are simultaneous, or that the critical operation itself has a zero duration. That way, there can be no possibility of another critical operation's invocation or response "sneaking in" between the invocation and response of the operation in question.

4.5.4.3 Making an Operation Atomic

But how can we transform a critical operation into an atomic operation? The easiest and most reliable way to accomplish this is to use a special object called a *mutex*. A mutex is an object provided by the operating system that acts like a padlock, in that it can be locked and unlocked by a thread. Given two critical

图 4.31。上图：关键操作 A 可以说是原子执行的，因为它没有被同一共享对象上其他关键操作的调用或响应打断。下图：三种场景中，关键操作 A 以非原子方式执行，因为它被同一对象上其他关键操作的调用和/或响应打断。

对于特定共享数据对象的操作，我们通过获取互斥锁来保护每个操作的调用，并在每个操作响应时释放互斥锁。由于操作系统保证一个互斥锁每次只能被一个线程获取，因此我们可以确保一个操作的调用或响应永远不会发生在另一个操作的调用和响应之间。从并发系统中事件的全局顺序来看，使用互斥锁保护的关键操作似乎是瞬时的。

互斥锁是操作系统提供的并发工具集合的一部分，称为线程同步原语。我们将在 4.6 节中探讨线程同步原语。

4.5.4.4 原子性作为序列化

假设有一组线程，所有线程都试图对一个共享数据对象执行单个操作。如果没有原子性，这些操作可能会同时发生，或者随着时间的推移，它们可能会以各种不可预测的方式重叠。

如图 4.32 所示。

然而，将操作原子化可以保证在任何给定时刻只有一个线程执行该操作。这具有序列化操作的效果——以前是一堆重叠的操作，

Figure 4.31. Top: Critical operation A can be said to have executed atomically, because it was not interrupted by any other invocations or responses from critical operations on the same shared object. Bottom: Three scenarios in which critical operation A executed nonatomically because it was interrupted by the invocation and/or response of another critical operation on the same object.

operations on a particular shared data object, we guard the invocation of each operation with the acquistion of the mutex, and release the mutex at each one's response. Because the OS guarantees that a mutex can only be acquired by one thread at a time, we can thus be certain that the invocation or response of one operation can never happen in between the invocation and response of the other. From the point of view of the global ordering of the events in a concurrent system, a critical operation guarded with a mutex lock *appears* to be instantaneous.

Mutexes are part of a collection of concurrency tools provided by the operating system known as *thread synchronization primitives*. We'll explore thread synchronization primitives in Section 4.6.

4.5.4.4 Atomicity as Serialization

Consider a group of threads, all attempting to perform a single operation on a shared data object. Without atomicity, these operations might happen simultaneously, or they might overlap in all sorts of unpredictable ways over time. This is illustrated in Figure 4.32.

However, making the operation atomic guarantees that only one thread will ever be performing it at any given moment in time. This has the effect of *serializing* the operations—what used to be a jumble of overlapping opera-

图 4.32。如果没有原子性，多个线程执行的操作可能会随着时间的推移以不可预测的方式重叠。

图 4.33 通过将每个关键操作包装在互斥锁-解锁对中，我们强制操作按顺序执行。

操作被转换为有序的原子操作序列。将操作原子化后，我们无法控制最终的顺序；我们唯一能确定的是，操作将按照某种顺序执行。图 4.33 演示了这一思路。

4.5.4.5 以数据为中心的一致性模型

并发系统中的原子性和操作序列化概念属于一个更大的主题，即以数据为中心的一致性模型。一致性模型是数据存储（例如并发系统中的共享数据对象或分布式系统中的数据库）与共享该数据存储的线程集合之间的契约。它使推断数据存储的行为变得更容易——只要线程遵循契约规则，程序员就可以确保数据存储的行为一致且可预测，并且其数据不会损坏。

能够保证原子性的数据存储可以说是可线性化的。以数据为中心的一致性这个话题略微超出了我们的讨论范围，但您可以在线阅读更多相关内容。以下是一些不错的入门资料：

Figure 4.32. Without atomicity, operations performed by multiple threads can overlap in unpredictable ways over time.

Figure 4.33. By wrapping each critical operation in a mutex lock-unlock pair, we force the operations to execute *sequentially*.

tions is transformed into an orderly sequential sequence of atomic operations. Making an operation atomic gives us no control over what the order will end up being; all we can say for certain is that the operations will be performed in *some* sequential order. This idea is illustrated in Figure 4.33.

4.5.4.5 Data-Centric Consistency Models

The concepts of atomicity and the serialization of operations within a concurrent system are part of a larger topic known as *data-centric consistency models*. A consistency model is a *contract* between a data store, such as a shared data object in a concurrent system or a database in a distributed system, and a collection of threads that share that data store. It makes reasoning about the behavior of the data store easier—as long as the threads follow the rules of the contract, the programmer can be certain that the data store will behave in a consistent and predictable manner, and its data will not become corrupted.

A data store that provides a guarantee of atomicity can be said to be *linearizable*. The topic of data-centric consistency is a bit beyond our scope here, but you can read more about it online. Here are a few good places to start:

• 在维基百科上搜索“一致性模型”和“线性化”；

•https://www.cse.buffalo.edu/~stevko/courses/cse486/spring13/lectures/26-consistency2.pdf；

•http://www.cs.cmu.edu/~srini/15-446/S09/lectures/10-consistency.pdf。

4.6 线程同步原语

每个支持并发的操作系统都提供了一套称为线程同步原语的工具。这些工具为并发程序员提供两项服务：

1. 通过使关键操作原子化，实现线程之间共享资源的能力。

2.同步两个或多个线程的操作的能力：

   a. 通过使线程在等待资源可用或等待一个或多个其他线程完成任务时进入睡眠状态，并且

   b. 通过启用正在运行的线程来唤醒一个或多个休眠线程并通知它们。

这里需要注意的是，虽然这些线程同步原语健壮且相对易于使用，但它们通常非常昂贵。这是因为这些工具是由内核提供的。因此，与其中任何一个进行交互都需要进行内核调用，这需要上下文切换到保护模式。这种上下文切换可能要花费多达 1000 个时钟周期。由于成本高昂，一些并发程序员更倾向于实现自己的原子性和同步工具，或者转向无锁编程来提高并发软件的效率。然而，对这些同步原语的透彻理解是任何并发程序员工具包的重要组成部分。

4.6.1 互斥锁

互斥锁是一个操作系统对象，它允许将关键操作原子化。互斥锁可以处于以下两种状态之一：解锁或锁定。（这两种状态有时分别称为释放和获取，或发出信号和未发出信号。）

- Search for “consistency model” and “linearizability” on Wikipedia;
- https://www.cse.buffalo.edu/~stevko/courses/cse486/spring13/lectures/26-consistency2.pdf;
- http://www.cs.cmu.edu/~srini/15-446/S09/lectures/10-consistency.pdf.

4.6 Thread Synchronization Primitives

Every operating system that supports concurrency provides a suite of tools known as *thread synchronization primitives*. These tools provide two services to concurrent programmers:

1. The ability to share resources between threads by making critical operations *atomic*.
2. The ability to *synchronize* the operation of two or more threads:
   a. by enabling a thread to *go to sleep* while it waits for a resource to become available or for one or more other threads to complete a task, and
   b. by enabling a running thread to *notify* one or more sleeping threads by waking them up.

We should note here that while these thread synchronization primitives are robust and relatively easy to use, they are generally quite expensive. This is because these tools are provided by the kernel. Interacting with any of them therefore requires a *kernel call*, which involves a context switch into protected mode. Such context switches can cost upwards of 1000 clock cycles. Because of their high cost, some concurrent programmers prefer to implement their own atomicity and synchronization tools, or they turn to *lock-free programming* to improve the efficiency of their concurrent software. Nevertheless, a solid understanding of these synchronization primitives is an important part of any concurrent programmer’s toolkit.

4.6.1 Mutexes

A mutex is an operating system object that allows critical operations to be made atomic. A mutex can be in one of two states: *unlocked* or *locked*. (These two states are sometimes called *released* and *acquired*, or *signaled* and *nonsignaled*, respectively.)

互斥锁最重要的属性是它保证在任何给定时刻只有一个线程持有它的锁。因此，如果我们将某个共享数据对象的所有关键操作都封装在一个互斥锁中，那么这些操作彼此之间就具有原子性。换句话说，这些操作是互斥的。这就是“互斥锁”名称的由来——它是“mutual exclusion”（互斥）的缩写。

互斥锁可以用常规 C++ 对象或不透明内核对象的句柄来表示。它的 API 通常包含以下函数：

1. create() 或 init() 。创建互斥锁的函数调用或类构造函数。

2. destroy() 。销毁互斥锁□□的函数调用或析构函数。

3. lock() 或 acquire() 。一个阻塞函数，它代表调用线程锁定互斥锁，但如果锁当前被另一个线程持有，则使该线程进入睡眠状态（参见 4.4.6.4 节）。

4. try_lock() 或 try_acquire() 。这是一个非阻塞函数，尝试锁定互斥锁，但如果无法获取锁，则立即返回。

5. unlock() 或 release() 。一个非阻塞函数，用于释放互斥锁。在大多数操作系统中，只有锁定互斥锁的线程才被允许解锁它。

当互斥锁被系统中的线程锁定时，我们称它处于非信号状态。当线程释放锁时，互斥锁变为信号状态。如果一个或多个其他线程处于休眠状态（阻塞）并等待互斥锁，则向其发出信号的操作会导致内核选择其中一个等待线程并将其唤醒。在某些操作系统中，线程可以显式等待内核对象（例如互斥锁）变为信号状态。在 Windows 系统中，WaitForSingleObject() 和 WaitForMultipleObjects() 操作系统调用就是用于此目的。

4.6.1.1 POSIX

现在我们已经了解了互斥锁的工作原理，让我们看几个例子。POSIX 线程库通过 C 风格的函数式接口暴露内核互斥锁对象。下面我们将使用它来将 4.5.3.2 节中的共享计数器示例转换为原子操作：

```
#include <pthread.h>

int g_count = 0;
pthread_mutex_t g_mutex;
```


The most important property of a mutex is that it guarantees that only one thread will ever be holding a lock on it at any given time. So, if we wrap all critical operations for a particular shared data object in a mutex lock, those operations become atomic relative to one another. In other words, the operations become *mutually exclusive*. This is where the name “mutex” comes from—it’s short for “mutual exclusion.”

A mutex is represented either by a regular C++ object or by a handle to an opaque kernel object. Its API is typically comprised of the following functions:

1. `create()` or `init()`. A function call or class constructor that creates the mutex.
2. `destroy()`. A function call or destructor that destroys the mutex.
3. `lock()` or `acquire()`. A blocking function that locks the mutex on behalf of the calling thread, but puts the thread to sleep (see Section 4.4.6.4) if the lock is currently held by another thread.
4. `try_lock()` or `try_acquire()`. A non-blocking function that attempts to lock the mutex, but returns immediately if the lock cannot be acquired.
5. `unlock()` or `release()`. A non-blocking function that releases the lock on the mutex. In most operating systems, only the thread that locked a mutex is permitted to unlock it.

When a mutex is locked by a thread in the system, we say it is in a *non-signaled* state. When the thread releases the lock, the mutex becomes *signaled*. If one or more other threads is asleep (blocked) waiting on the mutex, the act of signaling it causes the kernel to select one of these waiting threads and wake it up. In some operating systems, it’s possible for a thread to explicitly wait for a kernel object such as a mutex to become signaled. Under Windows, the `WaitForSingleObject()` and `WaitForMultipleObjects()` OS calls serve this purpose.

4.6.1.1 POSIX

Now that we have an understanding of how mutexes work, let’s take a look at a few examples. The POSIX thread library exposes kernel mutex objects via a C-style functional interface. Here’s how we’d use it to turn our shared counter example from Section 4.5.3.2 into an atomic operation:

```
#include <pthread.h>

int g_count = 0;
pthread_mutex_t g_mutex;
```

```
inline void IncrementCount()
{
    pthread_mutex_lock(&g_mutex);
    ++g_count;
    pthread_mutex_unlock(&g_mutex);
}
```

请注意，为了清晰和简洁起见，我们省略了通常由主线程执行的代码，该代码在生成将使用互斥锁的线程之前调用 pthread_mutex_init() 来初始化互斥锁，并在所有其他线程退出后调用 pthread_mutex_destroy() 来销毁互斥锁□□。

4.6.1.2 C++标准库

从 C++11 开始，C++ 标准库通过 std::mutex 类公开了内核互斥锁。下面演示了如何使用它来实现共享计数器的原子增量：

```
#include <mutex>

int g_count = 0;
std::mutex g_mutex;

inline void IncrementCount()
{
    g_mutex.lock();
    ++g_count;
    g_mutex.unlock();
}
```

std::mutex 类的构造函数和析构函数处理底层内核互斥对象的初始化和销毁□□，使其比 pthread_mutex_t 更容易使用。

4.6.1.3 Windows

在 Windows 下，互斥锁由一个不透明的内核对象表示，并通过句柄引用。使用通用的 WaitForSingleObject() 函数，可以通过等待互斥锁发出信号来“锁定”它。调用 ReleaseMutex() 可以解锁互斥锁。使用 Windows 互斥锁重写我们的简单示例，并再次省略互斥锁对象的创建和销毁细节，我们得到以下代码：


```
inline void IncrementCount()
{
    pthread_mutex_lock(&g_mutex);
    ++g_count;
    pthread_mutex_unlock(&g_mutex);
}
```

Note that in the interest of clarity and brevity, we've omitted the code, normally executed by the main thread, that calls `pthread_mutex_init()` to initialize the mutex prior to spawning the threads that will use it, and that calls `pthread_mutex_destroy()` to destroy the mutex once all other threads have exited.

4.6.1.2 C++ Standard Library

Starting with C++11, the C++ standard library exposes kernel mutexes via the class `std::mutex`. Here's how we'd use it to make the increment of a shared counter atomic:

```
#include <mutex>

int g_count = 0;
std::mutex g_mutex;

inline void IncrementCount()
{
    g_mutex.lock();
    ++g_count;
    g_mutex.unlock();
}
```

The constructor and destructor of the `std::mutex` class handles the initialization and destruction of the underlying kernel mutex object, making it a little bit easier to use than `pthread_mutex_t`.

4.6.1.3 Windows

Under Windows, a mutex is represented by an opaque kernel object and referenced through a handle. A mutex is "locked" by waiting for it to become signaled, using the general-purpose `WaitForSingleObject()` function. Unlocking a mutex is accomplished by calling `ReleaseMutex()`. Rewriting our simple example using Windows mutexes, and again omitting the details of the creation and destruction of the mutex object, we arrive at the following code:

```
#include <windows.h>

int g_count = 0;
HANDLE g_hMutex;

inline void IncrementCount()
{
    if (WaitForSingleObject(g_hMutex, INFINITE)
        == WAIT_OBJECT_0)
    {
        ++g_count;
        ReleaseMutex(g_hMutex);
    }
    else
    {
        // learn to deal with failure...
    }
}
```

4.6.2 临界区

在大多数操作系统中，互斥锁可以在进程之间共享。因此，它是一种由内核内部管理的数据结构。这意味着对互斥锁执行的所有操作都涉及内核调用，因此会导致 CPU 上下文切换到保护模式。这使得互斥锁的开销相对较大，即使没有其他线程竞争锁。

一些操作系统提供了比互斥锁更便宜的替代品。例如，Microsoft Windows 提供了一种称为临界区的锁定机制。其术语和 API 看起来与互斥锁略有不同，但 Windows 下的临界区实际上只是一个低成本的互斥锁。

临界区的 API 如下所示：

1. InitializeCriticalSection(). 构造一个临界区对象。

2. DeleteCriticalSection(). 销毁已初始化的临界区对象。

3. EnterCriticalSection() . 一个阻塞函数，它代表调用线程锁定关键部分，但如果锁当前由另一个线程持有，则该函数将处于忙等待状态或使该线程进入睡眠状态。

4. TryEnterCriticalSection() . 一个非阻塞函数，尝试锁定关键部分，但如果无法获取锁，则立即返回。


```
#include <windows.h>

int g_count = 0;
HANDLE g_hMutex;

inline void IncrementCount()
{
    if (WaitForSingleObject(g_hMutex, INFINITE)
        == WAIT_OBJECT_0)
    {
        ++g_count;
        ReleaseMutex(g_hMutex);
    }
    else
    {
        // learn to deal with failure...
    }
}
```

4.6.2 Critical Sections

In most operating systems, a mutex can be shared between processes. As such, it is a data structure that is managed internally by the kernel. This means that all operations performed on a mutex involve a kernel call, and hence a context switch into protected mode on the CPU. This makes mutexes relatively expensive, even when no other threads are contending for the lock.

Some operating systems provide less-expensive alternatives to a mutex. For example, Microsoft Windows provides a locking mechanism known as a *critical section*. The terminology and API look a bit different to that of a mutex, but a critical section under Windows is really just a low-cost mutex.

The API of a critical section looks like this:

1. `InitializeCriticalSection()`. Constructs a critical section object.
2. `DeleteCriticalSection()`. Destroys an initialized critical section object.
3. `EnterCriticalSection()`. A blocking function that locks a critical section on behalf of the calling thread, but busy-waits or puts the thread to sleep if the lock is currently held by another thread.
4. `TryEnterCriticalSection()`. A non-blocking function that attempts to lock a critical section, but returns immediately if the lock cannot be acquired.

5. LeaveCriticalSection(). 一个非阻塞函数，用于释放临界区对象上的锁。

下面展示了如何使用 Windows 临界区 API 实现原子增量：

```
#include <windows.h>

int g_count = 0;
CRITICAL_SECTION g_critsec;

inline void IncrementCount()
{
    EnterCriticalSection(&g_critsec);
    ++g_count;
    LeaveCriticalSection(&g_critsec);
}
```

和之前一样，我们省略了一些细节。主线程通常会在生成使用临界区的线程之前初始化临界区，并且当然会在所有线程退出后清理它。

临界区成本低是如何实现的？当一个线程首次尝试进入（锁定）已被另一个线程锁定的临界区时，会使用一个成本低廉的自旋锁来等待，直到另一个线程离开（解锁）该临界区。自旋锁不需要上下文切换到内核，因此比互斥锁节省几千个时钟周期。只有当线程忙等待时间过长时，才会像使用常规互斥锁一样进入睡眠状态。这种成本较低的方法之所以有效，是因为临界区与互斥锁不同，它不能跨进程共享。我们将在 4.9.7 节中更深入地讨论自旋锁。

其他一些操作系统也提供了“廉价”的互斥体变体。例如，Linux 支持一种名为“futex”的东西，其作用类似于 Windows 下的临界区。它的使用超出了我们的讨论范围，但你可以在 https://www.akkadia.org/drepper/futex.pdf 上阅读更多关于 futex 的内容。

4.6.3 条件变量

在并发编程中，我们经常需要在线程之间发送信号来同步它们的活动。一个例子就是我们在 4.4.8.1 节中介绍过的无处不在的生产者-消费者问题。在这个问题中，我们有两个线程：一个生产者线程计算或以其他方式生成一些数据，而该数据被一个消费者线程读取并供其使用。显然，消费者线程在生产者线程完成之前无法消费数据。

5. `LeaveCriticalSection()`. A non-blocking function that releases the lock on a critical section object.

Here's how we'd implement an atomic increment using the Windows critical section API:

```
#include <windows.h>

int g_count = 0;
CRITICAL_SECTION g_critsec;

inline void IncrementCount()
{
    EnterCriticalSection(&g_critsec);
    ++g_count;
    LeaveCriticalSection(&g_critsec);
}
```

As before, we've omitted some details. The main thread normally initializes the critical section prior to spawning the threads that use it, and would of course clean it up once the threads have all exited.

How is the low cost of a critical section achieved? When a thread first attempts to enter (lock) a critical section that is already locked by another thread, an inexpensive *spin lock* is used to wait until the other thread has left (unlocked) that critical section. A spin lock does not require a context switch into the kernel, making it a few thousand clock cycles cheaper than a mutex. Only if the thread busy-waits for too long is the thread put to sleep, as it would be with a regular mutex. This less-expensive approach works because, unlike a mutex, a critical section cannot be shared across process boundaries. We'll discuss spin locks in more depth in Section 4.9.7.

Some other operating systems provide "cheap" mutex variants as well. For example, Linux supports a thing called a "futex" that acts somewhat like a critical section under Windows. Its use is beyond our scope here, but you can read more about futexes at https://www.akkadia.org/drepper/futex.pdf.

4.6.3 Condition Variables

In concurrent programming, we often need to send signals between threads in order to synchronize their activities. One example of this is the ubiquitous producer-consumer problem which we introduced in Section 4.4.8.1. In this problem, we have two threads: A *producer* thread calculates or otherwise generates some data, and that data is read and put to use by a *consumer* thread. Obviously the consumer thread cannot consume the data until the producer

已经生成了数据。因此，生产者线程需要一种方法来通知消费者其数据已准备好供消费。

我们可以考虑使用全局布尔变量作为信号机制。以下代码片段使用 POSIX 线程演示了这个想法。
（为了清楚起见，省略了一些细节。）

```
Queue           g_queue;
pthread_mutex_t g_mutex;
bool            g_ready = false;

void* ProducerThread(void*)
{
    // keep on producing forever...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // fill the queue with data
        ProduceDataInto(&g_queue);

        g_ready = true;
        pthread_mutex_unlock(&g_mutex);

        // yield the remainder of my timeslice
        // to give the consumer a chance to run
        pthread_yield();
    }
    return nullptr;
}

void* ConsumerThread(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        while (true)
        {
            // read the value into a local,
            // making sure to lock the mutex
            pthread_mutex_lock(&g_mutex);
            const bool ready = g_ready;
            pthread_mutex_unlock(&g_mutex);

            if (ready)
                break;
```


has produced it. Hence the producer thread needs a way to notify the consumer that its data is ready for consumption.

We could consider using a global Boolean variable as a signalling mechanism. The following code snippet illustrates the idea, using POSIX threads. (Some details have been omitted for clarity.)

```
Queue           g_queue;
pthread_mutex_t g_mutex;
bool            g_ready = false;

void* ProducerThread(void*)
{
    // keep on producing forever...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // fill the queue with data
        ProduceDataInto(&g_queue);

        g_ready = true;
        pthread_mutex_unlock(&g_mutex);

        // yield the remainder of my timeslice
        // to give the consumer a chance to run
        pthread_yield();
    }
    return nullptr;
}

void* ConsumerThread(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        while (true)
        {
            // read the value into a local,
            // making sure to lock the mutex
            pthread_mutex_lock(&g_mutex);
            const bool ready = g_ready;
            pthread_mutex_unlock(&g_mutex);

            if (ready)
                break;
```

```
        }

        // consume the data
        pthread_mutex_lock(&g_mutex);
        ConsumeDataFrom(&g_queue);
        g_ready = false;
        pthread_mutex_unlock(&g_mutex);

        // yield the remainder of my timeslice
        // to give the producer a chance to run
        pthread_yield();
    }
    return nullptr;
}
```

除了这个例子本身有点设计缺陷之外，它还有一个大问题：消费者线程会不停地循环，轮询 g_ready 的值。正如我们在 4.4.6.4 节中讨论的那样，这种忙等待机制会浪费宝贵的 CPU 周期。

理想情况下，我们希望有一种方式，在生产者工作时阻塞消费者线程（使其进入睡眠状态），然后在数据准备好被消费时将其唤醒。这可以通过使用一种称为条件变量 (CV) 的新内核对象来实现。

条件变量实际上并不是一个存储条件的变量。相反，它是一个等待（休眠）线程的队列，并结合了一种机制，允许正在运行的线程在其选择的时间唤醒休眠线程。（也许“等待队列”更适合这些。）休眠和唤醒操作是在程序提供的互斥锁的帮助下以原子方式执行的，内核也提供了一些帮助。

条件变量的 API 通常如下所示：

1. create() 或 init() 。创建条件变量的函数调用或类构造函数。

2. destroy() 。用于销毁条件变量的函数调用或析构函数。

wait() . 阻塞函数，使调用线程进入睡眠状态。

4.notify() . 一个非阻塞函数，用于唤醒当前因等待条件变量而处于休眠状态的所有线程。

让我们使用 CV 重写简单的生产者-消费者示例：

```
Queue           g_queue;
pthread_mutex_t g_mutex;
```



```
        }

        // consume the data
        pthread_mutex_lock(&g_mutex);
        ConsumeDataFrom(&g_queue);
        g_ready = false;
        pthread_mutex_unlock(&g_mutex);

        // yield the remainder of my timeslice
        // to give the producer a chance to run
        pthread_yield();
    }
    return nullptr;
}
```

Besides the fact that this example is somewhat contrived, there's one big problem with it: The consumer thread spins in a tight loop, polling the value of `g_ready`. As we discussed in Section 4.4.6.4, busy-waiting like this wastes valuable CPU cycles.

Ideally, we'd like a way to *block* the consumer thread (put it to sleep) while the producer does its work, and then wake it up when the data is ready to be consumed. This can be accomplished by making use of a new kind of kernel object called a *condition variable* (CV).

A condition variable isn't actually a variable that stores a condition. Rather, it's a queue of waiting (sleeping) threads, combined with a mechanism that allows a running thread to wake up the sleeping threads at a time of its choosing. (Perhaps "wait queue" would have been a better name for these things.) The sleep and wake operations are performed in an atomic way with the help of a mutex provided by the program, plus a little help from the kernel.

The API for a condition variable typically looks something like this:

1. `create()` or `init()`. A function call or class constructor that creates a condition variable.
2. `destroy()`. A function call or destructor that destroys a condition variable.
3. `wait()`. A blocking function that puts the calling thread to sleep.
4. `notify()`. A non-blocking function that wakes up any threads that are currently asleep waiting on the condition variable.

Let's rewrite our simple producer-consumer example using a CV:

```
Queue           g_queue;
pthread_mutex_t g_mutex;
```

```
bool            g_ready = false;
pthread_cond_t  g_cv;

void* ProducerThreadCV(void*)
{
    // keep on producing forever...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // fill the queue with data
        ProduceDataInto(&g_queue);

        // notify and wake up the consumer thread
        g_ready = true;
        pthread_cond_signal(&g_cv);
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}

void* ConsumerThreadCV(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        pthread_mutex_lock(&g_mutex);
        while (!g_ready)
        {
            // go to sleep until notified... the mutex
            // will be relased for us by the kernel
            pthread_cond_wait(&g_cv, &g_mutex);

            // when it wakes up, the kernel makes sure
            // that this thread holds the mutex again
        }

        // consume the data
        ConsumeDataFrom(&g_queue);
        g_ready = false;
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}
```

消费者线程调用 pthread_cond_wait() 进入睡眠状态，直到


```
bool            g_ready = false;
pthread_cond_t  g_cv;

void* ProducerThreadCV(void*)
{
    // keep on producing forever...
    while (true)
    {
        pthread_mutex_lock(&g_mutex);

        // fill the queue with data
        ProduceDataInto(&g_queue);

        // notify and wake up the consumer thread
        g_ready = true;
        pthread_cond_signal(&g_cv);
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}

void* ConsumerThreadCV(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        pthread_mutex_lock(&g_mutex);
        while (!g_ready)
        {
            // go to sleep until notified... the mutex
            // will be relased for us by the kernel
            pthread_cond_wait(&g_cv, &g_mutex);

            // when it wakes up, the kernel makes sure
            // that this thread holds the mutex again
        }

        // consume the data
        ConsumeDataFrom(&g_queue);
        g_ready = false;
        pthread_mutex_unlock(&g_mutex);
    }
    return nullptr;
}
```

The consumer thread calls `pthread_cond_wait()` to go to sleep until

g_ready 变为 true。生产者工作一段时间，生成其数据。当数据准备就绪时，生产者将全局 g_ready 标志设置为 true ，然后通过调用 pthread_cond_signal() 唤醒处于休眠状态的消费者。然后，消费者开始消费数据。在本例中，消费者和生产者就这样无限地来回切换。

您可能注意到，消费者线程在进入检查 g_ready 标志的 while 循环之前锁定了它的互斥锁。当它等待条件变量时，它显然会在持有互斥锁的情况下进入睡眠状态！通常情况下，这是大忌：如果线程在持有锁的情况下进入睡眠状态，几乎肯定会导致死锁（参见 4.7.1 节）。然而，使用条件变量时，这不是问题。这是因为内核实际上做了一些小处理，在线程安全进入睡眠状态后解锁了互斥锁。之后，当睡眠线程被唤醒时，内核会做一些小处理，以确保锁再次被新唤醒的线程持有。

您可能注意到了另一个奇怪的现象：尽管消费者线程也使用条件变量来等待 g_ready 标志变为 true，但它仍然使用 while 循环来检查该标志的值。之所以需要这个循环，是因为线程有时会被内核虚假唤醒。因此，当 pthread_cond_wait() 调用返回时， g_ready 的值可能实际上尚未变为 true。因此，我们必须不断循环轮询，直到条件真正变为 true。

4.6.4 信号量

就像互斥量就像一个原子布尔标志一样，信号量就像一个原子计数器，其值永远不会低于零。我们可以将信号量视为一种特殊的互斥量，允许多个线程同时获取它。

信号量可用于允许一组线程共享一组有限的资源。例如，假设我们正在实现一个渲染系统，允许将文本和 2D 图像渲染到屏幕外的缓冲区，用于绘制游戏的平视显示器 (HUD) 和游戏内菜单。由于内存限制，我们进一步假设只能分配四个这样的缓冲区。信号量可用于确保在任何给定时刻最多允许四个线程渲染到这些缓冲区。

信号量的 API 通常由以下函数组成：

1. init() 初始化信号量对象，并将其计数器设置为指定的

`g_ready` becomes true. The producer works for a while producing its data. When the data is ready, the producer sets the global `g_ready` flag to `true`, and then wakes up the sleeping consumer by calling `pthread_cond_signal()`. The consumer then consumes the data. In this example, the consumer and producer ping-pong back and forth like this indefinitely.

You probably noticed that the consumer thread locks its mutex before entering the while loop that checks the `g_ready` flag. When it waits on the condition variable, it apparently goes to sleep *while holding the mutex lock!* Normally this would be a big no-no: If a thread goes to sleep while holding a lock, it will almost certainly lead to a *deadlock* situation (see Section 4.7.1). However, this is not a problem when using a condition variable. That's because the kernel actually does a little slight of hand, *unlocking the mutex* after the thread has been safely put to bed. Later, when the sleeping thread is woken back up, the kernel does some more slight of hand to ensure that the lock will once again be held by the freshly awoken thread.

You may have noticed another oddity: The consumer thread still uses a while loop to check the value of `g_ready`, despite also using a condition variable to wait for the flag to become true. The reason that this loop is necessary is that threads can sometimes be awoken spuriously by the kernel. As a result, when the call to `pthread_cond_wait()` returns, the value of `g_ready` might *not* actually be true yet. So we must keep polling in a loop until the condition really is true.

4.6.4 Semaphores

Just as a mutex acts like an atomic Boolean flag, a *semaphore* acts like an atomic counter whose value is never allowed to drop below zero. We can think of a semaphore as a special kind of mutex that allows *more than one thread* to acquire it simultaneously.

A semaphore can be used to permit a group of threads to share a limited set of resources. For example, let's suppose that we're implementing a rendering system that allows text and 2D images to be rendered to offscreen buffers, for the purposes of drawing the game's heads-up display (HUD) and in-game menus. Due to memory constraints, let's further assume that we can only afford to allocate four of these buffers. A semaphore can be used to ensure that no more than four threads are permitted to render into these buffers at any given moment.

The API of a semaphore is typically comprised of the following functions:

1. `init()`. Initializes a semaphore object, and sets its counter to a specified

初始值。

2. destroy() . 销毁信号量对象。

3. take() 或 wait() 。如果给定信号量封装的计数器值大于零，此函数将减少计数器并立即返回。如果其计数器值当前为零，此函数将阻塞（使线程进入睡眠状态），直到信号量的计数器再次升至零以上。

4. give() 、 post() 或 signal() 。将封装的计数器值加一，从而为另一个线程打开一个“槽位”，以便 take() 信号量。如果在调用 give() 时，某个线程正在等待信号量并处于休眠状态，则该线程将从 take() 调用中被唤醒。

or `wait()`.[7]

因此，要实现一个最多可同时被 N 个线程访问的资源池，我们只需创建一个信号量，并将其计数器初始化为 N 即可。线程通过调用 take() 来访问资源池，并在访问完成后调用 give() 来释放对资源池的持有。

我们称信号量在计数大于零时处于信号状态，在计数等于零时处于非信号状态。这就是为什么在某些 API 中，接收和发送信号量的函数分别被命名为 wait() 和 signal() ：如果线程调用此函数时信号量未处于信号状态，则该线程将等待信号量变为信号状态。

4.6.4.1 互斥锁与二进制信号量

初始值为 1 的信号量称为二进制信号量。有人可能会认为二进制信号量与互斥量相同。当然，这两个对象每次都只允许一个线程获取。然而，这两个同步对象并不等同，并且通常用于完全不同的目的。

互斥锁和二进制信号量之间的关键区别在于，互斥锁只能由锁定它的线程解锁。而信号量的计数器可以由一个线程递增，然后由另一个线程递减。这意味着二进制信号量可以由与“锁定”它不同的线程“解锁”。或者更确切地说，我们应该说，二进制信号量可以由与获取它的线程不同的线程发出。

互斥锁和二进制信号量之间看似微妙的差异导致这两种同步对象的用例非常不同。

7 当你阅读有关信号量的文章时，你可能会发现一些作者使用函数名 p () 和 v () 来代替 wait () 和 signal ()。这两个字母来自荷兰语中这两个操作的名称。

initial value.

2. `destroy()`. Destroys a semaphore object.
3. `take()` or `wait()`. If the counter value encapsulated by a given semaphore is greater than zero, this function decrements the counter and returns immediately. If its counter value is currently zero, this function blocks (puts the thread to sleep) until the semaphore's counter rises above zero again.
4. `give()`, `post()` or `signal()`. Increments the encapsulated counter value by one, thereby opening up a "slot" for another thread to `take()` the semaphore. If a thread is currently asleep waiting on the semaphore when `give()` is called, that thread will wake up from its call to `take()` or `wait()`.[7]

So, to implement a resource pool that can be accessed by up to *N* threads at a time, we would simply create a semaphore and initialize its counter to *N*. A thread gains access to the resource pool by calling `take()`, and releases its hold on the resource pool when it is done by calling `give()`.

We say that a semaphore is *signaled* whenever its count is greater than zero, and it is *nonsignaled* when its counter is equal to zero. This is why the functions that take and give a semaphore are named `wait()` and `signal()`, respectively, in some APIs: If the semaphore isn't signaled when a thread calls this function, the thread will *wait* for the semaphore to become signaled.

4.6.4.1 Mutex versus Binary Semaphore

A semaphore whose initial value is set to 1 is called a *binary semaphore*. One might think that a binary semaphore is identical to a mutex. Certainly both objects permit only one thread to acquire it at a time. However, these two synchronization objects are not equivalent, and are typically used for quite different purposes.

The key difference between a mutex and a binary semaphore is that a mutex can only be unlocked by the thread that locked it. A semaphore's counter, on the other hand, can be incremented by one thread and later decremented by another thread. This implies that a binary semaphore can be "unlocked" by a different thread than the one that "locked" it. Or really, we should say that a binary semaphore can be *given* by a different thread than the one that *takes* it.

This seemingly subtle difference between mutexes and binary semaphores leads to very different use cases for these two kinds of synchronization object.

[7] When you read about semaphores, you may discover some authors using the function names *p*() and *v*() instead of *wait*() and *signal*(). These letters come from the Dutch names for these two operations.

互斥锁用于使操作原子化。但二进制信号量通常用于从一个线程向另一个线程发送信号。

再次考虑我们的生产者-消费者示例，其中生产者需要在其生成的数据可供消费时通知消费者。此通知机制可以使用两个二进制信号量来实现，一个允许生产者唤醒消费者，另一个允许消费者唤醒生产者。我们可以将这些信号量视为表示两个线程共享的缓冲区中已使用和空闲元素的数量，尽管在这个简单的示例中，缓冲区只能容纳单个元素。因此，我们将这两个信号量分别称为 g_semUsed 和 g_semFree 。使用 POSIX 信号量的代码如下所示：

```
Queue g_queue;
sem_t g_semUsed; // initialized to 0
sem_t g_semFree; // initialized to 1

void* ProducerThreadSem(void*)
{
    // keep on producing forever...
    while (true)
    {
        // produce an item (can be done non-
        // atomically because it's local data)
        Item item = ProduceItem();

        // decrement the free count
        // (wait until there's room)
        sem_wait(&g_semFree);

        AddItemToQueue(&g_queue, item);

        // increment the used count
        // (notify consumer that there's data)
        sem_post(&g_semUsed);
    }
    return nullptr;
}

void* ConsumerThreadSem(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // decrement the used count
```


A mutex is used to make an operation *atomic*. But a binary semaphore is typically used to send a *signal* from one thread to another.

Consider again our producer-consumer example, in which the producer needs to notify the consumer when the data it produces is ready for consumption. This notification mechanism can be implemented using two binary semaphores, one that allows the producer to wake the consumer, and one that allows the consumer to wake the producer. We can think of these semaphores as representing the number of elements that are used and free within a buffer that's shared between the two threads, although in this simple example the buffer can only hold a single item. As such, we'll call the semaphores `g_semUsed` and `g_semFree`, respectively. Here's what the code would look like, using POSIX semaphores:

```
Queue g_queue;
sem_t g_semUsed; // initialized to 0
sem_t g_semFree; // initialized to 1

void* ProducerThreadSem(void*)
{
    // keep on producing forever...
    while (true)
    {
        // produce an item (can be done non-
        // atomically because it's local data)
        Item item = ProduceItem();

        // decrement the free count
        // (wait until there's room)
        sem_wait(&g_semFree);

        AddItemToQueue(&g_queue, item);

        // increment the used count
        // (notify consumer that there's data)
        sem_post(&g_semUsed);
    }
    return nullptr;
}

void* ConsumerThreadSem(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // decrement the used count
```

```
        // (wait for the data to be ready)
        sem_wait(&g_semUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // increment the free count
        // (notify producer that there's room)
        sem_post(&g_semFree);

        // consume the item (can be done non-
        // atomically because it's local data)
        ConsumeItem(item);
    }
    return nullptr;
}
```

4.6.4.2 实现信号量

事实证明，我们可以用互斥锁、条件变量和整数来实现信号量。从这个意义上讲，信号量比互斥锁或条件变量“更高级”。其实现如下：

```
class Semaphore
{
private:
    int             m_count;
    pthread_mutex_t m_mutex;
    pthread_cond_t  m_cv;

public:
    explicit Semaphore(int initialCount)
    {
        m_count = initialCount;
        pthread_mutex_init(&m_mutex, nullptr);
        pthread_cond_init(&m_cv, nullptr);
    }

    void Take()
    {
        pthread_mutex_lock(&m_mutex);
        // put the thread to sleep as long as
        // the count is zero
        while (m_count == 0)
            pthread_cond_wait(&m_cv, &m_mutex);
```



```
        // (wait for the data to be ready)
        sem_wait(&g_semUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // increment the free count
        // (notify producer that there's room)
        sem_post(&g_semFree);

        // consume the item (can be done non-
        // atomically because it's local data)
        ConsumeItem(item);
    }
    return nullptr;
}
```

4.6.4.2 Implementing a Semaphore

It turns out that one can implement a semaphore in terms of a mutex, a condition variable and an integer. In that sense, a semaphore is a “higher-level” construct than either a mutex or a condition variable. Here’s what the implementation looks like:

```
class Semaphore
{
private:
    int             m_count;
    pthread_mutex_t m_mutex;
    pthread_cond_t  m_cv;

public:
    explicit Semaphore(int initialCount)
    {
        m_count = initialCount;
        pthread_mutex_init(&m_mutex, nullptr);
        pthread_cond_init(&m_cv, nullptr);
    }

    void Take()
    {
        pthread_mutex_lock(&m_mutex);
        // put the thread to sleep as long as
        // the count is zero
        while (m_count == 0)
            pthread_cond_wait(&m_cv, &m_mutex);
```

```
    --m_count;
    pthread_mutex_unlock(&m_mutex);
}

void Give()
{
    pthread_mutex_lock(&m_mutex);
    ++m_count;
    // if the count was zero before the
    // increment, wake up a waiting thread
    if (m_count == 1)
        pthread_cond_signal(&m_cv);
    pthread_mutex_unlock(&m_mutex);
}

// aliases for other commonly-used function names
void Wait()   { Take(); }
void Post()   { Give(); }
void Signal() { Give(); }
void Down()   { Take(); }
void Up()     { Give(); }
void P()      { Take(); } // Dutch "proberen" = "test"
void V()      { Give(); } // Dutch "verhogen" =
                          // "increment"
};
```

4.6.5 Windows 事件

Windows 提供了一种称为事件对象的机制，其功能类似于条件变量，但使用起来要简单得多。一旦创建了事件对象，线程就可以通过调用 WaitForSingleObject() 进入睡眠状态，而该线程可以通过调用 SetEvent() 被另一个线程唤醒。使用事件对象重写我们的生产者-消费者示例，可以得到以下非常简单的实现：

```
#include <windows.h>

Queue g_queue;
Handle g_hUsed; // initialized to false (nonsignaled)
Handle g_hFree; // initialized to true (signaled)

void* ProducerThreadEv(void*)
{
    // keep on producing forever...
    while (true)
    {
        // produce an item (can be done non-
```



```
    --m_count;
    pthread_mutex_unlock(&m_mutex);
}

void Give()
{
    pthread_mutex_lock(&m_mutex);
    ++m_count;
    // if the count was zero before the
    // increment, wake up a waiting thread
    if (m_count == 1)
        pthread_cond_signal(&m_cv);
    pthread_mutex_unlock(&m_mutex);
}

// aliases for other commonly-used function names
void Wait()   { Take(); }
void Post()   { Give(); }
void Signal() { Give(); }
void Down()   { Take(); }
void Up()     { Give(); }
void P()      { Take(); } // Dutch "proberen" = "test"
void V()      { Give(); } // Dutch "verhogen" =
                          // "increment"
};
```

4.6.5 Windows Events

Windows provides a mechanism called an *event object* that is similar in function to a condition variable, but much simpler to use. Once an event object has been created, a thread can go to sleep by calling `WaitForSingleObject()`, and that thread can be awoken by another thread by calling `SetEvent()`. Rewriting our producer-consumer example using event objects yields the following very simple implementation:

```
#include <windows.h>

Queue g_queue;
Handle g_hUsed; // initialized to false (nonsignaled)
Handle g_hFree; // initialized to true (signaled)

void* ProducerThreadEv(void*)
{
    // keep on producing forever...
    while (true)
    {
        // produce an item (can be done non-
```

```
        // atomically because it's local data)
        Item item = ProduceItem();

        // wait until there's room
        WaitForSingleObject(&g_hFree);

        AddItemToQueue(&g_queue, item);

        // notify consumer that there's data
        SetEvent(&g_hUsed);
    }
    return nullptr;
}

void* ConsumerThreadEv(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        WaitForSingleObject(&g_hUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // notify producer that there's room
        SetEvent(&g_hFree);

        // consume the item (can be done non-
        // atomically because it's local data)
        ConsumeItem(item);
    }
    return nullptr;
}

void MainThread()
{
    // create event in the nonsignalled state
    g_hUsed = CreateEvent(nullptr, false,
                          false, nullptr);
    g_hFree = CreateEvent(nullptr, false,
                          true, nullptr);

    // spawn our threads
    CreateThread(nullptr, 0x2000, ConsumerThreadEv,
                 0, 0, nullptr);
    CreateThread(nullptr, 0x2000, ProducerThreadEv,
                 0, 0, nullptr);
```



```
        // atomically because it's local data)
        Item item = ProduceItem();

        // wait until there's room
        WaitForSingleObject(&g_hFree);

        AddItemToQueue(&g_queue, item);

        // notify consumer that there's data
        SetEvent(&g_hUsed);
    }
    return nullptr;
}

void* ConsumerThreadEv(void*)
{
    // keep on consuming forever...
    while (true)
    {
        // wait for the data to be ready
        WaitForSingleObject(&g_hUsed);

        Item item = RemoveItemFromQueue(&g_queue);

        // notify producer that there's room
        SetEvent(&g_hFree);

        // consume the item (can be done non-
        // atomically because it's local data)
        ConsumeItem(item);
    }
    return nullptr;
}

void MainThread()
{
    // create event in the nonsignalled state
    g_hUsed = CreateEvent(nullptr, false,
                          false, nullptr);
    g_hFree = CreateEvent(nullptr, false,
                          true, nullptr);

    // spawn our threads
    CreateThread(nullptr, 0x2000, ConsumerThreadEv,
                 0, 0, nullptr);
    CreateThread(nullptr, 0x2000, ProducerThreadEv,
                 0, 0, nullptr);
```

```
    // ...
}
```

4.7 基于锁的并发问题

在 4.5.3.2 节中，我们了解到数据竞争可能导致并发系统中程序行为异常。我们发现，解决这个问题的方法是使对共享数据对象的操作具有原子性。实现原子性的一种方法是将这些操作包装在锁中，锁通常使用操作系统提供的线程同步原语（例如互斥锁）来实现。

然而，原子性只是并发问题的一部分。即使所有共享数据操作都已通过锁精心保护，并发系统仍可能存在其他问题。在接下来的章节中，我们将简要探讨其中最常见的问题。

4.7.1 死锁

死锁是指系统中所有线程都无法继续运行，从而导致程序挂起的情况。发生死锁时，所有线程都处于“阻塞”状态，等待某个资源可用。但由于没有线程处于“可运行”状态，这些资源都无法使用，因此整个程序都会挂起。

要发生死锁，我们至少需要两个线程和两个资源。
例如，线程 1 持有资源 A，但正在等待资源 B；同时，线程 2 持有资源 B，但正在等待资源 A。以下代码片段说明了这种情况：

```
void Thread1()
{
    g_mutexA.lock(); // holds lock for Resource A
    g_mutexB.lock(); // sleeps waiting for Resource B
    // ...
}

void Thread2()
{
    g_mutexB.lock(); // holds lock for Resource B
    g_mutexA.lock(); // sleeps waiting for Resource A
    // ...
}
```



```
    // ...
}
```

4.7 Problems with Lock-Based Concurrency

In Section 4.5.3.2, we learned that *data races* can lead to incorrect program behavior in a concurrent system. We saw that the solution to this problem is to make operations on *shared* data objects *atomic*. One way to achieve atomicity is to wrap these operations in *locks*, which are often implemented using operating-system provided thread synchronization primitives such as mutexes.

However, atomicity is only part of the concurrency story. There are other problems that can plague a concurrent system, even when all shared-data operations have been carefully protected by locks. In the following sections, we'll briefly explore the most common of these problems.

4.7.1 Deadlock

Deadlock is a situation in which no thread in the system can make progress, resulting in a hang. When a deadlock occurs, all threads are in their Blocked states, waiting on some resource to become available. But because no threads are Runnable, none of those resources can ever become available, so the entire program hangs.

For a deadlock to occur, we need at least two threads and two resources. For example, Thread 1 holds Resource A but is waiting for Resource B; and at the same time, Thread 2 holds Resource B but is waiting for Resource A. This situation is illustrated by the following code snippet:

```
void Thread1()
{
    g_mutexA.lock(); // holds lock for Resource A
    g_mutexB.lock(); // sleeps waiting for Resource B
    // ...
}

void Thread2()
{
    g_mutexB.lock(); // holds lock for Resource B
    g_mutexA.lock(); // sleeps waiting for Resource A
    // ...
}
```

图 4.34。左图：深色箭头表示线程当前占用的资源。浅色虚线箭头表示正在等待资源可用的线程。右图：我们可以排除资源，并简单地绘制线程之间的依赖关系（浅色虚线箭头）。此类线程依赖关系图中的循环表示发生死锁。

当然，也可能出现其他更复杂的死锁类型，涉及更多线程和更多资源。但定义任何死锁情况的关键因素是线程与其资源之间的循环依赖关系。

为了分析系统发生死锁的可能性，我们可以绘制一个线程、资源以及它们之间依赖关系的图表，如图 4.34 所示。在此图中，我们使用正方形表示线程，用圆圈表示资源（或者更准确地说，是保护它们的互斥锁）。实线箭头将资源连接到当前持有其锁的线程。虚线箭头将线程连接到它们正在等待的资源。实际上，如果愿意，我们可以为了简单起见消除图中的资源节点，只留下连接正在等待其他线程的线程的虚线。如果这样的依赖关系图包含循环，则会发生死锁。

实际上，依赖图中的循环并不足以产生死锁。严格来说，死锁的产生需要满足四个必要充分条件，即科夫曼条件：

1. 互斥。可以通过互斥锁授予单个线程对单个资源的独占访问权限。

2. 持有并等待。当线程进入休眠状态并等待另一个锁时，它必须持有一个锁。

3. 禁止锁抢占。任何人（甚至内核）都不允许强制

Figure 4.34. Left: Dark arrows show resources currently held by threads. Light dashed arrows show threads waiting on resources to become available. Right: We can eliminate the resources and simply draw the dependencies (light dashed arrows) between threads. A cycle in such a thread dependency graph indicates a deadlock.

Other more complex kinds of deadlock can of course occur as well, involving more threads and more resources. But the key factor that defines any deadlock situation is a *circular* dependency between threads and their resources.

To analyze a system for the possibility of deadlock, we can draw a graph of our threads, our resources, and the dependencies between them, as shown in Figure 4.34. In this graph, we've used squares to represent threads and circles to represent resources (or more precisely, the mutex locks that protect them). Solid arrows connect resources to the threads that currently hold locks on them. Dashed arrows connect threads to the resources they are waiting for. We can actually eliminate the resource nodes in the graph for simplicity if we like, leaving only the dashed lines connecting threads that are waiting on other threads. If such a dependency graph ever contains a *cycle*, we have a deadlock.

Actually a cycle in the dependency graph isn't quite enough to produce a deadlock. Strictly speaking, there are four necessary and sufficient conditions for deadlock, known as the *Coffman conditions:*

1. *Mutual exclusion*. A single thread can be granted exclusive access to a single resource via a mutex lock.

2. *Hold and wait*. A thread must be holding one lock when it goes to sleep waiting on another lock.

3. *No lock preemption*. No one (not even the kernel) is allowed to forcibly

打破休眠线程所持有的锁。

4. 循环等待。线程依赖图中一定存在一个循环。

避免死锁归根结底在于阻止一个或多个科夫曼条件成立。由于违反条件1和条件3属于“作弊”，因此解决方案通常侧重于避免条件2和

#4.
通过减少锁的数量可以避免持有和等待的情况。在我们这个简单的例子中，如果资源 A 和资源 B 都由一个锁 L 保护，那么就不会发生死锁。要么线程 1 获得锁，并在线程 2 等待时获得对两个资源的独占访问权；要么线程 2 获得锁，而在线程 1 等待时。

通过对系统中所有锁的获取施加全局顺序，可以避免循环等待的情况。在我们简单的双线程示例中，如果我们确保资源 A 的锁始终先于资源 B 的锁获取，就可以避免死锁。这样做是可行的，因为总有一个线程会在尝试获取任何其他锁之前获取资源 A 的锁。这实际上会将所有其他竞争线程置于睡眠状态，从而确保获取资源 B 锁的尝试始终成功。

4.7.2 活锁

解决死锁问题的另一种方法是让线程在不进入休眠状态的情况下尝试获取锁（使用 pthread_mutex_trylock() 之类的函数）。如果无法获取锁，线程会休眠一小段时间，然后重试获取锁。

当线程使用显式策略（例如定时重试）来避免或解决死锁时，可能会出现一个新问题：线程最终可能会将所有时间都花在尝试解决死锁上，而不是执行任何实际工作。这被称为活锁。

举一个简单的活锁例子，再次思考一下之前的例子：两个线程 1 和 2 争夺两个资源 A 和 B。每当一个线程无法获取锁时，它就会释放已经持有的所有锁，并等待一个固定的超时时间，然后再尝试获取。如果两个线程使用相同的超时时间，我们就会陷入一种情况：同样的劣化情况会不断重复出现。我们的线程会永远“卡住”，试图解决冲突，而没有一个线程有机会完成其真正的工作。活锁就像国际象棋中的僵局。

活锁可以通过使用非对称死锁解决算法来避免。例如，我们可以确保只有一个线程，要么选择

break a lock held by a sleeping thread.

4. *Circular wait*. There must exist a cycle in the thread dependency graph.

Avoiding deadlock always boils down to preventing one or more of the Coffman conditions from holding true. Since violating conditions #1 and #3 would be “cheating,” solutions usually focus on avoiding conditions #2 and #4.

Hold and wait can be avoided by reducing the number of locks. In our simple example, if Resource A and Resource B were both protected by a single lock L, then deadlock could not occur. Either Thread 1 would obtain the lock, and gain exclusive access to both resources while Thread 2 waits, or Thread 2 would obtain the lock while Thread 1 waits.

The circular wait condition can be avoided by imposing a global order to all lock-taking in the system. In our simple two-threaded example, if we were to ensure that Resource A’s lock is always taken before Resource B’s lock, deadlock would be avoided. This would work because one thread will always obtain a lock on Resource A before attempting to take any other locks. This effectively puts all other contending threads to sleep, thereby ensuring that the attempt to take a lock on Resource B will always succeed.

4.7.2 Livelock

Another solution to the deadlock problem is for threads to *try* to take locks without going to sleep (using a function such as `pthread_mutex_trylock()`). If a lock cannot be obtained, the thread sleeps for some short period of time and then *retries* the lock.

When threads use an explicit strategy, such as timed retries, in order to avoid or resolve deadlocks, a new problem can arise: The threads can end up spending all of their time just trying to resolve the deadlock, rather than performing any real work. This is known as *livelock*.

As a simple example of livelock, consider again our example of two threads 1 and 2 contending over two resources A and B. Whenever a thread is unable to obtain a lock, it releases any locks it already holds and waits for a fixed timeout before trying again. If both threads use the same timeout, we can get into a situation in which the same degenerate situation simply repeats over and over. Our threads become “stuck” forever trying to resolve the conflict, and neither one ever gets a chance to do its real job. Livelock is akin to a stalemate in chess.

Livelock can be avoided by using an asymmetric deadlock resolution algoirthm. For example, we could ensure that only one thread, either chosen at

随机或基于优先级，在检测到死锁时采取行动解决死锁。

4.7.3 饥饿

饥饿是指一个或多个线程无法获得 CPU 执行时间的情况。当一个或多个高优先级线程无法放弃 CPU 控制权，从而导致低优先级线程无法运行，就会发生饥饿。活锁是另一种饥饿情况，在这种情况下，死锁解决算法会有效地剥夺所有线程执行“实际”工作的能力。

通常情况下，通过确保高优先级线程不会运行太长时间，可以避免基于优先级的资源匮乏。理想情况下，多线程程序应该由一个低优先级线程池组成，这些线程通常会公平地共享系统的 CPU 资源。偶尔，一个高优先级线程会运行起来，快速处理完自己的任务，然后结束运行，将 CPU 资源归还给低优先级线程。

4.7.4 优先级反转

互斥锁可能导致一种称为优先级反转的情况，即低优先级线程会表现得好像自己是系统中最高优先级的线程。假设有两个线程 L 和 H，它们的优先级分别为低和高。线程 L 获得一个互斥锁，然后被 H 抢占。如果 H 尝试获得同一个锁，H 将被置于睡眠状态，因为 L 已经持有该锁。这允许 L 运行，即使它的优先级低于 H——这违反了“低优先级线程不应在高优先级线程可运行时运行”的原则。

如果中等优先级的线程 M 在线程 L 持有 H 所需的锁时抢占了它，也可能发生优先级反转。在这种情况下，当 M 运行时，L 会进入休眠状态，导致其无法释放锁。因此，当 H 运行时，它无法获取锁；它会进入休眠状态，此时 M 的优先级实际上已与线程 H 的优先级反转。

优先级反转的后果可能微不足道。例如，如果低优先级线程立即放弃锁，优先级反转的持续时间可能很短，可能不会被察觉，或者只产生轻微的负面影响。然而，在极端情况下，优先级反转可能会导致死锁或其他类型的系统故障。例如，优先级反转可能导致高优先级线程错过关键的截止时间。

优先级反转问题的解决方案包括：

random or based on priority, ever takes action to resolve a deadlock when one is detected.

4.7.3 Starvation

Starvation is defined as any situation in which one or more threads fail to receive any execution time on the CPU. Starvation can happen when one or more higher-priority threads fail to relinquish control of the CPU, thereby preventing lower-priority threads from running. Livelock is another kind of starvation, in which a deadlock-resolution algorithm effectively starves all threads of their ability to do “real” work.

Priority-based starvation is normally avoided by ensuring that high-priority threads never run for very long. Ideally a multithreaded program would consist of a pool of lower-priority threads that normally share the system’s CPU resources fairly. Once in a while, a higher-priority thread runs, takes care of its business quickly, and then ends, returning the CPU resources to the lower-priority threads.

4.7.4 Priority Inversion

Mutex locks can lead to a situation known as *priority inversion*, in which a low-priority thread acts as though it is the highest-priority thread in the system. Consider two threads, L and H, with a low and high priority, respectively. Thread L takes a mutex lock and then is preempted by H. If H attempts to take this same lock, then H will be put to sleep because L already holds the lock. This permits L to run even though it is lower priority than H—in violation of the principle that lower-priority threads should not run while a higher-priority thread is runnable.

Priority inversion can also occur if a medium-priority thread M preempts thread L while it is holding a lock needed by H. In this case, L goes to sleep while M runs, preventing it from releasing the lock. When H runs, it is therefore unable to obtain the lock; it goes to sleep, and now M’s priority has effectively been inverted with that of thread H.

The consequences of priority inversion may be neglible. For example, if the lower-priority thread immediately relinquishes the lock, the duration of the priority inversion may be short and it may go unnoticed or produce only minor ill effects. However, in extreme cases, priority inversion can lead to deadlock or other kinds of system failure. For example, a priority inversion can cause a high-priority thread to miss a critical deadline.

Solutions to the priority inversion problem include:

- 避免低优先级和高优先级线程都占用的锁。
  （这个解决方案往往不可行。）
- 为互斥锁本身分配非常高的优先级。任何获取该互斥锁的线程的优先级都会暂时提升到与该互斥锁相同的水平，从而确保该线程在持有锁时不会被抢占。
- 随机优先级提升。在这种方法中，主动持有锁的线程的优先级会被随机提升，直到它们退出临界区。Windows 调度模型中就使用了这种方法。

4.7.5 哲学家就餐

著名的哲学家就餐问题很好地诠释了死锁、活锁和饥饿问题。它描述了这样一种情况：五位哲学家围坐在一张圆桌旁，每人面前都放着一盘意大利面。每位哲学家之间都放着一根筷子。这些哲学家希望在思考（他们不用筷子也可以）和进食（哲学家需要两根筷子才能完成）之间交替进行。问题在于为这些哲学家定义一种行为模式，确保他们能够在思考和进食之间交替，而不会遇到死锁、活锁或饥饿问题。（显然，哲学家代表线程，筷子代表互斥锁。）

您可以在网上找到关于这个众所周知的问题的资料，所以我们就不在这里花太多篇幅来讨论了。不过，考虑一下这个问题的一些最常见的解决方案将会很有启发：

- 全局顺序。如果哲学家总是先拿起左边的筷子（或总是先拿起右边的筷子），就会出现依赖循环。解决这个问题的一个方法是给每根筷子分配一个唯一的全局索引。每当哲学家想要吃饭时，他总是先拿起索引最小的筷子。这样可以避免依赖循环，从而避免死锁。

- 中央仲裁员。在这个解决方案中，中央仲裁员（“服务员”）要么授予哲学家两根筷子，要么不授予。这避免了“持有并等待”的问题，因为它确保了任何哲学家都不会陷入只持有一根筷子的境地，从而避免了死锁。

- Chandra-Misra 方案。在这个方案中，筷子会被标记为脏的或干净的，哲学家们会互相发送请求筷子的信息。您可以在线搜索“chandymisra”了解更多关于这个方案的信息。

- Avoiding locks that can be taken by both low- and high-priority threads. (This solution is often not feasible.)
- Assigning a very high priority to the mutex itself. Any thread that takes the mutex has its priority temporarily raised to that of the mutex, thus ensuring it cannot be preempted while holding the lock.
- Random priority boosting. In this approach, threads that are actively holding locks are randomly boosted in priority until they exit their critical sections. This approach is used in the Windows scheduling model.

4.7.5 The Dining Philosophers

The famous *dining philosophers* problem is a great illustration of the problems of deadlock, livelock and starvation. It describes a situation in which five philosophers sit around a circular table, each with a plate of spaghetti in front of her. Between each philosopher sits a single chopstick. The philosophers wish to alternate between thinking (which they can do without any chopsticks), and eating (a task for which a philosopher requires two chopsticks). The problem is to define a pattern of behavior for the philosophers that ensures they can all alternate between thinking and eating without experiencing deadlock, livelock or starvation. (Obviously the philosophers represent threads, and the chopsticks represent mutex locks.)

You can read about this well-known problem online, so we won't devote a lot of space to discussing it here. However, it will be instructive to consider a few of the most common solutions to the problem:

- *Global order*. A dependency cycle can occur if a philosopher were always to pick up the chopstick to her left first (or always the one to her right). One solution to this problem is to assign each chopstick a unique global index. Whenever a philosopher wishes to eat, he or she always picks up the chopstick with the lowest index first. This prevents the dependency cycle, and hence avoids deadlock.

- *Central arbitor*. In this solution, a central arbitor (the "waiter") either grants a philosopher two chopsticks or none. This avoids the hold and wait problem by ensuring that no philosopher ever gets into a situation in which he or she holds only one chopstick, thereby preventing deadlock.

- *Chandra-Misra*. In this solution, chopsticks are marked either dirty or clean, and the philosophers send each other messages requesting chopsticks. You can read more about this solution by searching for "chandymisra" online.

• N − 1 位哲学家。对于一张有 N 位哲学家和 N 根筷子的桌子，可以使用一个整数信号量将允许在任意时刻拿起筷子的哲学家数量限制为 N − 1 位。这解决了死锁和活锁问题，因为即使在最差的情况下，也至少会有一位哲学家成功获得两根筷子。然而，这会导致其中一位哲学家挨饿，除非引入额外的“公平”标准。

4.8 并发的一些经验法则

上一节中我们描述的哲学家就餐问题的解决方案，暗示了一些通用的原则和经验法则，这些原则和法则几乎可以应用于任何并发编程问题。让我们简要地看一下其中的几个。

4.8.1 全局排序规则

在并发程序中，事件发生的顺序并不像单线程程序那样由程序中指令的顺序决定。如果并发系统中需要排序，则该排序必须在所有线程上全局强制执行。

这就是为什么双向链表不是并发数据结构的原因之一。双向链表的设计初衷是为了在链表的任何位置提供快速（O(N)）的元素插入和删除。这种设计基于一个假设：程序顺序等同于数据顺序——即对列表执行有序的操作序列时，结果列表将包含相应有序的元素。例如，假设我们有一个包含有序元素 { A, B, C } 的链表。如果我们执行以下两个操作：

1. 在 C 之前插入 D，
2. 在 C 之前插入 E，

假设结果列表将包含有序元素 { A, B, D, E, C }.
在单线程程序中，这个假设成立。但在多线程系统中，程序顺序不再决定数据顺序。如果一个线程在 C 之前插入 D，而另一个线程在 C 之前插入 E，就会出现竞争条件，可能导致以下任一结果：

- $N-1$ *philosophers*. For a table with N philosophers and N chopsticks, an integer semaphore can be used to limit the number of philosophers who are permitted to pick up chopsticks at any given time to $N-1$. This solves the deadlock and livelock problems, because even in degenerate situations, at least one philosopher will succeed in obtaining two chopsticks. It does, however, permit one of the philosophers to starve, unless an additional “fairness” criterion is introduced.

4.8 Some Rules of Thumb for Concurrency

The solutions to the dining philosophers problem we described in the preceding section hint at some general principles and rules of thumb which we can apply to virtually any concurrent programming problem. Let’s take a brief look at a few of them.

4.8.1 Global Ordering Rules

In a concurrent program, the *order* in which events occur is not dictated by the order of the instructions in the program, as it is in a single-threaded program. If ordering is required in a concurrent system, that ordering must be imposed *globally*, across all threads.

This is one reason why a doubly linked list is not a concurrent data structure. A doubly linked list is designed the way it is in order to provide fast ($O(N)$) insertion and deletion of elements at any location within the list. Underlying this design is an assumption that *program order* is equivalent to *data order*—that when an ordered sequence of operations is performed on a list, the resulting list will contain correspondingly ordered elements. For example, let’s say we are given a linked list that contains the ordered elements { A, B, C }. If we execute the following two operations:

1. insert D before C,
2. insert E before C,

the assumption is that the resulting list will contain the ordered elements { A, B, D, E, C }.

In a single-threaded program, this assumption holds true. But in a multi-threaded system, program order no longer dictates data order. If one thread performs the insertion of D before C, and another thread performs the insertion of E before C, we have a race condition that could lead to any of the following results:

- { A, B, D, E, C },
- { A, B, E, D, C }, or
- { A，B，损坏的数据}。

如果数据结构的操作未使用关键代码段进行妥善保护，则可能会出现损坏的列表。例如，D 和 E 的“next”指针最终可能都指向 C。

全局排序规则是解决此问题的唯一可行方案。我们首先需要问自己，为什么列表中元素的顺序很重要，以及它是否真的重要。如果顺序不重要，我们可以使用单链表并始终追加元素——这个操作可以通过锁或无锁方式可靠地实现。如果需要全局排序，我们需要确定一个稳定且确定性的排序标准，该标准不依赖于程序事件发生的顺序。例如，我们可以按字母顺序、优先级或其他一些有用的标准对列表进行排序。这些问题的答案反过来又决定了我们要使用什么样的数据结构。尝试以并发方式使用双向链表（即，为多个线程提供对列表的可变访问权限）就像试图将方形钉子塞进圆孔中。

4.8.2 基于交易的算法

在哲学家就餐问题的中央仲裁方案中，仲裁者或“服务员”会成对地分发筷子：哲学家要么获得他所需的全部资源（两根筷子），要么什么都得不到。这被称为一次交易。

事务可以更精确地定义为一组不可分割的资源和/或操作。并发系统中的线程将事务请求提交给某种中央仲裁器。事务要么完全成功，要么完全失败（因为请求到达时其他线程的事务正在被处理）。如果事务失败，其线程会不断重新提交事务请求，直到成功（两次重试之间可能需要等待一小段时间）。

基于事务的算法在并发和分布式系统编程中很常见。正如我们将在 4.9 节中看到的，事务的概念是大多数无锁数据结构和算法的基础。

4.8.3 最小化争用

最高效的并发系统应该是所有线程都无需等待锁即可运行。但这种理想状态永远无法完全实现。

- { A, B, D, E, C },
- { A, B, E, D, C }, or
- { A, B, *corrupted data* }.

A corrupted list can occur if the data structure's operations aren't properly protected with critical sections. For example, both D's and E's "next" pointers might end up pointing to C.

A global ordering rule is the only viable solution to this problem. We first need to ask ourselves why the order of the elements in the list matters, and if it even matters at all. If order is unimportant, we can use a singly-linked list and always append the elements—an operation that can be implemented reliably either with locks or in a lock-free manner. And if a global ordering *is* required, we need to identify a stable and deterministic ordering criterion that does not depend on the order in which program events happen to occur. For example, we might sort the list alphabetically, or by priority, or some other useful criterion. The answers to these questions in turn dictate what kind of data structure we'll want to use. Attempting to use a doubly-linked list in a concurrent way (i.e., giving multiple threads mutable access to the list) is like trying to fit a square peg into a round hole.

4.8.2 Transaction-Based Algorithms

In the central arbiter solution to the dining philosophers problem, the arbiter or "waiter" hands out chopsticks in pairs: Either a philosopher receives all of the resources he needs (two chopsticks), or he receives none of them. This is known as a *transaction*.

A transaction can be more precisely defined as an indivisible bundle of resources and/or operations. Threads in a concurrent system submit transaction requests to a central arbiter of some kind. A transaction either succeeds in its entirety, or it fails in its entirety (because some other thread's transaction is actively being processed when the request arrives). If the transaction fails, its thread keeps resubmitting the transaction request until it succeeds (possibly waiting for a short time between retries).

Transaction-based algorithms are common in concurrent and distributed systems programming. And as we'll see in Section 4.9, the concept of a transaction underlies most *lock-free* data structures and algorithms.

4.8.3 Minimizing Contention

The most efficient concurrent system would be one in which all threads run without ever having to wait for a lock. This ideal can never be fully achieved,

当然，但并发系统程序员确实试图尽量减少线程之间的争用量。

举个例子，假设有一组线程正在生成数据并将其存储到一个中央存储库。每当其中一个线程尝试将其数据存储到存储库时，它都会与所有其他线程争夺这个共享资源。一个有时可行的简单解决方案是为每个线程提供自己的私有存储库。这样，这些线程就可以彼此独立地生成数据，而不会发生争用。当所有线程都完成输出后，中央线程可以整理结果。

在哲学家就餐问题中，与此方法类似的是，一开始就给每个哲学家两根筷子。这样做当然会消除问题中的所有并发性——没有任何共享资源，就没有并发性。在现实世界的并发系统中，我们无法消除所有资源共享，但我们当然可以寻找方法来最小化资源共享，从而最大限度地减少锁争用。

4.8.4 线程安全

一般而言，当一个类或函数式 API 的函数可以被多线程进程中的任何线程安全地调用时，我们就称其为线程安全的。对于任何一个函数，线程安全通常是通过进入函数体顶部的临界区、执行一些工作，然后在返回之前离开临界区来实现的。所有函数都是线程安全的类有时被称为监视器。（“监视器”一词也用于指在内部使用条件变量以允许客户端在等待其受保护的资源可用时处于休眠状态的类。）线程安全是类或接口提供的一项便捷功能。但它也会带来有时不必要的开销。例如，如果接口在单线程程序中使用，或者在多线程程序中仅由一个线程使用，那么这种开销就是浪费。

当接口函数需要被重入调用时，线程安全也可能成为一个问题。例如，如果一个类提供了两个线程安全函数 A() 和 B()，那么这些函数不能互相调用，因为每个函数都独立地进入和离开临界区。解决这个问题的一个方法是使用可重入锁（参见第 4.9.7.3 节）。另一种方法是在接口中实现函数的“不安全”版本，然后将每个函数“包装”在线程安全的变体中，该变体只需进入临界区，调用“不安全”函数，然后离开临界区。这样，我们可以在系统内部调用“不安全”函数，但

of course, but concurrent systems programmers do attempt to minimize the amount of *contention* between threads.

As an example, consider a group of threads that are producing data and storing it into a central repository. Every time one of these threads attempts to store its data in the repository, it contends with all of the other threads for this shared resource. A simple solution that can sometimes work is to give each thread its own private repository. The threads can now produce data independently of one another, with no contention. When all of the threads are done producing their output, a central thread can collate the results.

The analog to this approach in the case of the dining philosophers problem would be to give each philosopher two chopsticks from the outset. Doing so would of course remove all concurrency from the problem—without any shared resources, there is no concurrency. In real-world concurrent systems we can't remove *all* sharing of resources, but we can certainly look for ways to *minimize* resource sharing in order to minimize lock contention.

4.8.4 Thread Safety

Generally speaking, a class or functional API is called *thread-safe* when its functions can be safely called by any thread in a multithreaded process. For any one function, thread safety is typically achieved by entering a critical section at the top of the function's body, performing some work, and then leaving the critical section just prior to returning. A class whose functions are all thread-safe is sometimes called a *monitor*. (The term "monitor" is also used to refer to a class that uses condition variables internally to permit clients to sleep while waiting for its protected resources to become available.)

Thread-safety is a convenient feature for a class or interface to offer. But it also imposes overhead that is sometimes unnecessary. For example, this overhead would be wasteful if the interface is being used in a single-threaded program, or if it's being used exclusively by one thread in a multithreaded program.

Thread-safety can also become a problem when an interface function needs to be called reentrantly. For example, if a class provides two thread-safe functions `A()` and `B()`, then these functions cannot call one another because each one independently enters and leaves the critical section. One solution to this problem is to use reentrant locks (see Section 4.9.7.3). Another is to implement "unsafe" versions of the functions in an interface, and then "wrap" each of them in thread-safe variant that simply enters a critical section, calls the "unsafe" function, and then leaves the critical section. That way, we can call the "unsafe" functions internally to the system, but the external interface of the

系统保持线程安全。

在我看来，试图通过简单地创建 100% 线程安全的类和 API 来处理并发编程是一个糟糕的主意。这样做会导致接口变得不必要的繁重，并鼓励程序员忽略他们正在并发环境中工作的事实。相反，我们应该认识到并接受软件中并发的存在，并致力于设计能够明确处理这种并发的数据结构和算法。目标应该是创建一个能够最大限度地减少线程间争用和依赖，并最大限度地减少锁使用的软件系统。实现一个完全无锁的系统需要大量的工作，而且很难做到。（参见 4.9 节。）但是，努力朝着无锁的方向发展（即最大限度地减少锁的使用）远比过度使用锁来徒劳地创建“无懈可击”的接口，让程序员完全忽略并发性，要好得多。

4.9 无锁并发

到目前为止，我们针对并发系统中竞争条件问题的所有解决方案都围绕着使用互斥锁来实现关键操作的原子化，并可能利用条件变量和内核将线程置于休眠状态的功能，以避免它们在忙等待循环中浪费宝贵的 CPU 周期。在那次讨论中，我们提到了另一种可能更高效的避免竞争条件的方法。这种方法被称为无锁并发。

与普遍的看法相反，“无锁”一词实际上并不是指消除互斥锁，尽管互斥锁确实是该方法的一个组成部分。事实上，“无锁”指的是防止线程在等待资源可用时进入睡眠状态的做法。换句话说，在无锁编程中，我们绝不允许线程阻塞。因此，“无阻塞”一词或许更具描述性。

无锁编程实际上只是一系列非阻塞并发编程技术中的一种。当一个线程被阻塞时，它将停止执行。所有这些技术的目标都是为系统中线程以及整个系统的执行提供保证。我们可以将这些技术分为以下几类，按其提供的执行保证的强度依次递增：

- 阻塞。阻塞算法是指线程在等待共享资源可用时进入睡眠状态的算法。阻塞

system remains thread-safe.

In my view, it's a bad idea to attempt to handle concurrent programming by simply creating classes and APIs that are 100% thread-safe. Doing so leads to interfaces that are unnecessarily heavy-weight, and it encourages programmers to ignore the fact that they are working in a concurrent environment. Instead, we should recognize and embrace the presence of concurrency in our software, and aim to design data structures and algorithms that address this concurrency explicitly. The goal should be to produce a software system that minimizes contention and dependencies between threads, and minimizes the use of locks. Achieving a completely lock-free system is a lot of work and is tough to get right. (See Section 4.9.) But striving in the *direction* of lock-freedom (i.e., minimizing the use of locks) is a far better strategy than over-using locks in a futile attempt to create "water-tight" interfaces that allow programmers to be oblivious of concurrency.

4.9 Lock-Free Concurrency

Thus far, all of our solutions to the problem of race conditions in a concurrent system have revolved around using *mutex locks* to make critical operations atomic, and possibly leveraging *condition variables* and the kernel's ability to *put threads to sleep* to avoid them wasting valuable CPU cycles in a busy-wait loop. During that discussion, we mentioned that there's another way to avoid race conditions that can potentially be more efficient. That approach is known as *lock-free* concurrency.

Contrary to popular belief, the term "lock-free" doesn't actually refer to the elimination of mutex locks, although that is certainly a component of the approach. In fact, "lock-free" refers to the practice of preventing threads from going to sleep while waiting on a resource to become available. In other words, in lock-free programming we never allow a thread to *block*. So perhaps the term "blocking-free" would have been more descriptive.

Lock-free programming is actually just one of a collection of *non-blocking* concurrent programming techniques. When a thread is blocked, it ceases to make progress. The goal of all of these techniques is to provide guarantees about the *progress* that can be made by the threads in the system, and by the system as a whole. We can organize these techniques into the following categories, listed in order of increasing strength of the progress guarantees they provide:

- *Blocking*. A blocking algorithm is one in which a thread can be put to sleep while waiting for shared resources to become available. A block-

ing算法容易出现死锁、活锁、饥饿和优先级反转。

• 无阻塞。如果我们能够保证当系统中所有其他线程突然挂起时，单个线程始终能够在有限的步骤内完成其工作，则该算法是无阻塞的。在这种情况下，继续运行的单个线程被称为单独执行，而无阻塞算法有时被称为单独终止算法，因为该单独线程最终会在所有其他线程被挂起时终止。任何使用互斥锁或自旋锁的算法都不可能实现无阻塞，因为如果任何线程在持有锁时被挂起，该单独线程可能会永远陷入等待该锁的状态。

• 锁自由。锁自由的技术定义是，在程序无限长时间的运行中，将完成无限数量的操作。直观地说，无锁算法保证系统中的某个线程始终可以取得进展；换句话说，如果一个线程被任意挂起，所有其他线程仍然可以取得进展。这再次排除了使用互斥锁或自旋锁的可能性，因为如果持有锁的线程被挂起，它可能会导致其他线程阻塞。无锁算法通常基于事务：如果另一个线程中断事务，则事务可能会失败，在这种情况下，事务将被回滚并重试，直到成功。这种方法避免了死锁，但它可能导致某些线程饿死。换句话说，某些线程可能会陷入无限期失败和重试其事务的循环中，而其他线程的事务则始终成功。

• 免等待。免等待算法不仅提供免锁的所有保证，还保证免饥饿。换句话说，所有线程都可以继续执行，并且不允许任何线程无限期地处于饥饿状态。

“无锁编程”这个术语有时被宽泛地用来指代任何避免阻塞的算法，但从技术上讲，无阻塞、无锁和无等待算法的正确术语是“非阻塞算法”。

非阻塞算法的主题非常广泛，至今仍是一个开放的研究领域。要完整地讨论这个主题，可能需要写一本书。在本章中，我们将介绍一些无锁编程的基本原理。我们将首先探讨数据竞争漏洞的真正原因。然后，我们将了解互斥锁在以下环境中的实际实现方式：

ing algorithm is prone to deadlock, livelock, starvation and priority inversion.

- *Obstruction freedom*. An algorithm is obstruction-free if we can guarantee that a single thread will always complete its work in a bounded number of steps, when all of the other threads in the system are suddenly suspended. The single thread that continues to run is said to be performing a *solo execution* in this scenario, and an obstruction-free algorithm is sometimes called *solo-terminating* because the solo thread eventually terminates while all other threads are suspended. No algorithm that uses a mutex lock or spin lock can be obstruction-free, because if any thread were to be suspended while it is holding a lock, the solo thread might become stuck forever waiting for that lock.

- *Lock freedom*. The technical definition of lock freedom is that in any infinitely-long run of the program, an infinite number of operations will be completed. Intuitively, a lock-free algorithm guarantees that *some* thread in the system can always make progress; in other words, if one thread is arbitrarily suspended, all others can still make progress. This again precludes the use of mutex or spin locks, because if a thread holding a lock is suspended, it can cause other threads to block. A lock-free algorithm is typically transaction-based: A transaction may fail if another thread interrupts it, in which case the transaction is rolled back and retried until it succeeds. This approach avoids deadlock, but it can allow some threads to starve. In other words, certain threads might get stuck in a loop of failing and retrying their transactions indefinitely, while other threads' transactions always succeed.

- *Wait freedom*. A wait-free algorithm provides all the guarantees of lock freedom, but also guarantees starvation freedom. In other words, all threads can make progress, and no thread is allowed to starve indefinitely.

The term "lock-free programming" is sometimes used loosely to refer to *any* algorithm that avoids blocking, but technically speaking the correct term for obstruction-free, lock-free and wait-free algorithms as a whole is "*non-blocking algorithm*."

The topic of non-blocking algorithms is vast, and is still an open area of research. A complete discussion of the topic would require an entire book of its own. In this chapter, we'll introduce some of the basic principles of lock-free programming. We'll begin by exploring the true casues of data race bugs. Then we'll have a look at how mutexes are actually implemented under the

并学习如何实现我们自己的廉价自旋锁。最后，我们将介绍一个简单的无锁链表的实现。这些讨论应该足以让您了解无锁数据结构和算法的大致情况，并为进一步阅读和实验无锁技术提供一个坚实的起点。

4.9.1 数据竞争错误的原因

在 4.5.3.2 节中，我们提到，当一个关键操作被针对同一共享数据的另一个关键操作打断时，就会发生数据竞争错误。事实证明，数据竞争错误还可能通过另外两种方式出现，如果我们要实现自己的自旋锁或编写无锁算法，就需要理解所有这些方式。数据竞争错误可能被引入并发程序中：

- 通过一个关键操作中断另一个关键操作，
- 通过编译器和 CPU 执行的指令重新排序优化，以及
- 由于硬件特定的内存排序语义。

让我们进一步分解一下：

- 由于抢占式多任务处理和/或在多核上运行，线程会一直相互中断。然而，当一个关键操作被同一共享数据对象上的另一个关键操作中断时，可能会发生数据竞争错误。

- 优化编译器通常会重新排序指令，以尽量减少流水线停顿。同样，CPU 内部的乱序执行逻辑也可能导致指令的执行顺序与程序顺序不同。指令重新排序可以保证不会改变单线程程序的可观察行为。但它可能会改变两个或多个线程协作共享数据的方式，从而将错误引入并发程序。

- 由于计算机内存控制器内部的积极优化，读取或写入指令的效果有时会相对于系统中的其他读取和/或写入操作有所延迟。与编译器优化和乱序执行一样，这些内存控制器优化的设计目的并非改变单线程程序的可观察行为。然而，这些优化可能会改变并发系统中关键读取和/或写入操作对的顺序。

hood, and learn how to implement our own inexpensive *spin locks*. Finally, we'll present an implementation of a simple lock-free linked list. This discussion ought to be sufficient to give you a flavor for what lock-free data structures and algorithms tend to look like, and should provide a solid jumping off point for further reading and experimentation with lock-free techniques.

4.9.1 Causes of Data Race Bugs

In Section 4.5.3.2, we said that data race bugs occur when a critical operation is interrupted by another critical operation on the same shared data. As it turns out, there are two other ways in which data race bugs can arise, and if we're going to implement our own spin locks or write lock-free algorithms, we'll need to understand them all. Data race bugs can be introduced into a concurrent program:

- via the *interruption* of one critical operation by another,
- by the *instruction reordering* optimizations performed by the compiler and CPU, and
- as a result of hardware-specific *memory ordering semantics*.

Let's break these down a bit further:

- Threads interrupt one another all the time as a result of preemptive multitasking and/or by running them on multiple cores. However, when a *critical* operation is interrupted by another critical operation on the *same* shared data object, a data race bug can occur.

- Optimizing compilers often reorder instructions in an attempt to minimize pipeline stalls. Likewise, the out-of-order execution logic within the CPU can cause instructions to be executed in an order that differs from program order. Instruction reordering is guaranteed not to alter the observable behavior of a single-threaded program. But it *can* alter the way in which two or more threads *cooperate* to share data, thereby introducing bugs into a concurrent program.

- Thanks to aggressive optimizations within a computer's memory controller, the *effects* of a read or write instruction can sometimes become *delayed* relative to other reads and/or writes in the system. As with compiler optimizations and OOO execution, these memory controller optimizations are designed never to alter the observable behavior of a single-threaded program. However, these optimizations *can* change the order of a critical pair of reads and/or writes in a concurrent system,

从而阻止线程以可预测的方式共享数据。本书中，我们将这类并发错误称为“内存排序错误”。

为了保证关键操作不存在数据竞争，从而具有原子性，我们必须确保这三件事都不会发生。

4.9.2 实现原子性

首先，让我们解决如何使关键操作原子化（即不可中断）的问题。之前，我们稍微动了一下脑筋，说通过将关键操作包装在互斥锁的加锁/解锁对中，就能神奇地将它们转换为原子操作。但是互斥锁究竟是如何工作的呢？

4.9.2.1 通过禁用中断实现原子性

为了防止其他线程中断我们的操作，我们可以尝试在执行操作之前禁用中断，并确保在操作完成后重新启用中断。这当然可以防止内核在原子操作过程中上下文切换到另一个线程。但是，这种方法仅适用于使用抢占式多任务的单核机器。

可以通过执行机器语言指令（例如，在 Intel x86 架构上为 cli ，即“清除中断使能位”）来禁用中断。但此类指令仅影响执行该指令的内核。其他内核将继续运行其线程（中断和抢占式多线程仍然处于启用状态），而这些线程仍然可能中断我们的操作。

因此这种方法在现实世界中的适用性有限。

4.9.2.2 原子指令

“原子”一词指的是将操作分解成越来越小的部分，直到达到不可分割的粒度。这个想法引出了一个问题：是否存在一些天生具有原子性的机器语言指令？换句话说，CPU 是否保证某些指令是不可中断的？

这些问题的答案是“是”，但有一些注意事项。当然，有些机器语言指令永远不能被假定为原子执行。其他指令是原子的，但仅限于操作某些类型的数据时。某些 CPU 允许通过在汇编语言中为指令指定前缀来强制所有指令以原子方式执行。（英特尔 x86 ISA 的 lock 前缀就是一个例子。）

thereby preventing threads from sharing data in a predictable way. In this book, we'll refer to these kinds of concurrency bugs as *memory ordering bugs*.

In order to guarantee that a critical operation is free from data races, and is therefore atomic, we must ensure that none of these three things can happen.

4.9.2 Implementing Atomicity

First, let's tackle the problem of making a critical operation atomic (i.e., uninterruptable). Before, we waved our hands a bit and said that by wrapping our critical operations in a *mutex* lock/unlock pair, we can magically transform them into atomic operations. But how does a mutex actually work?

4.9.2.1 Atomicity by Disabling Interrupts

To prevent other threads from interrupting our operation, we could try disabling interrupts just prior to performing the operation, making sure to reenable them after the operation has been completed. That would certainly prevent the kernel from context-switching to another thread in the middle of our atomic operation. However, this approach only works on a single-core machine using preemptive multitasking.

Interrupts are disabled by executing a machine language instruction (such as `cli`, "clear interrupt enable bit," on an Intel x86 architecture). But this kind of instruction only affects the core that executed it. The other cores would continue to run their threads (with interrupts, and therefore preemptive multithreading, still enabled), and those threads could still interrupt our operation. So this approach has only limited applicability in the real world.

4.9.2.2 Atomic Instructions

The term "atomic" suggests the notion of breaking an operation down into smaller and smaller pieces, until we reach a granularity that is indivisible. This idea raises the question: Are there any machine language instructions that are *naturally* atomic? In other words, does the CPU guarantee that some instructions are uninterruptable?

The answer to these questions is "yes," but with a few caveats. There are most certainly some machine language instructions that can *never* be assumed to execute atomically. Other instructions are atomic, but only when operating on certain kinds of data. Some CPUs permit virtually any instruction to be *forced* to execute atomically by specifying a prefix on the instruction in assembly language. (The Intel x86 ISA's `lock` prefix is one example.)

这对并发程序员来说是个好消息。事实上，正是这些原子指令的存在，才使得我们能够实现诸如互斥锁和自旋锁之类的原子性工具，进而使我们能够将更大规模的操作原子化。

不同的 CPU 和 ISA 提供不同的原子指令集，并受不同的规则约束。但我们可以将所有原子指令概括为两类：

- 原子读写，以及
- 原子读-修改-写（RMW）指令。

4.9.2.3 原子读取和写入

在大多数 CPU 上，我们可以合理地确定，对四字节对齐的 32 位整数的读写操作是原子性的。话虽如此，每个处理器都有所不同，因此在依赖任何特定指令的原子性之前，务必先查阅 CPU 的 ISA 文档。

某些 CPU 还支持对较小或较大对象（例如单字节或 64 位整数）进行原子读写，前提是它们对齐到自身大小的倍数。这是因为在大多数 CPU 上，读写一个位宽等于或小于寄存器宽度（有时是缓存行宽度）的对齐整数可以在单个内存访问周期内完成。由于 CPU 使用离散时钟同步执行操作，因此内存周期不会被中断，即使是被其他核心中断。因此，读写操作实际上是原子的。

未对齐的读写通常不具有这种原子性属性。
这是因为，为了读取或写入未对齐的对象，CPU 通常会组合两次对齐的内存访问。因此，读取或写入操作可能会被中断，并且我们无法假设它是原子性的。（有关对齐的更多详细信息，请参阅第 3.3.7.1 节。）

4.9.2.4 原子读取-修改-写入

原子读写不足以实现一般意义上的原子操作。为了实现像互斥锁这样的锁定机制，我们需要能够从内存中读取变量的内容，对该变量执行某些操作，然后将结果写回内存，并且整个过程不会被打断。

所有现代 CPU 都通过提供至少一条原子读-修改-写 (RMW) 指令来支持并发。在接下来的章节中，我们将介绍几种不同类型的原子 RMW 指令。每种指令都有其优缺点。所有这些指令都可以用来实现互斥锁或自旋锁。

This is good news for concurrent programmers. In fact, it is the existence of these *atomic instructions* that permits us to implement atomicity tools such as mutexes and spin locks, which in turn permit us to make larger-scale operations atomic.

Different CPUs and ISAs provide different sets of atomic instructions, governed by different rules. But we can generalize all atomic instructions as falling into two categories:

- atomic reads and writes, and
- atomic read-modify-write (RMW) instructions.

4.9.2.3 Atomic Reads and Writes

On most CPUs, we can be reasonably certain that a read or write of a four-byte-aligned 32-bit integer will be atomic. That being said, every processor is different, so it's important to always consult your CPU's ISA documentation before relying on the atomicity of any particular instruction.

Some CPUs also support atomic reads and writes of smaller or larger objects, such as single bytes or 64-bit integers, again assuming they are aligned to a multiple of their own size. This is true because on most CPUs, reading and writing an aligned integer whose width in bits is equal to or smaller than the width of a register (or sometimes the width of a cache line) can be performed in a single *memory access cycle*. Because a CPU performs its operation in lock-step with a discrete clock, a memory cycle can't be interrupted, even by another core. As a result, the read or write is effectively atomic.

Misaligned reads and writes usually don't have this atomicity property. This is because in order to read or write a misaligned object, the CPU usually composes two aligned memory accesses. As such, the read or write might be interrupted, and we cannot assume it will be atomic. (See Section 3.3.7.1 for more details on alignment.)

4.9.2.4 Atomic Read-Modify-Write

Atomic reads and writes aren't enough to implement atomic operations in a general sense. In order to implement a locking mechanism like a mutex, we need to be able to read a variable's contents from memory, perform some operation on that variable, and then write the results back to memory, all without being interrupted.

All modern CPUs support concurrency by providing at least one atomic read-modify-write (RMW) instruction. In the following sections, we'll have a look at a few different kinds of atomic RMW instructions. Each has its pros and cons. All of them can be used to implement a mutex or spin lock.

4.9.2.5 测试和设置

最简单的 RMW 指令称为测试并设置 (TAS)。TAS 指令实际上并不测试并设置值。相反，它原子地将布尔变量设置为 1（真），并返回其先前的值（以便可以测试该值）。

```
// pseudocode for the test-and-set instruction
bool TAS(bool* pLock)
{
    // atomically...
    const bool old = *pLock;
    *pLock = true;
    return old;
}
```

测试并设置指令可用于实现一种称为自旋锁的简单锁。以下是一些伪代码来说明这一概念。在本例中，我们使用一个假设的编译器内部函数 _tas() 来将一条 TAS 机器语言指令发送到我们的代码中。如果目标 CPU 支持该指令，不同的编译器会为该指令提供不同的内部函数。例如，在 Visual Studio 中，TAS 内部函数名为 _interlockedbittest andset().

```
void SpinLockTAS(bool* pLock)
{
    while (_tas(pLock) == true)
    {
        // someone else has lock -- busy-wait...
        PAUSE();
    }

    // when we get here, we know that we successfully
    // stored a value of true into *pLock AND that it
    // previously contained false, so no one else has
    // the lock -- we're done
}
```

这里，PAUSE() 宏表示使用编译器内部函数（例如 Intel 的 SSE2 _mm_pause()）来降低忙等待循环期间的功耗。有关为什么建议在忙等待循环中尽可能使用暂停指令的详细信息，请参阅第 4.4.6.4 节。

这里需要强调的是，上述示例仅用于说明目的。由于缺乏适当的内存，它并非 100% 正确。

4.9.2.5 Test and Set

The simplest RMW instruction is known as *test-and-set* (TAS). The TAS instruction doesn't actually test and set a value. Rather, it atomically sets a Boolean variable to 1 (true) and returns its *previous* value (so that the value can be tested).

```
// pseudocode for the test-and-set instruction
bool TAS(bool* pLock)
{
    // atomically...
    const bool old = *pLock;
    *pLock = true;
    return old;
}
```

The test-and-set instruction can be used to implement a simple kind of lock called a *spin lock*. Here's some pseudocode illustrating the idea. In this example, we using a hypothetical *compiler intrinsic* called `_tas()` to emit a TAS machine language instruction into our code. Different compilers will provide different intrinsics for this instruction, if the target CPU supports it. For example, under Visual Studio the TAS intrinsic is named `_interlockedbittest andset()`.

```
void SpinLockTAS(bool* pLock)
{
    while (_tas(pLock) == true)
    {
        // someone else has lock -- busy-wait...
        PAUSE();
    }

    // when we get here, we know that we successfully
    // stored a value of true into *pLock AND that it
    // previously contained false, so no one else has
    // the lock -- we're done
}
```

Here, the `PAUSE()` macro indicates the use of a compiler intrinsic, such as Intel's SSE2 `_mm_pause()`, to reduce power consumption during the busy-wait loop. See Section 4.4.6.4 for details on why it's advisable to use a pause instruction inside a busy-wait loop where possible.

It's important to stress here that the above example is intended for illustrative purposes only. It is not 100% correct because it lacks proper memory

围栏。我们将在 4.9.7 节中展示一个完整的自旋锁工作示例。

4.9.2.6 交换

某些 ISA（例如 Intel x86）提供了原子交换指令。该指令交换两个寄存器的内容，或者一个寄存器与内存中的位置。在 x86 上，当交换寄存器和内存时，该指令默认是原子的（这意味着它的行为就像该指令前面带有 lock 前缀一样）。

下面介绍如何使用原子交换指令实现自旋锁。在此示例中，我们使用 Visual Studio 的 _InterlockedExchange() 编译器内部函数将 Intel x86 xchg 指令添加到代码中。（再次强调，如果没有适当的内存隔离，此示例是不完整的，并且无法可靠运行。完整实现请参见第 4.9.7 节。）

```
void SpinLockXCHG(bool* pLock)
{
    bool old = true;
    while (true)
    {
        // emit the xchg instruction for 8-bit words
        _InterlockedExchange8(old, pLock);
        if (!old)
        {
            // if we get back false,
            // then the lock succeeded
            break;
        }
        PAUSE();
    }
}
```

在 Microsoft Visual Studio 中，所有以下划线开头的“互锁”函数都是编译器内部函数——它们只是将相应的汇编语言指令直接发送到你的代码中。Windows SDK 提供了一组名称类似但不带下划线的函数——这些函数尽可能地以内部函数的形式实现，但由于它们涉及内核调用，因此开销更大。

4.9.2.7 比较和交换

一些 CPU 提供了称为比较并交换 (CAS) 的指令。该指令检查内存位置中的现有值，并当且仅当现有值与预期值匹配时，原子地将其与新值交换。

fencing. We'll present a complete working example of a spin lock in Section 4.9.7.

4.9.2.6 Exchange

Some ISAs like Intel x86 offer an atomic *exchange* instruction. This instruction swaps the contents of two registers, or a register and a location in memory. On x86, it is atomic by default when exchanging a register with memory (meaning it acts as if the instruction had been preceded with the `lock` prefix).

Here's how to implement a spin lock using an atomic *exchange* instruction. In this example, we're using Visual Studio's `_InterlockedExchange()` *compiler intrinsic* to emit the Intel x86 `xchg` instruction into our code. (Again, note that without proper memory fencing this example is incomplete and won't work reliably. See Section 4.9.7 for a complete implementation.)

```
void SpinLockXCHG(bool* pLock)
{
    bool old = true;
    while (true)
    {
        // emit the xchg instruction for 8-bit words
        _InterlockedExchange8(old, pLock);
        if (!old)
        {
            // if we get back false,
            // then the lock succeeded
            break;
        }
        PAUSE();
    }
}
```

Under Microsoft Visual Studio, all of the "interlocked" functions named with a leading underscore are *compiler intrinsics*—they simply emit the appropriate assembly language instruction directly into your code. The Windows SDK provides a set of similarly-named functions *without* the leading underscore—those functions are implemented in terms of the intrinsic where possible, but they're much more expensive because they involve making a kernel call.

4.9.2.7 Compare and Swap

Some CPUs provide an instruction known as *compare-and-swap* (CAS). This instruction checks the existing value in a memory location, and atomically swaps it with a new value *if and only if* the existing value matches an expected value

由程序员提供。如果操作成功，则返回 true，表示内存位置包含预期值。如果操作失败，因为位置内容不符合预期，则返回 false。

CAS 可以对大于布尔值的值进行操作。通常至少为 32 位和 64 位整数提供变体，但也可能支持更小的字长。

CAS 指令的行为由以下伪代码说明：

```
// pseudocode for compare and swap
bool CAS(int* pValue, int expectedValue, int newValue)
{
    // atomically...
    if (*pValue == expectedValue)
    {
        *pValue = newValue;
        return true;
    }
    return false;
}
```

为了使用 CAS 实现任何原子读取-修改-写入操作，我们通常采用以下策略：

1. 读取我们尝试更新的变量的旧值。
2. 以我们认为合适的方式修改值。
3. 写入结果时，使用CAS指令，而不是常规写入。
4. 迭代直至CAS成功。

CAS 指令允许我们检测数据竞争，方法是将写入时内存位置的实际值与调用读取-修改-写入操作之前的值进行比较。在没有竞争的情况下，CAS 指令的行为与写入指令相同。但是，如果内存中的值在读取和写入之间发生了变化，我们就知道有其他线程抢先了一步。在这种情况下，我们会撤退并重试。

通过比较交换实现自旋锁看起来就像这样，同样使用一个假设的编译器内部函数 _cas() 来发出 CAS 指令。（此示例再次省略了使其在所有硬件上可靠工作所需的内存栅栏——有关功能齐全的自旋锁，请参阅第 4.9.7 节。）

provided by the programmer. It returns true if the operation succeeded, meaning that the memory location contained the expected value. It returns false if the operation failed because the location's contents were not as expected.

CAS can operate on values larger than a Boolean. Variants are typically provided at least for 32- and 64-bit integers, although smaller word sizes may also be supported.

The behavior of the CAS instruction is illustrated by the following pseudocode:

```
// pseudocode for compare and swap
bool CAS(int* pValue, int expectedValue, int newValue)
{
    // atomically...
    if (*pValue == expectedValue)
    {
        *pValue = newValue;
        return true;
    }
    return false;
}
```

To implement any atomic read-modify-write operation using CAS, we generally apply the following strategy:

1. Read the old value of the variable we're attempting to update.
2. Modify the value in whatever way we see fit.
3. When writing the result, use the CAS instruction instead of a regular write.
4. Iterate until the CAS succeeds.

The CAS instruction allows us to detect data races, by comparing the value that's actually in the memory location at the time of the write with its value *before* the invocation of our read-modify-write operation. In the absence of a race, the CAS instruction acts just like a write instruction. But if the value in memory changes between the read and the write, we know that some other thread beat us to the punch. In that case, we back off and try again.

Implementing a spin lock via compare-and-swap would look something like this, again using a hypothetical compiler intrinsic called `_cas()` to emit the CAS instruction. (This example once again omits the memory fences that would be required to make it work reliably on all hardware—see Section 4.9.7 for a fully-functional spin lock.)

```
void SpinLockCAS(int* pValue)
{
    const int kLockValue = -1; // 0xFFFFFFFF
    while (!_cas(pValue, 0, kLockValue))
    {
        // must be locked by someone else -- retry
        PAUSE();
    }
}
```

以下是我们使用 CAS 实现原子增量的方法。

```
void AtomicIncrementCAS(int* pValue)
{
    while (true)
    {
        const int oldValue = *pValue; // atomic read
        const int newValue = oldValue + 1;
        if (_cas(pValue, oldValue, newValue))
        {
            break; // success!
        }
        PAUSE();
    }
}
```

在 Intel x86 ISA 上，CAS 指令称为 cmpxchg ，它可以通过 Visual Studio 的 _InterlockedCompareExchange() 编译器内部函数发出。

4.9.2.8 ABA问题

值得一提的是，CAS 指令无法检测一种特定类型的数据争用。假设有一个原子 RMW 写入操作，其中读取操作看到的是值 A。在我们能够发出 CAS 指令之前，另一个线程抢占了我们的执行权，或者在另一个核心上运行，并将两个值写入我们尝试原子更新的位置：首先写入值 B，然后再次写入值 A。当我们的 CAS 指令最终执行时，它无法区分它读取的第一个 A 和另一个线程写入的 A 。因此，它会“认为”没有发生数据争用，而实际上发生了。这就是所谓的 ABA 问题。

4.9.2.9 链接加载/条件存储

一些 CPU 将比较交换操作分解成一对指令，称为链接加载指令和条件存储指令 (LL/SC)。链接加载指令


```
void SpinLockCAS(int* pValue)
{
    const int kLockValue = -1; // 0xFFFFFFFF
    while (!_cas(pValue, 0, kLockValue))
    {
        // must be locked by someone else -- retry
        PAUSE();
    }
}
```

And here's how we'd implement an atomic increment using CAS.

```
void AtomicIncrementCAS(int* pValue)
{
    while (true)
    {
        const int oldValue = *pValue; // atomic read
        const int newValue = oldValue + 1;
        if (_cas(pValue, oldValue, newValue))
        {
            break; // success!
        }
        PAUSE();
    }
}
```

On the Intel x86 ISA, the CAS instruction is called `cmpxchg`, and it can be emitted with Visual Studio's `_InterlockedCompareExchange()` compiler intrinsic.

4.9.2.8 ABA Problem

We should mention that the CAS instruction suffers from an inability to detect one specific kind of data race. Consider an atomic RMW write operation in which the read sees the value A. Before we're able to issue the CAS instruction, another thread preempts us, or runs on another core, and writes two values into the location we're trying to atomically update: First it writes the value B, and then it writes the value A again. When our CAS instruction does finally execute, it won't be able to tell the difference between the first A it read and the A that was written by the other thread. Hence it will "think" that no data race occurred, when in fact one did. This is known as the *ABA problem*.

4.9.2.9 Load Linked/Store Conditional

Some CPUs break the compare-and-swap operation into a *pair* of instructions known as *load linked* and *store conditional* (LL/SC). The *load linked* instruction

以原子方式读取内存位置的值，并将地址存储在一个称为链接寄存器的特殊 CPU 寄存器中。存储条件指令会将值写入给定地址，但前提是该地址与链接寄存器的内容匹配。如果写入成功，则返回 true；如果写入失败，则返回 false。

任何总线上的写入操作（包括条件存储）□□都会将链接寄存器清零。这意味着 LL/SC 指令对能够检测数据竞争，因为如果在 LL 和 SC 指令之间发生任何写入操作，SC 指令就会失败。

LL/SC 指令对的使用方式与常规读取与 CAS 指令配对的方式大致相同。具体来说，原子读取-修改-写入操作将使用以下策略实现：

1. 通过 LL 指令读取变量的旧值。
2. 以我们认为合适的方式修改值。
3. 使用 SC 指令写入结果。
4. 迭代直至 SC 成功。

以下是我们如何使用 LL/SC 实现原子增量（分别使用假设的编译器内部函数 _ll() 和 _sc() 发出 LL 和 SC 指令）：

```
void AtomicIncrementLLSC(int* pValue)
{
    while (true)
    {
        const int oldValue = _ll(*pValue);
        const int newValue = oldValue + 1;
        if (_sc(pValue, newValue))
        {
            break; // success!
        }
        PAUSE();
    }
}
```

由于任何总线写入都会清除链接寄存器，因此 SC 指令可能会意外失败。但这不会影响使用 LL/SC 实现的原子 RMW 的正确性——这只意味着我们的忙等待循环最终可能会比预期多执行几次迭代。

reads the value of a memory location atomically, and also stores the address in a special CPU register known as the *link register*. The *store conditional* instruction writes a value into the given address, but only if the address matches the contents of the link register. It returns true if the write succeeded, or false if it failed.

Any write operation on the bus (including a store conditional) clears the link register to zero. This means that an LL/SC instruction pair is capable of detecting data races, because if any write occurs between the LL and SC instructions, the SC will fail.

An LL/SC instruction pair is used in much the same way a regular read is paired with a CAS instruction. Specifically, an atomic read-modify-write operation would be implemented using the following strategy:

1. Read the old value of the variable via an LL instruction.
2. Modify the value in whatever way we see fit.
3. Write the result using the SC instruction.
4. Iterate until the SC succeeds.

Here's how we'd implement an atomic increment using LL/SC (using the hypothetical compiler intrinsics `_ll()` and `_sc()` to emit the LL and SC instructions, respectively):

```
void AtomicIncrementLLSC(int* pValue)
{
    while (true)
    {
        const int oldValue = _ll(*pValue);
        const int newValue = oldValue + 1;
        if (_sc(pValue, newValue))
        {
            break; // success!
        }
        PAUSE();
    }
}
```

Because the link register is cleared by *any* bus write, the SC instruction may fail spuriously. But that doesn't affect the correctness of an atomic RMW implemented with LL/SC—it just means that our busy-wait loop might end up executing a few more iterations than we'd expect.

图 4.35。比较交换 (CAS) 指令读取内存位置，进行比较，然后有条件地写入同一位置。因此，它需要两个内存访问阶段，这使得它在简单的流水线 CPU 架构中实现起来比链接/存储条件 (LL/SC) 指令对更困难，因为后两者每个指令只需要一个内存访问阶段。

4.9.2.10 LL/SC 相对于 CAS 的优势

LL/SC 指令对与单个 CAS 指令相比具有两个明显的优势。

首先，由于在总线上执行任何写入操作时 SC 指令都会失败，因此 LL/SC 对不易出现 ABA 问题。

其次，LL/SC 对比 CAS 指令更适合流水线。最简单的流水线由五个阶段组成：取指、解码、执行、内存访问和寄存器写回。但 CAS 指令需要两个内存访问周期：一个用于读取内存位置以便将其与预期值进行比较，另一个用于在比较通过后写入结果。这意味着支持 CAS 的流水线必须包含一个额外的内存访问阶段，而该阶段大□□多数情况下都不会使用。另一方面，LL 和 SC 指令各自只需要一个内存访问周期，因此它们更适合只有一个内存访问阶段的流水线。图 4.35 从流水线的角度比较了 CAS 和 LL/SC。

4.9.2.11 强弱比较交换

C++11 标准库提供了可移植的函数来执行原子比较交换操作。这些函数在某些目标硬件上可以通过 CAS 指令实现，在其他硬件上可以通过 LL/SC 指令对实现。由于条件存储指令可能出现虚假失败，C++11 提供了两种比较交换类型：强比较交换和弱比较交换。强比较交换可以“隐藏”虚假的条件存储失败，避免程序崩溃。

Figure 4.35. The compare and swap (CAS) instruction reads a memory location, performs a comparison, and then conditionally writes to that same location. It therefore requires two memory access stages, making it more difficult to implement within a simple pipelined CPU architecture than the linked/store conditional (LL/SC) instruction pair, each of which requires only a single memory access stage.

4.9.2.10 Advantages of LL/SC over CAS

The LL/SC instruction pair offers two distinct advantages over the single CAS instruction.

First, because the SC instruction fails whenever *any* write is performed on the bus, an LL/SC pair is not prone to the ABA problem.

Second, an LL/SC pair is more pipeline-friendly than a CAS instruction. The simplest pipeline is comprised of five stages: fetch, decode, execute, memory access and register write-back. But a CAS instruction requires two memory access cycles: One to read the memory location so it can be compared to the expected value, and one to write the result if the comparison passes. This means that a pipeline that supports CAS has to include an additional memory access stage that goes unused most of the time. On the other hand, the LL and SC instructions each only require a single memory access cycle, so they fit more naturally into a pipeline with only one memory access stage. A comparison of CAS and LL/SC from a pipelining perspective is shown in Figure 4.35.

4.9.2.11 Strong and Weak Compare-Exchange

The C++11 standard library provides portable functions for performing atomic compare-exchange operations. These may be implemented by a CAS instruction on some target hardware, or by an LL/SC instruction pair on other hardware. Because of the possibility of spurious failures of the store-conditional instruction, C++11 provides two varieties of compare-exchange: strong and weak. Strong compare-exchange “hides” spurious SC failures from the pro-

语法，而弱比较交换函数则不然。在线搜索“强比较和交换 Lawrence Crowl”，可以找到一篇关于 C++11 中强和弱比较交换函数背后原理的论文。

4.9.2.12 原子 RMW 指令的相对强度

值得注意的是，在并发系统中，TAS 指令在多线程之间达成共识方面比 CAS 和 LL/SC 指令弱。此处的“共识”指的是线程之间就共享变量的值达成一致（即使系统中某些线程发生故障）。

由于 TAS 指令仅对布尔值进行操作，因此它只能解决两个并发线程的无等待一致性问题。CAS 指令对 32 位值进行操作，因此它可以解决任意数量线程的这个问题。

无等待共识机制的主题远远超出了我们本文的讨论范围；它主要在构建容错系统时才会被关注。如果您对容错机制感兴趣，可以在维基百科上搜索“共识（计算机科学）”来了解更多关于共识机制的信息。

4.9.3 障碍

中断并非数据竞争错误的唯一原因。编译器和 CPU 还会通过执行指令重排序优化，将一些细微的错误引入我们的并发程序中，如第 4.2.5.1 节所述。

编译器优化和乱序执行的基本规则是，其优化不应影响单个线程的行为。然而，无论是编译器还是 CPU 的控制逻辑，都无法获知系统中可能正在运行的其他线程，以及它们可能正在执行的操作。因此，这条基本规则不足以防止指令重排序优化在并发程序中引入错误。

操作系统提供的线程同步原语（例如互斥锁）经过精心设计，可以避免指令重排序优化可能导致的并发错误。既然我们正在研究互斥锁的实现方式，那么接下来就让我们看看如何手动避免这些问题。

grammer, while weak compare-exchange does not. Search for “Strong Compare and Exchange Lawrence Crowl” online for a paper on the rationale behind strong and weak compare-exchange functions in C++11.

4.9.2.12 Relative Strength of Atomic RMW Instructions

It’s interesting to note that the TAS instruction is *weaker* than the CAS and LL/SC instructions in terms of achieving *consensus* between multiple threads in a concurrent system. Consensus in this context refers to an agreement between threads about the value of a shared variable (even if some threads in the system experience a failure).

Because the TAS instruction only operates on a Boolean value, it can only solve a problem known as the *wait-free consensus problem* for two concurrent threads. The CAS instruction operates on a 32-bit value, so it can solve this problem for any number of threads.

The topic of wait-free consensus is well beyond our scope here; it’s mostly of interest when building fault-tolerant systems. If you’re interested in fault tolerance, you can read more about the consensus problem by searching for “consensus (computer science)” on Wikipedia.

4.9.3 Barriers

Interruptions aren’t the only cause of data race bugs. Compilers and CPUs also conspire to introduce subtle bugs into our concurrent programs by means of the *instruction reordering* optimizations they perform, as described in Section 4.2.5.1.

The cardinal rule of compiler optimizations and out-of-order execution is that their optimizations shall have no visible effects on the behavior of *a single thread*. However, neither the compiler nor the CPU’s control logic has any knowledge of what other threads might be running in the system, or what they might be doing. As a result, the cardinal rule isn’t sufficient to prevent instruction reordering optimizations from introducing bugs into concurrent programs.

The thread synchronization primitives provided by the operating system (mutexes et al.) are carefully crafted to avoid the concurrency bugs that can be caused by instruction reordering optimizations. But now that we’re investigating how mutexes are implemented, let’s take a look at how to avoid these problems manually.

4.9.3.1 指令重新排序如何导致并发错误

为了说明指令重排序在并发软件中可能引发的各种问题，我们再次考虑 4.6.3 节中的生产者-消费者问题。我们简化了这个示例，并删除了互斥锁，以便揭示指令重排序可能引入的 bug。

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // produce some data
    g_data = 42;

    // inform the consumer
    g_ready = 1;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // consume the data
    ASSERT(g_data == 42);
}
```

在 32 位整数对齐读写操作具有原子性的 CPU 上，此示例实际上不需要互斥锁。然而，没有什么可以阻止编译器或 CPU 的乱序执行逻辑将生产者向 g_ready 写入 1 的操作重新排序，使其发生在向 g_data 写入 42 之前。同样，理论上，编译器可以重新排序消费者对 g_data 是否等于 42 的检查，使其发生在 while 循环之前。因此，即使我们所有的读写操作都是原子的，这段代码的行为也可能无法可靠。

指令重排序实际上发生在汇编语言层面，因此它可能比 C/C++ 程序中语句的重排序更加微妙。例如，以下 C/C++ 代码：

```
A = B + 1;
B = 0;
```


4.9.3.1 How Instruction Reordering Causes Concurrency Bugs

As an example of the kinds of problems instruction reordering can cause in concurrent software, consider again the producer-consumer problem from Section 4.6.3. We've simplified the example, and removed the mutexes so we can expose the bugs that can be introduced by instruction reordering.

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // produce some data
    g_data = 42;

    // inform the consumer
    g_ready = 1;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // consume the data
    ASSERT(g_data == 42);
}
```

On a CPU on which aligned reads and writes of 32-bit integers are atomic, this example doesn't actually require mutexes. However, there's nothing to prevent the compiler or the CPU's out-of-order execution logic from reordering the producer's write of 1 into `g_ready` so that it occurs *before* the write of 42 into `g_data`. Likewise, in theory the compiler could reorder the consumer's check that `g_data` is equal to 42 so that it happens *before* the while loop. So even though all of our reads and writes are atomic, this code may not behave reliably.

Instruction reordering really happens at the assembly language level, so it may be a lot more subtle than a reordering of the *statements* in a C/C++ program. For example, the following C/C++ code:

```
A = B + 1;
B = 0;
```

将产生以下 Intel x86 汇编代码：

```
mov  eax,[B]
add  eax,1
mov  [A],eax
mov  [B],0
```

编译器可以很好地按如下方式重新排序指令，而不会在单线程执行中产生任何明显的影响：

```
mov  eax,[B]
mov  [B],0  ;; write to B before A!
add  eax,1
mov  [A],eax
```

如果第二个线程在读取 A 的值之前等待 B 变为零，则在应用此编译器优化后它将停止正常运行。

Jeff Preshing 就此主题写了一篇很棒的博客文章，网址为 http://preshing.com/20120625/memory-ordering-at-compile-time/。（上面的汇编语言示例就出自这里。）我强烈推荐 Jeff 所有关于并发编程的文章，所以一定要去看看。

4.9.3.2 C/C++ 中的 Volatile（以及它为什么对我们没有帮助）

如何防止编译器对关键的读写操作进行重新排序？在 C 和 C++ 中， volatile 类型限定符可以保证对变量的连续读写操作不会被编译器“优化掉”，所以这听起来是个不错的主意。然而，由于多种原因，它并不可靠。

C/C++ 中的 volatile 限定符实际上旨在使内存映射 I/O 和信号处理程序可靠地工作。因此，它提供的唯一保证是，标记为 volatile 的变量的内容不会被缓存到寄存器中——每次访问时，该变量的值都会直接从内存中重新读取。一些编译器确实保证在读取或写入 volatile 变量时指令不会重新排序，但并非所有编译器都这样做，有些编译器仅在针对特定 CPU 时或仅在向编译器传递特定命令行选项时才提供此保证。C 和 C++ 标准并不要求这种行为，因此我们在编写可移植代码时当然不能依赖它。（有关此主题的深入讨论，请参阅 https://msdn.microsoft.com/en-us/magazine/dn973015.aspx。）

would produce the following Intel x86 assembly code:

```
mov  eax,[B]
add  eax,1
mov  [A],eax
mov  [B],0
```

The compiler could very well reorder the instructions as follows, without producing any noticeable effect in a single-threaded execution:

```
mov  eax,[B]
mov  [B],0  ;; write to B before A!
add  eax,1
mov  [A],eax
```

If a second thread were waiting for `B` to become zero before reading the value of `A`, it would cease to function correctly if this compiler optimization were to be applied.

Jeff Preshing wrote a great blog post on this topic, available at http://preshing.com/20120625/memory-ordering-at-compile-time/. (This is where the assembly language example above comes from.) I highly recommend all of Jeff's posts on concurrent programming, so do be sure to check them out.

4.9.3.2 Volatile in C/C++ (and Why It Doesn't Help Us)

How can we prevent the compiler from reordering critical sequences of reads and writes? In C and C++, the `volatile` type qualifier guarantees that consecutive reads or writes of a variable cannot be "optimized away" by the compiler, so this sounds like a promising idea. However, it doesn't work reliably for a number of reasons.

The `volatile` qualifier in C/C++ was really designed to make memory-mapped I/O and signal handlers work reliably. As such, the only guarantee it provides is that the contents of a variable marked `volatile` won't be cached in a register—the variable's value will be re-read directly from memory every time it's accessed. Some compilers do guarantee that instructions will not be reordered across a read or write of a `volatile` variable, but not all of them do, and some only provide this guarantee when targeting certain CPUs, or only when a particular command-line option is passed to the compiler. The C and C++ standards don't require this behavior, so we certainly cannot rely on it when writing portable code. (See https://msdn.microsoft.com/en-us/magazine/dn973015.aspx for an in-depth discussion of this topic.)

此外，C/C++ 中的 volatile 关键字无法阻止 CPU 的乱序执行逻辑在运行时重新排序指令。它也无法避免与缓存一致性相关的问题（参见 4.9.3 节）。因此，至少在 C 和 C++ 中， volatile 关键字无法帮助我们编写可靠的并发软件。8

4.9.3.3 编译器屏障

防止编译器跨关键操作边界重新排序读写指令的一个可靠方法是明确指示它不要这样做。这可以通过在代码中插入一条称为“编译器屏障”的特殊伪指令来实现。

不同的编译器使用不同的语法来表达屏障。在 GCC 中，可以通过一些内联汇编语法插入编译器屏障，如下所示；在 Microsoft Visual C++ 中，编译器内部函数 _ReadWrite Barrier() 具有相同的效果。

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // produce some data
    g_data = 42;

    // dear compiler: please don't reorder
    // instructions across this barrier!
    asm volatile("" ::: "memory")

    // inform the consumer
    g_ready = 1;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // dear compiler: please don't reorder
    // instructions across this barrier!
    asm volatile("" ::: "memory")
```

8 在某些语言（包括 Java 和 C#）中， volatile 类型限定符确实保证了原子性，并且可用于实现并发数据结构和算法。有关此主题的更多信息，请参见第 4.9.6 节。

Moreover, the `volatile` keyword in C/C++ does nothing to prevent the CPU's out-of-order execution logic from reordering the instructions at runtime. And it cannot prevent cache coherency related issues either (see Section 4.9.3). So, at least in C and C++, the `volatile` keyword won't help us to write reliable concurrent software.[8]

4.9.3.3 Compiler Barriers

One reliable way of preventing the compiler from reordering read and write instructions across critical operation boundaries is to explicitly instruct it not to do so. This can be accomplished by inserting a special pseudoinstruction into our code known as a *compiler barrier*.

Different compilers express barriers with different syntax. With GCC, a compiler barrier can be inserted via some inline assembly syntax as shown below; under Microsoft Visual C++, the compiler intrinsic `_ReadWrite Barrier()` has the same effect.

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread()
{
    // produce some data
    g_data = 42;

    // dear compiler: please don't reorder
    // instructions across this barrier!
    asm volatile("" ::: "memory")

    // inform the consumer
    g_ready = 1;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // dear compiler: please don't reorder
    // instructions across this barrier!
    asm volatile("" ::: "memory")
```

[8] In some languages including Java and C#, the volatile type qualifier *does* guarantee atomicity, and can be used to implement concurrent data structures and algorithms. See Section 4.9.6 for more on this topic.

```
    // consume the data
    ASSERT(g_data == 42);
}
```

还有其他方法可以防止编译器重新排序指令。例如，大多数函数调用都充当了隐式编译器屏障。这是有道理的，因为编译器对函数调用的副作用一无所知。因此，它无法假设调用前后内存状态相同，这意味着大多数优化在函数调用过程中都是不允许的。一些优化编译器确实会针对内联函数对此规则做出例外处理。

不幸的是，编译器屏障并不能阻止 CPU 的乱序执行逻辑在运行时重新排序指令。一些 ISA 为此提供了一条特殊指令（例如 PowerPC 的 isync 指令）。在 4.9.4.5 节中，我们将学习一组称为内存栅栏的机器语言指令，它们既可以作为编译器的指令重排序屏障，也可以作为 CPU 的指令重排序屏障，更重要的是，它们还可以防止内存重排序错误。因此，原子指令和栅栏是我们编写可靠互斥锁以及自旋锁和其他无锁算法真正需要的全部。

4.9.4 内存排序语义

在 4.9.1 节中，我们提到，除了编译器或 CPU 实际对程序中的机器语言指令进行重排序之外，在并发系统中，读写指令也可能发生有效的重排序。具体来说，在具有多级内存缓存的多核机器中，即使指令实际上是按照我们预期的顺序执行的，两个或多个核心有时也会对读写指令序列的执行顺序产生不一致。显然，这种不一致可能会导致并发软件中出现一些细微的错误。

这些神秘而棘手的问题只会发生在配备多级缓存的多核机器上。单个 CPU 核心始终会按照其执行顺序“看到”自身读写指令的效果；只有当有两个或多个核心时，才会出现不一致的情况。此外，不同的 CPU 具有不同的内存排序行为，这意味着这些

9 函数调用仅当编译器无法“看到”函数定义时（例如，当函数在单独的翻译单元中定义时），才会充当隐式屏障。链接时优化 (LTO) 可以为编译器优化器提供一种查看其原本无法看到的函数定义的方法，从而有效地消除这些隐式屏障，从而避免引入并发错误。


```
    // consume the data
    ASSERT(g_data == 42);
}
```

There are other ways to prevent the compiler from reordering instructions. For example, most function calls serve as an implicit compiler barrier. This makes sense, because the compiler doesn't know anything[9] about the side effects of a function call. As such, it can't assume that the state of memory will be the same before and after the call, which means most optimizations aren't permitted across a function call. Some optimizing compilers do make an exception to this rule for inline functions.

Unfortunately, compiler barriers don't prevent the CPU's out-of-order execution logic from reordering instructions at runtime. Some ISAs provide a special instruction for this purpose (e.g., PowerPC's `isync` instruction). In Section 4.9.4.5, we'll learn about a collection of machine language instructions known as *memory fences* which serve as instruction reordering barriers for both the compiler and the CPU, but more importantly also prevent *memory reordering* bugs. So atomic instructions and fences are all we really need to write reliable mutexes, as well as spin locks and other *lock-free* algorithms.

4.9.4 Memory Ordering Semantics

In Section 4.9.1, we said that in addition to the compiler or CPU actually reordering the machine language instructions in our programs, it's possible for read and write instructions to become *effectively reordered* in a concurrent system. Specifically, in a multicore machine with a multilevel memory cache, two or more cores can sometimes *disagree* about the apparent order in which a sequence of read and write instructions occurred, even when the instructions were *actually* executed in the order we intended. Obviously, such disagreements can cause subtle bugs in concurrent software.

These mysterious and vexing problems can only occur on a multicore machine with a multilevel cache. A single CPU core always "sees" the effects of its own read and write instructions in the order they were executed; only when there are two or more cores can disagreements arise. What's more, different CPUs have different memory ordering behavior, meaning that these

[9] Function calls only serve as implicit barriers when the compiler is unable to "see" the definition of the function, such as when the function is defined in a separate translation unit. Link time optimizations (LTO) can introduce concurrency bugs by providing a way for the compiler's optimizer to see the definitions of functions it otherwise could not have seen, thereby effectively eliminating these implicit barriers.

当运行完全相同的源程序时，不同的机器产生的奇怪效果可能会有所不同！

值得庆幸的是，一切还不算完。每个 CPU 都受一套严格的规则（称为内存排序语义）的约束。这些规则为读写操作如何在核心之间传递提供了各种保证，并且在默认语义不足时，它们为程序员提供了强制执行特定排序所需的工具。

有些 CPU 默认只提供弱保证，而有些 CPU 则提供更强的保证，因此程序员的干预较少。因此，如果我们能够了解如何在内存排序语义最弱的 CPU 上克服内存排序问题，那么我们就可以确信，这些技术在默认语义更强的 CPU 上也能有效。

4.9.4.1 重新审视内存缓存

为了理解这些神秘的内存重新排序效应是如何发生的，我们需要更仔细地研究多级内存缓存的工作原理。

在 3.5.4 节中，我们详细描述了内存缓存如何通过将常用数据保存在缓存中来避免主 RAM 极高的内存访问延迟。这意味着，只要数据对象存在于缓存中，CPU 就会始终尝试使用该缓存副本，而不是访问主 RAM 中的副本。

让我们通过考虑以下简单（且完全人为的）函数来简要回顾一下它的工作原理：

```
constexpr int      COUNT = 16;
alignas(64) float  g_values[COUNT];
float              g_sum = 0.0f;

void CalculateSum()
{
    g_sum = 0.0f;
    for (int i = 0; i < COUNT; ++i)
    {
        g_sum += g_values[i];
    }
}
```

第一条语句将 g_sum 设置为零。假设 g_sum 的内容尚未存在于 L1 缓存中，则包含它的缓存行此时将被读入 L1。同样，在循环的第一次迭代中，包含 g_values 数组所有元素的缓存行将被加载到 L1 中。（假设我们的缓存行至少有 64 个字节宽，它们应该都能加载进去，因为我们进行了对齐。）

strange effects can differ from machine to machine when running the exact same source program!

Thankfully, all is not lost. Every CPU is governed by a strict set of rules known as its *memory ordering semantics*. These rules provide various guarantees about how reads and writes propagate between cores, and they provide programmers with the tools necessary to enforce a particular ordering, when the default semantics are insufficient.

Some CPUs offer only weak guarantees by default, while others provide stronger guarantees and hence require less intervention on the part of the programmer. So, if we can understand how to overcome memory ordering issues on a CPU with the weakest memory ordering semantics, we can be pretty sure those techniques will also work on CPUs with stronger default semantics.

4.9.4.1 Memory Caching Revisited

In order to understand how these mysterious memory reordering effects can occur, we need to look more closely at how a multilevel memory cache works.

In Section 3.5.4, we described in detail how a memory cache avoids the very high memory access latency of main RAM by keeping frequently-used data in the cache. This means that as long as a data object is present in the cache, the CPU will always try to work with that cached copy, rather than reaching out to access the copy in main RAM.

Let's briefly review how this works by considering the following simple (and totally contrived) function:

```
constexpr int      COUNT = 16;
alignas(64) float  g_values[COUNT];
float              g_sum = 0.0f;

void CalculateSum()
{
    g_sum = 0.0f;
    for (int i = 0; i < COUNT; ++i)
    {
        g_sum += g_values[i];
    }
}
```

The first statement sets `g_sum` to zero. Presuming that the contents of `g_sum` aren't already in the L1 cache, the cache line containing it will be read into L1 at this point. Likewise, on the first iteration of the loop, the cache line containing all of the elements of the `g_values` array will be loaded into L1. (They should all fit presuming our cache lines are at least 64 bytes wide, because we aligned

使用 C++11 的 alignas 说明符将数组对齐到 64 字节边界。）后续迭代将读取驻留在 L1 缓存中的 g_values 副本，而不是从主 RAM 读取它们。

在每次迭代过程中，g_sum 都会更新。编译器可能会通过将和保存在寄存器中直到循环结束来优化这一点。但无论是否执行此优化，我们都知道 g_sum 变量将在此函数的某个时刻被写入。当这种情况发生时，CPU 会再次写入 L1 缓存中 g_sum 的副本，而不是直接写入主 RAM。

当然，最终必须更新 g_sum 的"主"副本。内存缓存硬件会通过触发写回操作自动执行此操作，该操作将缓存行从 L1 复制回主 RAM。但写回通常不会立即发生；它通常会推迟到再次读取修改后的变量时进行。10

4.9.4.2 多核缓存一致性协议

在多核机器中，内存缓存会变得更加复杂。图 4.36 展示了一台简单的双核机器，其中每个核心都有自己私有的 L1 缓存，两个核心共享一个 L2 缓存和一个较大的主 RAM 组。为了尽可能简化以下讨论，我们忽略 L2 缓存，将其视为与主 RAM 大致相同。

假设 4.9.3.3 节中所示的简化生产者-消费者示例正在这台双核机器上运行。生产者线程在核心 1 上运行，消费者线程在核心 2 上运行。为了便于讨论，我们进一步假设两个线程中的指令均未发生重排序。

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread() // running on Core 1
{
    g_data = 42;
    // assume no instruction reordering across this line
    g_ready = 1;
}
```

10 某些内存缓存硬件确实允许缓存写入操作立即直写至主内存。为了便于讨论，我们可以放心地忽略直写缓存。

the array to a 64-byte boundary with C++11's `alignas` specifier.) Subsequent iterations will read the copy of `g_values` that resides in the L1 cache, rather than reading them from main RAM.

During each iteration, `g_sum` is updated. The compiler might optimize this by keeping the sum in a register until the end of the loop. But whether or not this optimization is performed, we know that the `g_sum` variable will be written to at some point during this function. When that happens, again the CPU will write to the *copy* of `g_sum` that exists in the L1 cache, rather than writing directly to main RAM.

Eventually, of course, the "master" copy of `g_sum` has to be updated. The memory cache hardware does this automatically by triggering a *write-back* operation that copies the cache line from L1 back to main RAM. But the write-back doesn't usually happen right away; it's typically deferred until the modified variable is read again.[10]

4.9.4.2 Multicore Cache Coherency Protocols

In a multicore machine, memory caching gets a lot more complicated. Figure 4.36 illustrates a simple dual-core machine, in which each core has its own private L1 cache, and the two cores share an L2 cache and a large bank of main RAM. To keep the following discussion as simple as possible, let's ignore the L2 cache and treat it as being roughly equivalent to main RAM.

Suppose that the simplified producer-consumer example shown in Section 4.9.3.3 is running on this dual-core machine. The producer thread runs on Core 1, and the consumer thread runs on Core 2. Let's further assume for the purposes of this discussion that none of the instructions in either thread have been reordered.

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread() // running on Core 1
{
    g_data = 42;
    // assume no instruction reordering across this line
    g_ready = 1;
}
```

[10] Some memory cache hardware does allow cached writes to *write-through* to main RAM immediately. We can safely ignore write-through caches for the purposes of the present discussion.

图 4.36 一台双核机器，每个核心都有一个本地 L1 缓存，通过互连总线 (ICB) 连接到内存控制器。内存控制器实现了缓存一致性协议（例如 MESI），以确保两个核心在 CPU 缓存一致性域内对内存内容具有一致的视图。

```
void ConsumerThread() // running on Core 2
{
    while (!g_ready)
        PAUSE();
    // assume no instruction reordering across this line
    ASSERT(g_data == 42);
}
```

现在考虑一下当生产者（在核心 1 上）写入 g_ready 时会发生什么。为了提高效率，这次写入会导致核心 1 的 L1 缓存更新，但要过一段时间才会触发写回主内存。这意味着在写入发生后的一段有限时间内， g_ready 的最新值除了存在于核心 1 的 L1 缓存中之外，不会存在于任何其他位置。

假设生产者将 g_ready 设置为 1 后，消费者（运行在 Core 2 上）尝试读取该值。与 Core 1 类似，Core 2 也尽可能地从缓存中读取，以避免读取主内存的高昂开销。Core 2 的本地 L1 缓存不包含 g_ready 的副本，但 Core 1 的 L1 缓存包含。因此，理想情况下，Core 2 会向 Core 1 请求获取其副本，因为这仍然比从主内存获取数据开销低得多。而且，在这种特殊情况下，这样做还有一个明显的优势，那就是可以返回最新的值。

缓存一致性协议是一种通信机制，允许核心以这种方式在其本地 L1 缓存之间共享数据。大多数 CPU 使用 MESI 或 MOESI 协议。

Figure 4.36. A dual core machine with a local L1 cache per core, connected to a memory controller via the interconnect bus (ICB). The memory controller implements a cache coherency protocol such as MESI to ensure that both cores have a consistent view of the contents of memory within the CPU's cache coherency domain.

```
void ConsumerThread() // running on Core 2
{
    while (!g_ready)
        PAUSE();
    // assume no instruction reordering across this line
    ASSERT(g_data == 42);
}
```

Now consider what happens when the producer (on Core 1) writes to `g_ready`. In the name of efficiency, this write causes Core 1's L1 cache to be updated, but it won't trigger a *write-back* to main RAM until some time later. This means that for some finite amount of time after the write has occurred, the most up-to-date value of `g_ready` don't exist anywhere but inside Core 1's L1 cache.

Let's say that the consumer (running on Core 2) attempts to read `g_ready` at some time after the producer set it to 1. Like Core 1, Core 2 prefers to read from the cache whenever possible, to avoid the high cost of reading main RAM. Core 2's local L1 cache does not contain a copy of `g_ready`, but Core 1's L1 cache *does*. So ideally Core 2 would like to ask Core 1 for its copy, because that would still be a lot less expensive than getting the data from main RAM. And in this particular case, doing so would also have the distinct advantage of returning the most up-to-date value.

A *cache coherency protocol* is a communication mechanism that permits cores to share data between their local L1 caches in this way. Most CPUs use either the MESI or MOESI protocol.

4.9.4.3 MESI 协议

在 MESI 协议下，每个缓存行可以处于以下四种状态之一：

- 已修改。此缓存行已在本地被修改（写入）。
- 独占。与此缓存行对应的主 RAM 内存块仅存在于此内核的 L1 缓存中 - 其他内核均没有它的副本。
- 共享。与该缓存行对应的主 RAM 内存块存在于多个核心的 L1 缓存中，并且所有核心都有其相同的副本。
- 无效。此缓存行不再包含有效数据——下一次读取需要从另一个核心的 L1 缓存或主 RAM 获取该行。

MOESI 协议添加了另一个名为 O wned 的状态，允许核心共享修改后的数据，而无需先将其写回主 RAM。为了简单起见，我们这里主要讨论 MESI。

在 MESI 协议下，所有核心的 L1 缓存都通过一种称为互连总线 (ICB) 的特殊总线连接。L1 缓存、任何更高级别的缓存以及主 RAM 共同构成了所谓的缓存一致性域。该协议确保所有核心对该域中的数据拥有一致的“视图”。

通过回到我们的例子，我们可以了解 MESI 状态机是如何工作的。

- 假设核心 1（生产者）出于某种原因首先尝试读取 g_ready 的当前值。假设该变量在任何核心的 L1 缓存中均不存在，则包含该变量的缓存行将被加载到核心 1 的 L1 缓存中。该缓存行将被置于 E 独占状态，这意味着没有其他核心拥有该缓存行。
- 现在假设核心 2（消费者）尝试读取 g_ready 。一条 Read 消息通过 ICB 发送。核心 1 拥有此缓存行，因此它会以数据副本作为响应。此时，两个核心上的缓存行均被置为 S 共享状态，表明两个核心都拥有该缓存行的相同副本。
- 接下来，核心 1 上的生产者将 1 写入 g_ready 。这会更新核心 1 的一级缓存中的值，并将其该行副本置于“已修改”状态。一条 Invalidate 消息会通过 ICB 发送，导致核心 2 的该行副本置于“无效”状态。这表明核心 2 的包含 g_ready 的行副本不再是最新的。

4.9.4.3 The MESI Protocol

Under the MESI protocol, each cache line can be in one of four states:

- **M**odified. This cache line has been modified (written to) locally.
- **E**xclusive. The main RAM memory block corresponding to this cache line exists *only* in this core's L1 cache—no other core has a copy of it.
- **S**hared. The main RAM memory block corresponding to this cache line exists in *more than one* core's L1 cache, and all cores have an identical copy of it.
- **I**nvalid. This cache line no longer contains valid data—the next read will need to obtain the line either from another core's L1 cache, or from main RAM.

The MOESI protocol adds another state named **O**wned, which allows cores to share modified data without writing it back to main RAM first. We'll focus on MESI here in the name of simplicity.

Under the MESI protocol, all cores' L1 caches are connected via a special bus called the *interconnect bus* (ICB). Collectively, the L1 caches, any higher-level caches, and main RAM form what is known as a *cache coherency domain*. The protocol ensures that all cores have a consistent "view" of the data in this domain.

We can get a feel for how the MESI state machine works by returning to our example.

- Let's assume that Core 1 (the producer) first tries to read the current value of `g_ready` for some reason. Presuming that this variable doesn't already exist in any core's L1 cache, the cache line that contains it is loaded into Core 1's L1 cache. The cache line is put into the **E**xclusive state, meaning that no other core has this line.
- Now let's say that Core 2 (the consumer) attempts to read `g_ready`. A *Read* message is sent over the ICB. Core 1 has this cache line, so it responds with a copy of the data. At this point, the cache line is put into the **S**hared state on both cores, indicating that both have an identical copy of the line.
- Next, the producer on Core 1 writes a 1 into `g_ready`. This updates the value in Core 1's L1 cache, and puts its copy of the line into the **M**odified state. An *Invalidate* message is sent across the ICB, causing Core 2's copy of the line to be put into the Invalid state. This indicates that Core 2's copy of the line containing `g_ready` is no longer up-to-date.

- 下次核心 2（消费者）尝试读取 g_ready 时，它发现其本地缓存的副本无效。它通过 ICB 发送一条读取消息，并从核心 1 的 L1 缓存中获取新修改的行。这会导致两个核心的缓存行再次进入共享状态。

  它还会触发将行写回主 RAM。

对 MESI 协议的完整讨论超出了我们的范围，但这个例子应该能让您很好地了解它是如何工作的，以允许多个核心在其 L1 缓存之间共享数据，同时最大限度地减少对主 RAM 的访问。

4.9.4.4 MESI 可能出现的问题

根据上一节对 MESI 协议的讨论，多核机器中 L1 缓存之间的数据共享问题似乎已经得到了无懈可击的解决。那么，我们之前提到的内存排序错误究竟是如何发生的呢？

这个问题的答案只有一个词：优化。在大多数硬件上，MESI 协议都经过高度优化，以最大程度地降低延迟。这意味着，某些操作在通过 ICB 接收消息时实际上并不会立即执行。相反，它们会被推迟执行以节省时间。与编译器优化和 CPU 乱序执行优化一样，MESI 优化经过精心设计，以至于单线程无法察觉。但是，正如你所料，并发程序再次遭遇了这种困境。

例如，我们的生产者（运行在核心 1 上）将 42 写入 g_data ，然后立即将 1 写入 g_ready 。在某些情况下，MESI 协议中的优化可能会导致 g_ready 的新值在 g_data 的更新值可见之前对缓存一致性域内的其他核心可见。例如，如果核心 1 的本地一级缓存中已有 g_ready 的缓存行，但尚未有 g_data 的缓存行，则可能会发生这种情况。这意味着消费者（运行在核心 2 上）可能会在看到 g_data 中的值 42 之前先看到 g_ready 的值 1，从而导致数据争用错误。

这种状况可以概括如下：

> 从系统中其他核心的角度来看，缓存一致性协议中的优化可以使两个读取和/或写入指令看起来发生的顺序与指令实际执行的顺序相反。

- The next time that Core 2 (the consumer) tries to read `g_ready`, it finds that its locally-cached copy is **I**nvalid. It sends a *Read* message across the ICB and obtains the newly-modified line from Core 1's L1 cache. This causes both cores' cache lines to be put into the **S**hared state once again. It also triggers a *write-back* of the line to main RAM.

A complete discussion of the MESI protocol is beyond our scope, but this example should give you a good feel for how it works to allow multiple cores to share data between their L1 caches while minimizing accesses to main RAM.

4.9.4.4 How MESI Can Go Wrong

Based on the discussion of the MESI protocol in the preceding section, it would seem that the problem of data sharing between L1 caches in a multicore machine has been solved in a watertight way. How, then, can the memory ordering bugs we've hinted at actually happen?

There's a one-word answer to that question: Optimization. On most hardware, the MESI protocol is highly optimized to minimize latency. This means that some operations aren't actually performed immediately when messages are received over the ICB. Instead, they are deferred to save time. As with compiler optimizations and CPU out-of-order execution optimizations, MESI optimizations are carefully crafted so as to be undetectable by a single thread. But, as you might expect, concurrent programs once again get the raw end of this deal.

For example, our producer (running on Core 1) writes 42 into `g_data` and then immediately writes 1 into `g_ready`. Under certain circumstances, optimizations in the MESI protocol can cause the new value of `g_ready` to become visible to other cores within the cache coherency domain *before* the updated value of `g_data` becomes visible. This can happen, for example, if Core 1 already has `g_ready`'s cache line in its local L1 cache, but does *not* have `g_data`'s line yet. This means that the consumer (on Core 2) can potentially see a value of 1 for `g_ready` *before* it sees a value of 42 in `g_data`, resulting in a data race bug.

This state of affairs can be summarized as follows:

> Optimizations within a cache coherency protocol can make two read and/or write instructions *appear* to happen, from the point of view of other cores in the system, in an order that is opposite to the order in which the instructions were actually executed.

4.9.4.5 记忆栅栏

当两条指令的表观顺序被缓存一致性协议反转时，我们说第一条指令（按程序顺序）已经超过了第二条指令。一条指令超过另一条指令的方式有四种：

1. 一个读可以传递另一个读，
2. 读可以传递写，
3. 一个写入可以传递另一个写入，或者
4. 写入可以通过读取。

为了防止读取或写入指令传递其他读取和/或写入指令的内存效应，现代 CPU 提供了特殊的机器语言指令，称为内存栅栏，也称为内存屏障。

理论上，CPU 可以提供单独的栅栏指令来防止这四种情况的发生。例如，ReadRead 栅栏只会阻止读取操作绕过其他读取操作，而不会阻止任何其他情况。此外，栅栏指令可以是单向的，也可以是双向的。单向栅栏可以保证按程序顺序在它之前的读取或写入操作不会对它之后的操作产生影响，但反之亦然。另一方面，双向栅栏可以防止内存效应在栅栏两侧的任一方向上“泄漏”。因此，理论上，我们可以想象一个 CPU 提供十二条不同的栅栏指令——上面列出的四种基本栅栏类型的双向、正向和反向变体。

值得庆幸的是，实际的 CPU 通常不会提供所有 12 种栅栏指令。相反，ISA 通常会指定一些栅栏指令，这些指令是这些理论栅栏类型的组合。

最强大的栅栏称为完全栅栏。它确保按程序顺序在栅栏之前发生的所有读写操作都不会看起来像在栅栏之后发生；同样，在栅栏之后发生的所有读写操作也不会看起来像在栅栏之前发生。换句话说，完全栅栏是一种双向屏障，既影响读写操作，也影响读写操作。

完全栅栏在硬件上实现起来实际上非常昂贵。CPU 设计人员不喜欢强迫程序员使用昂贵的结构，因为更便宜的结构可以满足需求。因此，大多数 CPU 提供了各种成本较低的栅栏指令，这些指令提供的保证比完全栅栏提供的保证更弱。

所有围栏指令都有两个非常有用的副作用：

1. 它们充当编译器屏障，并且

4.9.4.5 Memory Fences

When the apparent order of two instructions gets reversed by our cache coherency protocol, we say that the first instruction (in program order) has *passed* the second. There are four ways in which one instruction can pass another:

1. A read can pass another read,
2. a read can pass a write,
3. a write can pass another write, or
4. a write can pass a read.

To prevent the memory effects of a read or write instruction passing other reads and/or writes, modern CPUs provide special machine language instructions known as *memory fences*, also known as *memory barriers*.

In theory, a CPU could provide individual fence instructions to prevent each of these four cases from happening. For example, a *ReadRead* fence would only prevent reads from passing other reads, but would not prevent any of the other cases. Also, a fence instruction could be unidirectional or bidirectional. A one-way fence would guarantee that no reads or writes that precede it in program order can end up having an effect after it, but not vice-versa. A bidirectional fence, on the other hand, would prevent “leakage” of memory effects in either direction across the fence. Theoretically, then, we could imagine a CPU that provides twelve distinct fence instructions—a bidirectional, forward, and reverse variant of each of the four basic fence types listed above.

Thankfully real CPUs don’t usually provide all twelve kinds of fences. Instead, an ISA typically specifies a handful of fence instructions which serve as combinations of these theoretical fence types.

The strongest kind of fence is called a *full fence*. It ensures that all reads and writes occurring before the fence in program order will never appear to have occurred after it, and likewise that all reads and writes occurring after it will never appear to have happened before it. In other words, a full fence is a two-way barrier that affects both reads and writes.

A full fence is actually very expensive to realize in hardware. CPU designers don’t like forcing programmers to use an expensive construct when a cheaper one will do, so most CPUs provide a variety of less-expensive fence instructions which provide weaker guarantees than those provided by a full fence.

All fence instructions have two very useful side-effects:

1. They serve as compiler barriers, and

2. 它们阻止 CPU 的无序逻辑跨栅栏重新排序指令。

这意味着，当我们使用栅栏来防止由 CPU 缓存一致性协议引起的内存排序错误时，它也能防止指令重排序。因此，原子指令和内存栅栏是我们编写可靠的互斥锁和自旋锁以及其他无锁算法真正需要的。

4.9.4.6 获取和释放语义

无论特定 ISA 下的特定围栏指令是什么样子，我们都可以通过思考它们提供的语义来推断它们的效果——换句话说，它们对系统中读写行为的保证。

内存排序语义实际上是读写指令的属性，而不是栅栏本身的属性。栅栏仅仅为程序员提供了一种方法来确保读写指令具有特定的内存排序语义。我们通常只需要担心三种内存排序语义：

• 释放语义。此语义保证对共享内存的写入操作永远不会被程序顺序中先于它的任何其他读取或写入操作所超越。当此语义应用于共享写入时，我们称之为写入-释放。此语义仅在正向操作中起作用——它并未提及如何防止在写入-释放之后发生的内存操作看起来像是在写入-释放之前发生的。

• 获取语义。此语义保证从共享内存读取的操作永远不会被程序顺序中在其之后发生的任何其他读取或写入操作所影响。当此语义应用于共享读取时，我们称之为读取-获取。此语义仅在反向操作中起作用——它不会阻止在读取-获取操作之后发生的内存操作产生影响。

• 完整的栅栏语义。这种双向语义确保所有内存操作在跨越代码中栅栏指令创建的边界时，都按程序顺序进行。按程序顺序发生在栅栏之前的任何读写操作，都不会看起来像发生在栅栏之后；同样，按程序顺序发生在栅栏之后的任何读写操作，也不会看起来像发生在栅栏之前。

2. they prevent the CPU's out-of-order logic from reordering instructions across the fence.

This means that when we use a fence to prevent the memory ordering bugs that are caused by our CPU's cache coherency protocol, it also serves to prevent instruction reordering. So atomic instructions and memory fences are all we really need to write reliable mutexes and spin locks, and to write other *lock-free* algorithms as well.

4.9.4.6 Acquire and Release Semantics

No matter what the specific fence instructions look like under a particular ISA, we can reason about their effects by thinking in terms of the *semantics* they provide—in other words, the guarantees they enforce about the behavior of reads and writes in the system.

Memory ordering semantics are really properties of read or write instructions, not properties of the fences themselves. The fences merely provide a way for programmers to ensure that a read or write instruction has a particular memory ordering semantic. There are really only three memory ordering semantics we typically need to worry about:

- *Release* semantics. This semantic guarantees that a *write* to shared memory can never be passed by any other read or write that *precedes* it in program order. When this semantic is applied to a shared write, we call it a *write-release*. This semantic operates in the *forward* direction only—it says nothing about preventing memory operations that occur after the write-release from appearing to happen before it.

- *Acquire* semantics. This semantic guarantees that a *read* from shared memory can never be passed by any other read or write that occurs *after* it in program order. When this semantic is applied to a shared read, we call it a *read-acquire*. This semantic operates in the *reverse* direction only—it does nothing to prevent memory operations that occur before the read-acquire from having their effects seen after it.

- *Full fence* semantics. This bidirectional semantic ensures that all memory operations appear to occur in program order across the boundary created by a fence instruction in the code. No read or write that occurs before the fence in program order can appear to have occurred after the fence, and likewise no read or write that is after the fence in program order can appear to have occurred before it.

任何特定 ISA 提供的单个栅栏指令通常至少提供这三种内存排序语义中的一种。每个栅栏指令如何实际提供这些语义保证的细节与 CPU 息息相关，对于大多数并发程序员来说，我们并不关心。只要我们能够在源代码中表达写-释放、读-获取和完全栅栏的概念，我们就应该能够编写可靠的自旋锁或其他无锁算法。

4.9.4.7 何时使用获取和释放语义

写入-释放最常用于生产者场景——一个线程执行两次连续的写入操作（例如，先写入 g_data，然后写入 g_ready），我们需要确保所有其他线程都能以正确的顺序看到这两次写入操作。我们可以通过将这两次写入操作中的第二次写入操作设置为写入-释放来强制执行此顺序。为了实现这一点，在写入-释放指令之前放置一条提供释放语义的栅栏指令。从技术角度来说，当核心执行具有释放语义的栅栏指令时，它会等到所有先前的写入操作都已完全提交到缓存一致性域内的内存中后，才会执行第二次写入操作（写入-释放）。

读取-获取通常用于消费者场景 - 其中线程执行两次连续读取，其中第二次读取取决于第一次读取（例如，仅在读取标志 g_ready 返回 true 后才读取 g_data）。我们通过确保第一次读取是读取-获取来强制执行此顺序。为了实现这一点，在读取-获取指令之后放置一条提供获取语义的栅栏指令。从技术上讲，当一个核心执行具有获取语义的栅栏指令时，它会等到来自其他核心的所有写入都已完全刷新到缓存一致性域中后，再继续执行第二次读取。这确保了第二次读取永远不会出现在读取-获取之前。

这里再次展示我们的生产者-消费者示例，完全无锁，使用获取和释放栅栏来强制执行必要的内存排序语义：

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread() // running on Core 1
{
    g_data = 42;

    // make the write to g_ready into a write-release
    // by placing a release fence *before* it
    RELEASE_FENCE();
```


The individual fence instructions provided by any particular ISA typically provide at least one of these three memory ordering semantics. The details of how each fence instruction actually provides these semantic guarantees are CPU-specific, and for the most part as concurrent programmers, we don't care. As long as we can express the concept of write-release, read-acquire and full fence in our source code, we should be able to write a reliable spin lock or code other lock-free algorithms.

4.9.4.7 When to Use Acquire and Release Semantics

A *write-release* is most often used in a producer scenario—in which a thread performs two consecutive writes (e.g., writing to `g_data` and then `g_ready`), and we need to ensure that all other threads will see the two writes in the correct order. We can enforce this ordering by making the second of these two writes a write-release. To implement this, a fence instruction that provides *release semantics* is placed *before* the write-release instruction. Technically speaking, when a core executes a fence instruction with release semantics, it waits until all prior writes have been fully committed to memory within the cache coherency domain before executing the second write (the write-release).

A *read-acquire* is typically used in a consumer scenario—in which a thread performs two consecutive reads in which the second is conditional on the first (e.g., only reading `g_data` after a read of the flag `g_ready` comes back `true`). We enforce this ordering by making sure that the first read is a read-acquire. To implement this, a fence instruction that provides *acquire semantics* is placed *after* the read-acquire instruction. Technically speaking, when a core executes a fence instruction with acquire semantics, it waits until all writes from other cores have been fully flushed into the cache coherency domain before it continues on to execute the second read. This ensures that the second read will never appear to have occurred before the read-acquire.

Here's our producer-consumer example again, in full lock-free glory, using acquire and release fences to impose the necessary memory ordering semantics:

```
int32_t  g_data = 0;
int32_t  g_ready = 0;

void ProducerThread() // running on Core 1
{
    g_data = 42;

    // make the write to g_ready into a write-release
    // by placing a release fence *before* it
    RELEASE_FENCE();
```

```
    g_ready = 1;
}

void ConsumerThread() // running on Core 2
{
    // make the read of g_ready into a read-acquire
    // by placing an acquire fence *after* it
    while (!g_ready)
        PAUSE();
    ACQUIRE_FENCE();

    // we can now read g_data safely...
    ASSERT(g_data == 42);
}
```

有关 MESI 缓存一致性协议下为何需要获取和释放围栏的详细介绍，请参阅 http://www.swedishcoding.com/2017/11/10/multi-core-programming-and-cachecoherency/。

4.9.4.8 CPU内存模型

我们在 4.9.4 节中提到，某些 CPU 默认提供比其他 CPU 更强的内存顺序语义。在具有强内存语义的 CPU 上，读取和/或写入指令默认的行为类似于某种栅栏，无需程序员显式指定栅栏指令。例如，DEC Alpha 默认的语义非常弱，几乎在所有情况下都需要谨慎地进行栅栏操作。另一方面，Intel x86 CPU 默认具有相当强的内存顺序语义。有关弱内存顺序和强内存顺序的详细讨论，请参阅 http://preshing。

com/20120930/弱与强记忆模型/。

4.9.4.9 真实 CPU 上的隔离指令

现在我们了解了内存排序语义背后的理论，让我们简要地看一下一些真实 CPU 上的内存栅栏指令。

Intel x86 ISA 指定了三种栅栏指令：sfence 提供释放语义，lfence 提供获取语义，mfence 充当完整栅栏。某些 x86 指令还可以以 lock 修饰符作为前缀，使其行为具有原子性，并在指令执行之前提供内存栅栏。x86 ISA 默认是强排序的，这意味着在很多情况下，栅栏实际上并不需要，而默认排序语义较弱的 CPU 则需要。但在某些情况下，这些栅栏指令是必需的。例如，请参阅帖子


```
    g_ready = 1;
}

void ConsumerThread() // running on Core 2
{
    // make the read of g_ready into a read-acquire
    // by placing an acquire fence *after* it
    while (!g_ready)
        PAUSE();
    ACQUIRE_FENCE();

    // we can now read g_data safely...
    ASSERT(g_data == 42);
}
```

For an excellent in-depth presentation of exactly why acquire and release fences are required under the MESI cache coherency protocol, see http://www.swedishcoding.com/2017/11/10/multi-core-programming-and-cache-coherency/.

4.9.4.8 CPU Memory Models

We mentioned in Section 4.9.4 that some CPUs provide stronger memory order semantics by default than others. On a CPU with strong memory semantics, read and/or write instructions behave like a certain kind of fence by default, without the programmer having to specify a fence instruction explicitly. For example, the DEC Alpha has notoriously weak semantics by default, requiring careful fencing in almost all situations. At the other end of the spectrum, an Intel x86 CPU has quite strong memory ordering semantics by default. For a great discussion of weak and strong memory ordering, see http://preshing.com/20120930/weak-vs-strong-memory-models/.

4.9.4.9 Fence Instructions on Real CPUs

Now that we understand the theory behind memory ordering semantics, let's take a very brief look at memory fence instructions on some real CPUs.

The Intel x86 ISA specifies three fence instructions: `sfence` provides release semantics, `lfence` provides acquire semantics, and `mfence` acts as a full fence. Certain x86 instructions may also be prefixed by a `lock` modifier to make them behave atomically, and to provide a memory fence prior to execution of the instruction. The x86 ISA is strongly ordered by default, meaning that fences aren't actually required in many cases where they would be on CPUs with weaker default ordering semantics. But there are some cases in which these fence instructions *are* required. For an example, see the post

题为“谁在 x86 上订购了内存栅栏？”，作者是 Bartosz Milewski（https://bit.ly/2HuXpfo）。

PowerPC ISA 的排序相当弱，因此通常需要显式的栅栏指令来确保正确的语义。PowerPC 区分对内存的读写和对 I/O 设备的读写，因此它提供了各种栅栏指令，这些指令主要在处理内存和 I/O 的方式上有所不同。PowerPC 上的完整栅栏由 sync 指令提供，但还有一个名为 lwsync 的“轻量级”栅栏，一个名为 eieio 的 I/O 操作栅栏（确保按顺序执行 I/O），甚至还有一个纯指令重新排序屏障 isync，它不提供任何内存排序语义。您可以在此处阅读有关 PowerPC 栅栏指令的更多信息：https://www.ibm.com/developerworks/systems/articles/powerpc.html。

ARM ISA 提供了一个名为 isb 的纯指令重排序屏障、两个全内存栅栏指令 dmb 和 dsb、一个单向读取-获取指令 ldar 以及一个单向写入-释放指令 stlr。有趣的是，该 ISA 将获取和释放语义嵌入到读取和写入指令本身中，而不是作为单独的栅栏指令。更多信息，请参阅 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc。

den0024a/CJAIAJFI.html。

4.9.5 原子变量

直接使用原子指令和内存栅栏可能非常繁琐且容易出错，更不用说完全不可移植了。值得庆幸的是，从 C++11 开始，类模板 std::atomic<T> 允许将几乎任何数据类型转换为原子变量。（一个名为 std::atomic_flag 的特化类封装了一个原子布尔变量。）除了原子性之外，std::atomic 系列模板还默认为其变量提供“完全栅栏”内存排序语义（尽管可以根据需要指定更弱的语义）。这使我们能够编写无锁代码，而无需担心数据竞争错误的三个原因。

使用这些设施，我们的生产者-消费者示例可以写如下：

```
std::atomic<float> g_data;
std::atomic_flag g_ready = false;

void ProducerThread()
{
    // produce some data
    g_data = 42;
```


entitled, “Who ordered memory fences on an x86?” by Bartosz Milewski (https://bit.ly/2HuXpfo).

The PowerPC ISA is quite weakly ordered, so explicit fence instructions are usually required to ensure correct semantics. The PowerPC makes a distinction between reads and writes to memory versus reads and writes to I/O devices, and hence it offers a variety of fence instructions that differ primarily in how they handle memory versus I/O. A full fence on PowerPC is provided by the `sync` instruction, but there's also a “lightweight” fence called `lwsync`, a fence for I/O operations called `eieio` (ensure in-order execution of I/O), and even a pure instruction reordering barrier `isync` that does not provide any memory ordering semantics. You can read more about the PowerPC's fence instructions here: https://www.ibm.com/developerworks/systems/articles/powerpc.html.

The ARM ISA provides a pure instruction reordering barrier called `isb`, two full memory fence instructions `dmb` and `dsb`, a one-way read-acquire instruction `ldar` and a one-way write-release instruction `stlr`. Interestingly, this ISA rolls acquire and release semantics into the read and write instructions themselves, rather than as separate fence instructions. For more information, see http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CJAIAJFI.html.

4.9.5 Atomic Variables

Using atomic instructions and memory fences directly can be tedious and error-prone, not to mention being totally non-portable. Thankfully, as of C++11, the class template `std::atomic<T>` allows virtually any data type to be converted into an *atomic variable*. (A specialized class called `std::atomic_flag` encapsulates an atomic Boolean variable.) In addition to atomicity, the `std::atomic` family of templates provides its variables with “full fence” memory ordering semantics by default (although weaker semantics can be specified if desired). This frees us to write lock-free code without having to worry about any of the three causes of data race bugs.

Using these facilities, our producer-consumer example can be written as follows:

```
std::atomic<float> g_data;
std::atomic_flag g_ready = false;

void ProducerThread()
{
    // produce some data
    g_data = 42;
```

```
    // inform the consumer
    g_ready = true;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // consume the data
    ASSERT(g_data == 42);
}
```

请注意，这段代码与我们在 4.5.3 节讨论竞争条件时首次提出的错误代码几乎完全相同。只需将变量包装在 std::atomic 中，我们就将一个容易出现数据竞争错误的并发程序转换为一个无竞争错误的程序。

在底层，std::atomic<T> 和 std:: 的实现 atomic_flag 的实现当然很复杂。标准 C++ 库必须具备可移植性，因此其实现会利用目标平台上恰好可用的任何原子和屏障机器语言指令。此外，std::atomic<T> 模板可以包装任何可以想到的类型，但 CPU 当然不提供用于操作任意大小数据对象的原子指令。因此，std::atomic<T> 模板必须根据大小进行特化：当模板应用于 32 位或 64 位类型时，它可以直接使用原子机器语言指令来实现，无需锁。但当它应用于更大的类型时，会使用互斥锁来确保正确的原子行为。（您可以对任何原子变量调用 is_lock_free() 来检查其实现在目标硬件上是否真正实现了无锁。）

4.9.5.1 C++ 内存顺序

默认情况下，C++ 原子变量使用完整的内存屏障，以确保它们在所有可能的用例中都能正常工作。但是，可以通过将内存顺序语义（类型为 std::memory_order 的可选参数）传递给操作原子变量的函数来放松这些保证。关于 std::memory_order 的文档相当混乱，所以让我们来揭秘一下。以下是可能的内存顺序设置及其含义：

1. 宽松。以宽松内存顺序执行的原子操作


```
    // inform the consumer
    g_ready = true;
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready)
        PAUSE();

    // consume the data
    ASSERT(g_data == 42);
}
```

Notice that this code looks nearly identical to the erroneous code we first presented when talking about race conditions in Section 4.5.3. By simply wrapping our variables in `std::atomic`, we've converted a concurrent program that is prone to data race bugs into one that is race-free.

Under the hood, the implementations of `std::atomic<T>` and `std::atomic_flag` are, of course, complex. The standard C++ library has to be portable, so its implementation makes use of whichever atomic and barrier machine language instructions happen to be available on the target platform. What's more, the `std::atomic<T>` template can wrap any type imaginable, but of course CPUs don't provide atomic instructions for manipulating arbitrarily-sized data objects. As such, the `std::atomic<T>` template must be specialized by size: When the template is applied to a 32- or 64-bit type, it can be implemented without locks, using atomic machine language instructions directly. But when it is applied to larger types, mutex locks are used to ensure correct atomic behavior. (You can call `is_lock_free()` on any atomic variable to find out whether its implementation really is lock-free on your target hardware.)

4.9.5.1 C++ Memory Order

By default, C++ atomic variables make use of full memory barriers, to ensure that they will work correctly in all possible use cases. However, it is possible to *relax* these guarantees, by passing a *memory order semantic* (an optional argument of type `std::memory_order`) to functions that manipulate atomic variables. The documentation regarding `std::memory_order` is pretty confusing, so let's demystify it. Here are the possible memory order settings, and what they mean:

1. *Relaxed*. An atomic operation performed with relaxed memory order

语义仅保证原子性。不使用屏障或栅栏。

2. 消耗。使用消耗语义执行的读取操作可确保同一线程内的其他读取或写入操作在此读取操作之前不会被重新排序。换句话说，此语义仅用于防止编译器优化和乱序执行对指令进行重新排序——它不会确保缓存一致性域内任何特定的内存排序语义。它通常通过指令重新排序屏障（例如 PowerPC 的 isync 指令）来实现。

3. 释放。使用释放语义执行的写入操作可确保此线程中的其他读取或写入操作不会在其后重新排序，并且保证该写入操作对读取同一地址的其他线程可见。它使用 CPU 缓存一致性域中的释放栅栏来实现这一点。

4. 获取。使用获取语义执行的读取操作不仅保证了消费语义，还保证了其他线程对同一地址的写入操作对该线程可见。它通过 CPU 缓存一致性域中的获取栅栏来实现这一点。

5. 获取/释放。此语义（默认）是最安全的，因为它应用了完整的内存防护。

重要的是要意识到，使用内存排序说明符并不能保证特定语义在所有平台上都能实际应用。它的作用只是保证语义至少达到一定强度——某些目标硬件可能会采用更强的语义。例如，在 Intel x86 上，由于 CPU 的默认内存排序语义相对较强，因此无法实现宽松的内存排序。在 Intel CPU 上，任何宽松的读取操作请求最终都会具有获取语义。

使用这些内存顺序说明符需要从 std:: 切换 atomic 的重载赋值和强制类型转换运算符显式调用了 store() 和 load() 。下面再次展示我们简单的生产者-消费者示例，这次使用 std::memory_order 说明符来提供释放和获取屏障：

```
std::atomic<float> g_data;
std::atomic<bool> g_ready = false;

void ProducerThread()
{
    // produce some data
```


semantics guarantees *only* atomicity. No barrier or fence is used.

2. *Consume*. A read performed with consume semantics guarantees that no other read or write *within the same thread* can be reordered before this read. In other words, this semantic only serves to prevent *compiler optimizations* and *out-of-order execution* from reordering the instructions—it does nothing to ensure any particular memory ordering semantics within the cache coherency domain. It is typically implemented with an instruction reordering barrier like the PowerPC's `isync` instruction.

3. *Release*. A write performed with *release* semantics guarantees that no other read or write in this thread can be reordered after it, and the write is guaranteed to be visible by other threads reading the same address. It employs a *release fence* in the CPU's cache coherency domain to accomplish this.

4. *Acquire*. A read performed with *acquire* semantics guarantees *consume* semantics, and in addition it guarantees that writes to the same address by other threads will be visible to this thread. It does this via an *acquire fence* in the CPU's cache coherency domain.

5. *Acquire/Release*. This semantic (the default) is the safest, because it applies a *full* memory fence.

It's important to realize that using a memory ordering specifier doesn't guarantee that a particular semantic will actually be used on every platform. All it does is to guarantee that the semantics will be *at least* that strong—a stronger semantic might be employed on some target hardware. For example, on Intel x86, the relaxed memory order isn't possible because the CPU's default memory ordering semantics are relatively strong. On an Intel CPU, any request for a relaxed read operation will actually end up having *acquire* semantics.

Using these memory order specifiers requires switching from `std::atomic`'s overloaded assignment and cast operators to explicit calls to `store()` and `load()`. Here's our simple producer-consumer example again, this time using `std::memory_order` specifiers to provide our release and acquire barriers:

```
std::atomic<float> g_data;
std::atomic<bool> g_ready = false;

void ProducerThread()
{
    // produce some data
```

```
    g_data.store(42, std::memory_order_relaxed);

    // inform the consumer
    g_ready.store(true, std::memory_order_release);
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready.load(std::memory_order_acquire))
        PAUSE();

    // consume the data
    ASSERT(g_data.load(std::memory_order_relaxed) == 42);
}
```

在使用像这样的“宽松”内存排序语义时，务必牢记 80/20 规则。这些语义很容易出错，因此，只有当您可以通过性能分析证明性能提升确实有必要，并且将代码更改为使用显式内存排序语义确实能带来预期的收益时，才应该在 std::atomic 中使用非默认内存排序！

关于如何在 C++11 中使用内存排序语义的完整讨论超出了我们的讨论范围，但您可以在线搜索 Michael Chynoweth 撰写的文章“为多核 Intel® EM64T 和 IA32 架构实现可扩展原子锁”，了解更多信息。这个论坛讨论也提供了一些有趣的见解，并说明了这种编程方式的复杂性：https://groups.google.com/forum/#!topic/boost-developers-archive/Qlrat5ASrnM。

4.9.6 解释型编程语言中的并发

到目前为止，我们仅在编译语言（例如 C 和 C++）和汇编语言的背景下讨论了并发。这些语言会编译或汇编为可由 CPU 直接执行的原始机器码。因此，原子操作和锁必须借助提供原子操作和缓存一致性内存屏障的特殊机器语言指令来实现，此外还需要内核（确保线程被适当地休眠和唤醒）和编译器（在优化代码时会考虑屏障指令）的帮助。

对于像 Java 和 C# 这样的解释型编程语言来说，情况有所不同。用这些语言编写的程序在虚拟机 (VM) 的上下文中执行：Java 程序在 Java 虚拟机中运行


```
    g_data.store(42, std::memory_order_relaxed);

    // inform the consumer
    g_ready.store(true, std::memory_order_release);
}

void ConsumerThread()
{
    // wait for the data to be ready
    while (!g_ready.load(std::memory_order_acquire))
        PAUSE();

    // consume the data
    ASSERT(g_data.load(std::memory_order_relaxed) == 42);
}
```


It's important to remember the 80/20 rule when using "relaxed" memory ordering semantics like this. These semantics are easy to get wrong, so you should probably only use non-default memory ordering with `std::atomic` when you can prove, via profiling, that the performance improvement is really necessary—and that changing your code to use explicit memory ordering semantics does, in fact, produce the benefits you expect!

A complete discussion of how to use memory ordering semantics in C++11 is beyond our scope, but you can read more by searching online for the article entitled, "Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures" by Michael Chynoweth. This forum discussion offers some interesting insights as well, and illustrates just how complex this kind of programming can get: https://groups.google.com/forum/#!topic/boost-developers-archive/Qlrat5ASrnM.

4.9.6 Concurrency in Interpreted Programming Languages

Thus far we've only discussed concurrency in the context of compiled languages like C and C++, and in assembly language. These languages compile or assemble down to raw machine code that is executed directly by the CPU. As such, atomic operations and locks must be implemented with the aid of special machine language instructions that provide atomic operations and cache coherent memory barriers, with additional help from the kernel (which makes sure threads are put to sleep and awoken appropriately) and the compiler (which respects barrier instructions when optimizing the code).

The story is somewhat different for interpreted programming languages like Java and C#. Programs written in these languages execute in the context of a *virtual machine* (VM): Java programs run inside the Java Virtual Machine

(JVM) 和 C# 程序在公共语言运行时 (CLR) 上下文中运行。虚拟机本质上是 CPU 的软件模拟，逐条读取字节码指令并执行。虚拟机的行为也有点像模拟的操作系统内核：它提供自己的“线程”概念（由字节码指令组成），并自行处理调度这些线程的所有细节。由于虚拟机的操作完全由软件实现，因此像 Java 或 C# 这样的解释型语言可以提供强大的并发同步功能，而且这些功能不像 C 或 C++ 这样的编译型语言那样受硬件限制。

此原则的一个实际应用示例是 volatile 类型限定符。我们在 4.6 节中提到，在 C/C++ 中， volatile 变量不是原子的。然而，在 Java 和 C# 中， volatile 类型限定符确实保证了原子性。在这些语言中，对 volatile 变量的操作无法优化，也无法被其他线程中断。此外，在 Java 和 C# 中，所有对 volatile 变量的读取操作实际上都是直接从主内存而不是缓存执行的，同样，所有写入操作实际上都是写入主 RAM 而不是缓存。所有这些保证之所以能够提供，部分原因是虚拟机完全控制着构成每个应用程序的字节码指令流的执行。

本书不会全面讨论 C# 和 Java 等解释型语言提供的原子性和线程同步功能。不过，既然你已经对原子性背后的底层原理有了扎实的理解，那么理解其他高级语言中的这些功能应该轻而易举。以下网站是进一步阅读的良好开端：

- **C#：在 https://docs.microsoft.com 上搜索“.NET Framework 中的并行处理和并发”。**
- Java：https://docs.oracle.com/javase/tutorial/essential/concurrency/。

4.9.7 自旋锁

在 4.9.2.2 节讨论原子机器语言指令时，我们提供了一些代码片段，演示了如何使用这些指令实现自旋锁。但由于指令重排序和内存排序语义的原因，这些示例并非 100% 正确。在本节中，我们将介绍一个工业级的自旋锁，并探索一些有用的变体。

(JVM), and C# programs run within the context of the Common Language Runtime (CLR). A virtual machine is essentially a software emulation of a CPU, reading bytecoded instructions one by one and executing them. A VM also acts a bit like an emulated operating system kernel: It provides its own notion of “threads” (comprised of bytecoded instructions), and it handles all of the details of scheduling those threads itself. Because the operation of a VM is implemented entirely in software, an interpreted language like Java or C# can provide powerful concurrent synchronization facilities that are not as constrained by the hardware as they would be in a compiled language like C or C++.

One example of this principle in action is the `volatile` type qualifier. We said in Section 4.6 that in C/C++ a `volatile` variable is not atomic. However, in Java and C#, the `volatile` type qualifier *does* guarantee atomicity. Operations on volatile variables in these languages cannot be optimized, and they cannot be interrupted by another thread. Moreover, all reads of a volatile variable in Java and C# are effectively performed directly from main memory rather than from the cache, and likewise all writes are effectively written to main RAM rather than to the cache. All of these guarantees can be provided in part because the virtual machine has full control over the execution of the bytecoded instruction streams that comprise each app.

A full discussion of the atomicity and thread synchronization facilities provided by interpreted languages like C# and Java is well beyond the scope of this book. But now that you have a solid understanding of the principles behind atomicity at the lowest levels, understanding these facilities in other higher-level languages should be a snap. For further reading, the following websites are a good start:

- C#: Search for “Parallel Processing and Concurrency in the .NET Framework” on https://docs.microsoft.com.
- Java: https://docs.oracle.com/javase/tutorial/essential/concurrency/.

4.9.7 Spin Locks

When discussing atomic machine language instructions in Section 4.9.2.2, we presented some code snippets illustrating how a spin lock might be implemented using each of those instructions. But because of instruction reordering and memory ordering semantics, those examples weren’t 100% correct. In this section, we’ll present an industrial-strength spin lock, and then explore a few useful variants.

4.9.7.1 基本自旋锁

自旋锁可以使用 std::atomic_flag 实现，可以封装在 C++ 类中，也可以通过简单的函数式 API 访问。自旋锁的获取方式是使用 TAS 指令原子地将标志设置为 true ，并在 while 循环中重试直到 TAS 成功。解锁方式是原子地将 false 写入标志。

获取自旋锁时，务必使用读取-获取内存顺序语义来读取锁的当前内容，这是 TAS 操作的一部分。此栅栏可以防止出现罕见的情况：当其他线程尚未完全退出其临界区时，锁被观察到已释放。在 C++11 中，可以通过将 std::memory_order_acquire 传递给 test_and_set() 调用来实现。在原始汇编语言中，我们会在 TAS 指令之后放置一条获取栅栏指令。

当释放自旋锁时，同样重要的是使用写入释放语义来确保在调用 Unlock() 之后执行的所有写入都不会被其他线程观察到，就好像它们发生在锁被释放之前一样。

以下是基于 TAS 的自旋锁的完整实现，使用正确且最小的内存排序语义：

```
class SpinLock
{
    std::atomic_flag  m_atomic;

public:
    SpinLock() : m_atomic(false) { }

    bool TryAcquire()
    {
        // use an acquire fence to ensure all subsequent
        // reads by this thread will be valid
        bool alreadyLocked = m_atomic.test_and_set(
                             std::memory_order_acquire);

        return !alreadyLocked;
    }

    void Acquire()
    {
        // spin until successful acquire
        while (!TryAcquire())
        {
```


4.9.7.1 Basic Spin Lock

A spin lock can be implemented using a `std::atomic_flag`, either wrapped in a C++ class or accessed via a simple functional API. The spin lock is acquired by using a TAS instruction to atomically set the flag to `true`, retrying in a while loop until the TAS succeeds. It is unlocked by atomically writing a value of `false` into the flag.

When acquiring a spin lock, it's important to use *read-acquire* memory ordering semantics to read the lock's current contents as part of the TAS operation. This fence guards against the rare scenario in which the lock is observed as having been released when in reality some other thread hasn't *completely* exited its critical section yet. In C++11, this can be accomplished by passing `std::memory_order_acquire` to the `test_and_set()` call. In raw assembly language, we'd place an acquire fence instruction after the TAS instruction.

When releasing a spin lock, it's likewise important to use *write-release* semantics to ensure that all writes performed *after* the call to `Unlock()` aren't observed by other threads as if they had happened *before* the lock was released.

Here's a complete implementation of a TAS-based spin lock, using correct and minimal memory ordering semantics:

```
class SpinLock
{
    std::atomic_flag  m_atomic;

public:
    SpinLock() : m_atomic(false) { }

    bool TryAcquire()
    {
        // use an acquire fence to ensure all subsequent
        // reads by this thread will be valid
        bool alreadyLocked = m_atomic.test_and_set(
                             std::memory_order_acquire);

        return !alreadyLocked;
    }

    void Acquire()
    {
        // spin until successful acquire
        while (!TryAcquire())
        {
```

```
            // reduce power consumption on Intel CPUs
            // (can substitute with std::this_thread::yield()
            // on platforms that don't support CPU pause, if
            // thread contention is expected to be high)
            PAUSE();
        }
    }

    void Release()
    {
        // use release semantics to ensure that all prior
        // writes have been fully committed before we unlock
        m_atomic.clear(std::memory_order_release);
    }
};
```

4.9.7.2 作用域锁

手动解锁互斥锁或自旋锁通常很不方便，而且容易出错，尤其是在使用锁的函数有多个返回点的情况下。在 C++ 中，我们可以使用一个名为“作用域锁”的简单包装类来确保在退出特定作用域时自动释放锁。它的工作原理很简单，只需在构造函数中获取锁，然后在析构函数中释放它即可。

```
template<class LOCK>
class ScopedLock
{
    typedef LOCK lock_t;
    lock_t* m_pLock;

public:
    explicit ScopedLock(lock_t& lock) : m_pLock(&lock)
    {
        m_pLock->Acquire();
    }

    ~ScopedLock()
    {
        m_pLock->Release();
    }
};
```

此作用域锁类可与任何具有一致接口的自旋锁或互斥锁一起使用（即任何支持 Acquire() 和 Release() 函数的锁类）。使用方法如下：


```
            // reduce power consumption on Intel CPUs
            // (can substitute with std::this_thread::yield()
            // on platforms that don't support CPU pause, if
            // thread contention is expected to be high)
            PAUSE();
        }
    }

    void Release()
    {
        // use release semantics to ensure that all prior
        // writes have been fully committed before we unlock
        m_atomic.clear(std::memory_order_release);
    }
};
```

4.9.7.2 Scoped Locks

It's often inconvenient and error-prone to manually unlock a mutex or spin lock, especially when the function using the lock has multiple return sites. In C++ we can use a simple wrapper class called a *scoped lock* to ensure that a lock is automatically released when a particular scope is exited. It works by simply acquiring the lock in the constructor, and releasing it in the destructor.

```
template<class LOCK>
class ScopedLock
{
    typedef LOCK lock_t;
    lock_t* m_pLock;

public:
    explicit ScopedLock(lock_t& lock) : m_pLock(&lock)
    {
        m_pLock->Acquire();
    }

    ~ScopedLock()
    {
        m_pLock->Release();
    }
};
```

This scoped lock class can be used with any spin lock or mutex that has a conformant interface (i.e., any lock class that supports the functions `Acquire()` and `Release()`). Here's how it might be used:

```
SpinLock g_lock;

int ThreadSafeFunction()
{
    // the scoped lock acts like a "janitor"
    // because it cleans up for us!
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...

    if (SomethingWentWrong())
    {
        // lock will be released here
        return -1;
    }

    // so some more work...

    // lock will also be released here
    return 0;
}
```

4.9.7.3 可重入锁

如果线程尝试重新获取已持有的锁，原生自旋锁将导致线程死锁。当两个或多个线程安全函数尝试在同一个线程内以可重入的方式相互调用时，就会发生这种情况。

例如，给定两个函数：

```
SpinLock g_lock;

void A()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...
}

void B()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...

    // make a call to A() while holding the lock
    A();  // deadlock!
```



```
SpinLock g_lock;

int ThreadSafeFunction()
{
    // the scoped lock acts like a "janitor"
    // because it cleans up for us!
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...

    if (SomethingWentWrong())
    {
        // lock will be released here
        return -1;
    }

    // so some more work...

    // lock will also be released here
    return 0;
}
```

4.9.7.3 Reentrant Locks

A vanilla spin lock will cause a thread to deadlock if it ever tries to reacquire a lock that it already holds. This can happen whenever two or more *thread-safe* functions attempt to call one another reentrantly from within the same thread. For example, given two functions:

```
SpinLock g_lock;

void A()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...
}

void B()
{
    ScopedLock<decltype(g_lock)> janitor(g_lock);

    // do some work...

    // make a call to A() while holding the lock
    A();  // deadlock!
```

```
    // do some more work...
}
```

如果我们能让自旋锁类缓存锁定它的线程 ID，就能放宽这个可重入限制。这样，自旋锁就能“知道”两个情况：一个线程试图重新获取自己的锁（我们希望允许这种情况发生）；另一个线程试图获取已被其他线程持有的锁（这会导致调用者等待）。为了确保 Acquire() 和 Release() 的调用成对出现，我们还需要为自旋锁类添加一个引用计数。以下是一个使用适当内存隔离机制的函数式实现：

```
class ReentrantLock32
{
    std::atomic<std::size_t>  m_atomic;
    std::int32_t              m_refCount;

public:
    ReentrantLock32() : m_atomic(0), m_refCount(0) { }

    void Acquire()
    {
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());

        // if this thread doesn't already hold the lock...
        if (m_atomic.load(std::memory_order_relaxed) != tid)
        {
            // ... spin wait until we do hold it
            std::size_t unlockValue = 0;
            while (!m_atomic.compare_exchange_weak(
                unlockValue,
                tid,
                std::memory_order_relaxed, // fence below!
                std::memory_order_relaxed))
            {
                unlockValue = 0;
                PAUSE();
            }
        }

        // increment reference count so we can verify that
        // Acquire() and Release() are called in pairs
        ++m_refCount;
```



```
    // do some more work...
}
```

We can relax this reentrancy restriction if we can arrange for our spin lock class to cache the id of the thread that has locked it. That way, the lock can “know” the difference between a thread trying to reacquire its own lock (which we wish to allow) versus a thread trying to grab a lock that’s already held by a different thread (which should cause the caller to wait). To make sure that calls to `Acquire()` and `Release()` are done in matching pairs, we’ll also want to add a reference count to the class. Here’s a functional implementation, using appropriate memory fencing:

```
class ReentrantLock32
{
    std::atomic<std::size_t>  m_atomic;
    std::int32_t              m_refCount;

public:
    ReentrantLock32() : m_atomic(0), m_refCount(0) { }

    void Acquire()
    {
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());

        // if this thread doesn't already hold the lock...
        if (m_atomic.load(std::memory_order_relaxed) != tid)
        {
            // ... spin wait until we do hold it
            std::size_t unlockValue = 0;
            while (!m_atomic.compare_exchange_weak(
                unlockValue,
                tid,
                std::memory_order_relaxed, // fence below!
                std::memory_order_relaxed))
            {
                unlockValue = 0;
                PAUSE();
            }
        }

        // increment reference count so we can verify that
        // Acquire() and Release() are called in pairs
        ++m_refCount;
```

```
        // use an acquire fence to ensure all subsequent
        // reads by this thread will be valid
        std::atomic_thread_fence(std::memory_order_acquire);
    }

    void Release()
    {
        // use release semantics to ensure that all prior
        // writes have been fully committed before we unlock
        std::atomic_thread_fence(std::memory_order_release);

        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());
        std::size_t actual = m_atomic.load(std::memory_order_relaxed);
        assert(actual == tid);

        --m_refCount;
        if (m_refCount == 0)
        {
            // release lock, which is safe because we own it
            m_atomic.store(0, std::memory_order_relaxed);
        }
    }

    bool TryAcquire()
    {
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());

        bool acquired = false;

        if (m_atomic.load(std::memory_order_relaxed) == tid)
        {
            acquired = true;
        }
        else
        {
            std::size_t unlockValue = 0;
            acquired = m_atomic.compare_exchange_strong(
                unlockValue,
                tid,
                std::memory_order_relaxed, // fence below!
                std::memory_order_relaxed);
        }
        if (acquired)
        {
            ++m_refCount;
```



```
        // use an acquire fence to ensure all subsequent
        // reads by this thread will be valid
        std::atomic_thread_fence(std::memory_order_acquire);
    }

    void Release()
    {
        // use release semantics to ensure that all prior
        // writes have been fully committed before we unlock
        std::atomic_thread_fence(std::memory_order_release);

        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());
        std::size_t actual = m_atomic.load(std::memory_order_relaxed);
        assert(actual == tid);

        --m_refCount;
        if (m_refCount == 0)
        {
            // release lock, which is safe because we own it
            m_atomic.store(0, std::memory_order_relaxed);
        }
    }

    bool TryAcquire()
    {
        std::hash<std::thread::id> hasher;
        std::size_t tid = hasher(std::this_thread::get_id());

        bool acquired = false;

        if (m_atomic.load(std::memory_order_relaxed) == tid)
        {
            acquired = true;
        }
        else
        {
            std::size_t unlockValue = 0;
            acquired = m_atomic.compare_exchange_strong(
                unlockValue,
                tid,
                std::memory_order_relaxed, // fence below!
                std::memory_order_relaxed);
        }
        if (acquired)
        {
            ++m_refCount;
```

```
            std::atomic_thread_fence(
                std::memory_order_acquire);
        }
        return acquired;
    }
};
```

4.9.7.4 读写锁

在多个线程可以读写共享数据对象的系统中，我们可以使用互斥锁或自旋锁来控制对该对象的访问。但是，应该允许多个线程并发读取共享对象。只有在修改共享对象时，我们才需要确保互斥性。我们需要一种允许任意数量的读取者并发获取的锁。每当写入者线程尝试获取锁时，它应该等到所有读取者都完成操作，然后以特殊的“独占”模式获取锁，这样在它完成对共享对象的修改之前，任何其他读取者或写入者都无法获得访问权限。这种锁称为读写锁（也称为共享排他锁或推送锁）。

我们可以用类似于实现可重入锁的方式实现读写锁。不过，我们不会将线程 ID 存储在原子变量中，而是存储一个引用计数，指示当前有多少个读者持有该锁。每当一个读者获取锁时，计数就会增加；每当一个读者释放锁时，计数就会减少。

那么，我们如何才能为写操作提供“独占”锁模式呢？我们需要做的就是保留一个（非常高的）引用计数值，并用它来表示当前有一个写操作持有该锁。如果我们的引用计数是一个无符号的 32 位整数，那么 0xFFFFFFFFU 就可以很好地作为保留值。更简单的方法是，我们可以直接保留最高有效位，这意味着从 0 到 0x7FFFFFFFU 的引用计数表示读操作锁，而保留值 0x80000000U 表示写操作锁（其他值均无效）。

读写锁存在饥饿问题：如果写入者持有锁的时间过长，可能会导致所有读取者都处于饥饿状态；同样，如果读取者过多，也会导致写入者处于饥饿状态。顺序锁是解决饥饿问题的一种可能方案（详情请参阅 https://en.wikipedia.org/wiki/Seqlock）。有关 Linux 内核中使用的另一种有趣的锁定技术的描述，请参阅 https://lwn.net/Articles/262464，该技术支持多个并发读取者和写入者，称为读取-复制-更新 (RCU)。


```
            std::atomic_thread_fence(
                std::memory_order_acquire);
        }
        return acquired;
    }
};
```

4.9.7.4 Readers-Writer Locks

In a system in which multiple threads can read and write a shared data object, we could control access to the object using a mutex or a spin lock. However, multiple threads should be allowed to *read* the shared object concurrently. It's only when the shared object is being *mutated* that we need to ensure mutual exclusivity. What we'd like is a kind of lock that allows any number of readers to acquire it concurrently. Whenever a writer thread attempts to acquire the lock, it should wait until all readers are finished, and then it should acquire the lock in a special "exclusive" mode that prevents any other readers or writers from gaining access until it has completed its mutation of the shared object. This kind of lock is called a *readers-writer lock* (also known as a *shared-exclusive lock* or a *push lock*).

We can implement a readers-writer lock in a manner similar to how we implemented our reentrant lock. However, instead of storing the thread id in the atomic variable, we'll store a reference count indicating how many readers currently hold the lock. Each time a reader acquires the lock, the count is incremented; each time a reader releases the lock, the count is decremented.

But how, then, can we also provide an "exclusive" lock mode for the writer? All we need to do is to reserve one (very high) reference count value and use it to denote that a writer currently holds the lock. If our reference count is an unsigned 32-bit integer, the value 0xFFFFFFFFU could do nicely as the reserved value. Even easier, we can simply reserve the most-significant bit, meaning that reference counts from 0 to 0x7FFFFFFFU represent reader locks, and the reserved value 0x80000000U represents a writer lock (with no other values being valid).

A readers-writer lock suffers from starvation problems: A writer that holds the lock too long can cause all of the readers to starve, and likewise a lot of readers can cause the writers to starve. A *sequential lock* is one possible alternative that tackles the starvation issue (see https://en.wikipedia.org/wiki/Seqlock for details). Check out https://lwn.net/Articles/262464 for a description of yet another interesting locking technique, used in the Linux kernel, that supports multiple concurrent readers and writers, called *read-copy-update* (RCU).

我们将读者-写者锁的实现留给你作为练习！不过，如果你想比较一下笔记，你可以在本书的网站（www.gameenginebook.com）上找到一个功能齐全的实现。

4.9.7.5 无需锁断言

无论如何，锁都是昂贵的。即使在没有竞争的情况下，互斥锁也是昂贵的。在低竞争的情况下，自旋锁相对便宜，但它仍然会给任何软件带来非零成本。

程序员通常事先知道不需要锁。例如，在游戏引擎中，游戏循环的每次迭代通常分阶段执行。如果在帧的早期阶段，一个共享数据结构被一个线程以独占方式访问，而同一数据在帧的后期阶段又被一个线程访问，那么我们实际上并不需要锁。没错，理论上这两个线程可以重叠，如果它们重叠，那么肯定需要锁。但实际上，考虑到游戏循环的结构，我们可能知道这种重叠永远不会发生。

在这种情况下，我们有几个选择。我们可以设置锁，以防万一。这样，如果有人重新安排了该帧中任务的执行顺序，导致这些线程重叠，我们就能得到保障。另一个选择是忽略重叠的可能性，不设置任何锁。

在这种情况下，我认为还有第三种选择更有吸引力：我们可以断言不需要锁。这有两个好处。首先，它的成本非常低，而且实际上可以在游戏发布之前删除断言。其次，如果我们对线程重叠的假设被证明是错误的，或者这些假设在之后的代码重构中被打破，它会自动检测到问题。这种断言没有标准化的名称，因此在本书中我们称之为“不需要锁的断言”。

那么，我们如何检测是否需要锁呢？一种方法是使用一个原子布尔变量，并进行适当的内存隔离，像互斥锁一样使用它。只不过，我们不是真正地获取互斥锁，而是简单地断言布尔值为假，然后以原子方式将其设置为真。同样，我们不是释放锁，而是断言布尔值为真，然后以原子方式将其设置为假。这种方法虽然有效，但它的开销与无竞争自旋锁一样高。我们可以做得更好。

诀窍在于，我们只关心检测共享对象上关键操作之间的重叠。而且这种检测不必 100% 可靠。90% 的命中率可能就足够了。如果两个关键操作确实重叠，有时我们可能无法检测到。但如果你的游戏每天都由 100 名或更多开发者组成的团队运行多次，

We'll leave the implementation of a readers-writer lock up to you as an exercise! However, if you'd like to compare notes, you can find a fully-functional implementation on this book's website (www.gameenginebook.com).

4.9.7.5 Lock-Not-Needed Assertions

No matter how you slice it, locks are expensive. Mutexes are expensive even in the absence of contention. In a low-contention scenario, spin locks are relatively cheap, but they still introduce a non-zero cost into any piece of software.

It's often the case that the programmer knows *a priori* that a lock isn't required. In a game engine, for example, each iteration of the game loop is usually performed in distinct stages. If a shared data structure is accessed exclusively by a single thread early in the frame, and that same data is accessed again by a single thread later in the frame, then we don't actually need a lock. Yes, in theory those two threads could overlap, and if they were to do so a lock would definitely be needed. But in practice, given the way our game loop is structured, we might know that such overlap can never occur.

In this kind of situation, we have a few choices. We could put the locks in, just in case. That way, if someone rearranges the order in which things are done during the frame, and ends up making these threads overlap, we'd be covered. Another option is to just ignore the possibility of overlap and not lock anything.

There is a third option which I find more appealing in such scenarios: We can *assert* that a lock isn't required. This has two benefits. First, it can be done very cheaply, and in fact the assertions can be stripped prior to shipping your game. Second, it automatically detects problems if our assumptions about the overlap of the threads proves to be incorrect—or if the assumptions are broken later on by a refactor of the code. There's no standardized name for this kind of assertion, so we'll call them *lock-not-needed assertions* in this book.

So how do we detect that a lock is needed? One way would be to use an atomic Boolean variable, complete with proper memory fencing, and use it like a mutex. Except that instead of actually acquiring a mutex lock, we would simply assert that the Boolean is false, and then set it to true atomically. And instead of releasing a lock, we assert that our Boolean is true, and then set it to false atomically. This approach would work, but it would be just as expensive as an uncontended spin lock. We can do better.

The "trick" is to realize that we only care about *detecting* overlaps between critical operations on a shared object. And that detection needn't be 100% reliable. A 90% hit rate is probably just fine. If two critical operations ever *do* overlap, there may be times when we fail to detect it. But if your game is being run multiple times a day, every day, by a team of 100 or more developers,

再加上至少由 10 或 20 人组成的 QA 部门，您可以非常肯定，如果存在问题，就会有人检测到。

因此，我们不用原子布尔值，而是使用 volatile 布尔值。正如我们所说， volatile 关键字对防止并发竞争错误作用不大。但它确实保证了布尔值的读写不会被优化掉，而这正是我们所需要的。这样一来，我们将获得相当不错的检测率，而且测试的开销非常小。

```
class UnnecessaryLock
{
    volatile bool    m_locked;

public:
    void Acquire()
    {
        // assert no one already has the lock
        assert(!m_locked);

        // now lock (so we can detect overlapping
        // critical operations if they happen)
        m_locked = true;
    }
    void Release()
    {
        // assert correct usage (that Release()
        // is only called after Acquire())
        assert(m_locked);

        // unlock
        m_locked = false;
    }
};

#if ASSERTIONS_ENABLED
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L)  (L).Acquire()
#define END_ASSERT_LOCK_NOT_NECESSARY(L)    (L).Release()
#else
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L)
#define END_ASSERT_LOCK_NOT_NECESSARY(L)
#endif

// Example usage...

UnnecessaryLock g_lock;
```


plus a QA department consisting of at least another 10 or 20 people, you can be pretty sure *someone* will detect the problem if one exists.

So, instead of an *atomic* Boolean, we'll just use a `volatile` Boolean. As we've stated, the `volatile` keyword doesn't do much to prevent concurrent race bugs. But it does guarantee that reads and writes of the Boolean won't be optimized away, and that's really all we need. We'll get a reasonably good detection rate, and the test is dirt cheap.

```
class UnnecessaryLock
{
    volatile bool    m_locked;

public:
    void Acquire()
    {
        // assert no one already has the lock
        assert(!m_locked);

        // now lock (so we can detect overlapping
        // critical operations if they happen)
        m_locked = true;
    }
    void Release()
    {
        // assert correct usage (that Release()
        // is only called after Acquire())
        assert(m_locked);

        // unlock
        m_locked = false;
    }
};

#if ASSERTIONS_ENABLED
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L)  (L).Acquire()
#define END_ASSERT_LOCK_NOT_NECESSARY(L)    (L).Release()
#else
#define BEGIN_ASSERT_LOCK_NOT_NECESSARY(L)
#define END_ASSERT_LOCK_NOT_NECESSARY(L)
#endif

// Example usage...

UnnecessaryLock g_lock;
```

```
void EveryCriticalOperation()
{
    BEGIN_ASSERT_LOCK_NOT_NECESSARY(g_lock);

    printf("perform critical op...\n");

    END_ASSERT_LOCK_NOT_NECESSARY(g_lock);
}
```

我们还可以将锁包裹在看门人中（参见第 3.1.1.6 节），如下所示：

```
class UnnecessaryLockJanitor
{
    UnnecessaryLock* m_pLock;
public:
    explicit
    UnnecessaryLockJanitor(UnnecessaryLock& lock)
        : m_pLock(&lock) { m_pLock->Acquire(); }
    ~UnnecessaryLockJanitor() { m_pLock->Release(); }
};

#if ASSERTIONS_ENABLED
#define ASSERT_LOCK_NOT_NECESSARY(J,L) \
    UnnecessaryLockJanitor J(L)
#else
#define ASSERT_LOCK_NOT_NECESSARY(J,L)
#endif

// Example usage...

UnnecessaryLock g_lock;

void EveryCriticalOperation()
{
    ASSERT_LOCK_NOT_NECESSARY(janitor, g_lock);

    printf("perform critical op...\n");
}
```

我们在顽皮狗实现了这个功能，它成功捕获了许多关键操作重叠的情况，而程序员们此前一直认为这些重叠情况根本不可能发生。所以，这个小妙招是经过验证的。

4.9.8 无锁事务

这本该是关于无锁编程的章节，但到目前为止，我们把所有时间都花在了编写自旋锁上！也许与直觉相反，


```
void EveryCriticalOperation()
{
    BEGIN_ASSERT_LOCK_NOT_NECESSARY(g_lock);

    printf("perform critical op...\n");

    END_ASSERT_LOCK_NOT_NECESSARY(g_lock);
}
```

We could also wrap the locks in a janitor (see Section 3.1.1.6), like this:

```
class UnnecessaryLockJanitor
{
    UnnecessaryLock* m_pLock;
public:
    explicit
    UnnecessaryLockJanitor(UnnecessaryLock& lock)
        : m_pLock(&lock) { m_pLock->Acquire(); }
    ~UnnecessaryLockJanitor() { m_pLock->Release(); }
};

#if ASSERTIONS_ENABLED
#define ASSERT_LOCK_NOT_NECESSARY(J,L) \
    UnnecessaryLockJanitor J(L)
#else
#define ASSERT_LOCK_NOT_NECESSARY(J,L)
#endif

// Example usage...

UnnecessaryLock g_lock;

void EveryCriticalOperation()
{
    ASSERT_LOCK_NOT_NECESSARY(janitor, g_lock);

    printf("perform critical op...\n");
}
```

We implemented this at Naughty Dog and it successfully caught a number of cases of critical operations overlapping when the programmers had assumed they never could do so. So this little gem is tried and true.

4.9.8 Lock-Free Transactions

This is supposed to be a section on lock-free programming, but thus far we've spent all of our time on writing spin locks! Perhaps counterintuitively, the

从自旋锁本身的实现角度来看，编写自旋锁的行为就是无锁编程的一个例子。在此过程中，我们还学习了很多关于原子指令、编译器屏障和内存栅栏的知识。所以这是一个很有意义的练习。然而，我们还没有真正探索无锁编程本身的原理；为此，看一个自旋锁以外的示例会很有启发。无锁和非阻塞算法是一个庞大的话题。它真的值得写一本书来探讨，所以我们不会在这里深入探讨。但我们至少可以了解一下无锁方法通常是如何工作的。

无锁编程的目标当然是避免使用会导致线程休眠或陷入自旋锁内忙等待循环的锁。为了以无锁方式执行关键操作，我们需要将每个此类操作视为一个事务，它要么完全成功，要么完全失败。如果失败，则只需重试该事务，直到成功为止。

为了实现任何事务，无论多么复杂，我们都会在本地执行大部分工作（即使用仅对当前线程可见的数据，而不是直接操作共享数据）。当一切准备就绪，事务准备提交时，我们会执行一条原子指令，例如 CAS 或 LL/SC。如果此原子指令成功执行，则表示我们已成功全局“发布”了事务——它将成为我们正在操作的共享数据结构的永久组成部分。但是，如果原子指令失败，则意味着其他线程正在与我们同时尝试提交事务。

这种“失败-重试”方法之所以有效，是因为每当一个线程提交事务失败时，我们都知道这是因为其他线程成功了。因此，系统中总会有一个线程在向前推进（只是可能不是我们）。这就是无锁的定义。

4.9.9 无锁链表

举个具体的例子，我们来看一个简单的无锁单链表。本次讨论中我们唯一支持的操作是 push_front()。

为了准备事务，我们分配新的 Node 并向其中填充数据。我们还将其 next 指针设置为指向当前链表头部的节点。事务现在可以原子提交了。

提交本身包含对 head 指针调用 compare_exchange_weak()，我们将 head 指针声明为指向 Node 的原子指针。如果此调用成功，则表示我们将新节点插入到链表的头部，操作完成。但如果失败，则需要重试。这需要重新初始化我们的

act of writing a spin lock is an example of lock-free programming, from the perspective of the implementation of the spin lock itself. We've also learned a lot about atomic instructions, compiler barriers and memory fences along the way. So this has been a useful exercise. However, we haven't really explored the principles of lock-free programming per se; for that purpose it will be instructive to look at an example other than a spin lock. The topic of lock-free and non-blocking algorithms is huge. It really deserves its own book, so we won't attempt to cover the topic in depth here. But we can at least get a feel for how lock-free approaches usually work.

The goal of lock-free programming is of course to avoid taking locks that will either put a thread to sleep, or cause it to get caught up in a busy-wait loop inside a spin lock. To perform a critical operation in a lock-free manner, we need to think of each such operation as a *transaction* that can either succeed in its entirety, or fail in its entirety. If it fails, the transaction is simply retried until it does succeed.

To implement any transaction, no matter how complex, we perform the majority of the work *locally* (i.e., using data that's visible only to the current thread, rather than operating directly on the shared data). When all of our ducks are in a row and the transaction is ready to *commit*, we execute a *single atomic instruction*, such as CAS or LL/SC. If this atomic instruction succeeds, we have successfully "published" our transaction globally—it becomes a permanent part of the shared data structure on which we're operating. But if the atomic instruction fails, that means some *other* thread was attempting to commit a transaction at the same time we were.

This fail-and-retry approach works because whenever one thread fails to commit its transaction, we know that it was because some *other* thread managed to *succeed*. As a result, one thread in the system is always making forward progress (just maybe not us). And that is the definition of *lock-free*.

4.9.9 A Lock-Free Linked List

As a concrete example, let's look at a simple lock-free singly-linked list. The only operation we'll support in this discussion is `push_front()`.

To prepare the transaction, we allocate the new `Node` and populate it with data. We also set its next pointer to point to whichever node is currently at the head of the linked list. The transaction is now ready to commit atomically.

The commit itself consists of a call to `compare_exchange_weak()` on the head pointer, which we declared as an atomic pointer to a `Node`. If this call succeeds, we've inserted our new node at the head of the linked list and we're done. But if it fails, we need to retry. This involves re-initializing our

图 4.37. 单链表头部插入操作的无锁实现。上图：事务准备阶段，将新节点的下一个指针设置为指向链表的当前头部。下图：事务提交阶段，使用原子 CAS 操作将头部指针与指向新节点的指针交换。如果 CAS 失败，则返回顶部重试，直到成功。

新节点的下一个指针指向现在可能是一个全新的头节点（可能是由另一个线程插入的——这也许就是我们一开始失败的原因）。这个两阶段的过程如图 4.37 所示。

在下面的代码中，你不会看到节点的 next 指针被显式地重新初始化。这是因为 compare_exchange_weak() 已经为我们完成了重新初始化的步骤。（多么方便！）代码如下所示：

```
template< class T >
class SList
{
    struct Node
    {
        T      m_data;
        Node*  m_pNext;
    };
    std::atomic< Node* > m_head { nullptr };

public:
    void push_front(T data)
    {
        // prep the transaction locally
        auto pNode = new Node();
        pNode->m_data = data;
```

Figure 4.37. A lock-free implementation of insertion at the head of a singly-linked list. Top: The transaction is prepared by setting the next pointer of the new node to point to the current head of the list. Bottom: The transaction is committed by using an atomic CAS operation to swap the head pointer with a pointer to the new node. If the CAS fails, we return to the top and try again until it succeeds.

new node's next pointer to point to what is now potentially a *brand new* head node (presumably inserted by another thread—this is perhaps why we failed in the first place). This two-stage process is illustrated in Figure 4.37.

In the code below, you won't see an explicit re-initialization of the node's next pointer. That's because `compare_exchange_weak()` does the re-initialization step for us. (How convenient!) Here's what the code would look like:

```
template< class T >
class SList
{
    struct Node
    {
        T      m_data;
        Node*  m_pNext;
    };
    std::atomic< Node* > m_head { nullptr };

public:
    void push_front(T data)
    {
        // prep the transaction locally
        auto pNode = new Node();
        pNode->m_data = data;
```

```
        pNode->m_pNext = m_head;

        // commit the transaction atomically
        // (retrying until it succeeds)
        while (!m_head.compare_exchange_weak(
                      pNode->m_pNext, pNode))
        { }
    }
};
```

4.9.10 关于无锁编程的进一步阅读

并发是一个博大精深的主题，本章我们只是略微触及了皮毛。与往常一样，本书的目标仅仅是帮助读者建立认知，并作为进一步学习的起点。

- 有关如何实现无锁单链表的完整讨论，请查看 Herb Sutter 在 CppCon 2014 上的演讲，上面的例子就出自此演讲。该演讲可在 YouTube 上观看，分为两部分：
  - https://www.youtube.com/watch?v=c1gO9aB9nbs，以及
  - https://www.youtube.com/watch?v=CmxkPChOcvw。
- 卡内基梅隆大学的 Geoff Langdale 的讲座提供了很好的概述：https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf。
- 另外，请务必查看 Samy Al Bahra 的此演示文稿，以获得与并发编程相关的几乎所有主题的清晰且易于理解的概述：http://concurrencykit.org/presentations/lockfree_introduction/#/。
- Mike Acton 关于并发思维的精彩演讲是必读之作；可在 http://cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf 获取。
- 这两本在线书籍是并发编程的优秀资源：http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf 和 https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf。
- 在 Jeff Preshing 的博客上可以找到一些关于无锁编程以及原子、屏障和栅栏如何工作的优秀文章：http://preshing.com/20120612/an-introduction-to-lock-free-programming。


```
        pNode->m_pNext = m_head;

        // commit the transaction atomically
        // (retrying until it succeeds)
        while (!m_head.compare_exchange_weak(
                      pNode->m_pNext, pNode))
        { }
    }
};
```

4.9.10 Further Reading on Lock-Free Programming

Concurrency is a vast and profound topic, and in this chapter we've only just scratched the surface. As always, the goal of this book is merely to build awareness and serve as a jumping-off point for further learning.

- For a complete discussion of implementing a lock-free singly-linked list, check out Herb Sutter's talk at CppCon 2014, which is where the example above came from. The talk is available on YouTube in two parts:
  - https://www.youtube.com/watch?v=c1gO9aB9nbs, and
  - https://www.youtube.com/watch?v=CmxkPChOcvw.
- This lecture by Geoff Langdale of CMU provides a great overview: https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf.
- Also be sure to check out this presentation by Samy Al Bahra for a clear and easily digestible overview of pretty much every topic under the sun related to concurrent programming: http://concurrencykit.org/presentations/lockfree_introduction/#/.
- Mike Acton's excellent talk on concurrent thinking is a must-read; it is available at http://cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf.
- These two online books are great resources for concurrent programming: http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf and https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf.
- Some excellent articles on lock-free programming and how atomics, barriers and fences work can be found on Jeff Preshing's blog: http://preshing.com/20120612/an-introduction-to-lock-free-programming.

- 此页面包含有关 Linux 上的内存屏障的大量信息：https://www.mjmwired.net/kernel/Documentation/memory-barriers.txt#305

4.10 SIMD/矢量处理

在 4.1.4 节中，我们介绍了一种称为单指令多数据 (SIMD) 的并行形式。这指的是大多数现代微处理器使用单条机器指令并行执行多个数据项的数学运算的能力。在本节中，我们将详细探讨 SIMD 技术，并在本章最后简要讨论如何将 SIMD 和多线程结合成一种称为单指令多线程 (SIMT) 的并行形式，这构成了所有现代 GPU 的基础。

英特尔于 1994 年在其奔腾系列 CPU 中首次引入了 MMX 11 指令集。这些指令允许对 8 个 8 位整数、4 个 16 位整数或 2 个 32 位整数执行 SIMD 计算，这些整数被封装在特殊的 64 位 MMX 寄存器中。随后，英特尔又对名为“流式 SIMD 扩展”（SSE）的扩展指令集进行了多次修订，其首个版本出现在奔腾 III 处理器中。

SSE 指令集使用 128 位寄存器，可以包含整数或 IEEE 浮点数据。游戏引擎最常用的 SSE 模式称为压缩 32 位浮点模式。在此模式下，四个 32 位浮点值被打包到单个 128 位寄存器中。因此，通过将其中两个 128 位寄存器作为输入，可以对四对浮点数并行执行加法或乘法等运算。此后，英特尔对 SSE 指令集进行了各种升级，分别称为 SSE2、SSE3、SSSE3 和 SSE4。2007 年，AMD 推出了自己的变体，分别称为 XOP、FMA4 和 CVT16。

2011年，英特尔推出了一种新的、更宽的SIMD寄存器文件及其配套指令集，称为高级矢量扩展 (AVX)。AVX寄存器宽256位，允许单条指令并行操作最多八个32位浮点操作数对。AVX2指令集是AVX的扩展。部分英特尔CPU现在支持AVX-512，这是AVX的扩展，允许将16个32位浮点数打包到512位寄存器中。

11 官方说法是，MMX 是英特尔注册的一个毫无意义的缩写。非正式说法是，开发者认为它代表“多媒体扩展”或“矩阵数学扩展”。

- This page has great information about memory barriers on Linux: https://www.mjmwired.net/kernel/Documentation/memory-barriers.txt#305

4.10 SIMD/Vector Processing

In Section 4.1.4, we introduced a form of parallelism known as *single instruction multiple data* (SIMD). This refers to the ability of most modern microprocessors to perform a mathematical operation on multiple data items *in parallel*, using a single machine instruction. In this section, we'll explore SIMD techniques in some detail, and conclude the chapter with a brief discussion of how SIMD and multithreading are combined into a form of parallelsm known *single instruction multiple thread* (SIMT), which forms the basis of all modern GPUs.

Intel first introduced its MMX[11] instruction set with their Pentium line of CPUs in 1994. These instructions permitted SIMD calculations to be performed on eight 8-bit integers, four 16-bit integers, or two 32-bit integers packed into special 64-bit MMX registers. Intel followed this up with various revisions of an extended instruction set called *streaming SIMD extensions*, or SSE, the first version of which appeared in the Pentium III processor.

The SSE instruction set utilizes 128-bit registers that can contain integer or IEEE floating-point data. The SSE mode most commonly used by game engines is called *packed 32-bit floating-point mode*. In this mode, four 32-bit `float` values are packed into a single 128-bit register. An operation such as addition or multiplication can thus be performed on four pairs of `floats` in parallel by taking two of these 128-bit registers as its inputs. Intel has since introduced various upgrades to the SSE instruction set, named SSE2, SSE3, SSSE3 and SSE4. In 2007, AMD introduced its own variants named XOP, FMA4 and CVT16.

In 2011, Intel introduced a new, wider SIMD register file and accompanying instruction set named *advanced vector extensions* (AVX). AVX registers are 256 bits wide, permitting a single instruction to operate on pairs of up to eight 32-bit floating-point operands in parallel. The AVX2 instruction set is an extension to AVX. Some Intel CPUs now support AVX-512, an extension to AVX permitting 16 32-bit floats to be packed into a 512-bit register.

[11] Officially, MMX is a meaningless initialism trademarked by Intel. Unofficially, developers consider it to stand for “multimedia extensions” or “matrix math extensions.”

图 4.38 32 位浮点模式下 SSE 寄存器的四个组件。

4.10.1 SSE指令集及其寄存器

SSE 指令集包含种类繁多的操作，其中有多种变体可用于操作 SSE 寄存器中不同大小的数据元素。然而，为了便于讨论，我们将重点讨论处理打包 32 位浮点数据的相对较小的指令子集。这些指令带有 ps 后缀，表示我们处理的是打包数据 (p)，并且每个元素都是单精度浮点数 (s)。不过，接下来的讨论大部分内容都会直观地延伸到 AVX 的 256 位和 512 位模式；有关 AVX 的概述，请参阅 https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions。

SSE 寄存器名为 XMM i ，其中 i 是 0 到 15 之间的整数（例如，XMM0、XMM 1 等等）。在压缩 32 位浮点模式下，每个 128 位 XMM i 寄存器包含四个 32 位浮点数。在 AVX 中，这些寄存器的宽度为 256 位，名为 YMM i ；在 AVX-512 中，这些寄存器的宽度为 512 位，名为 ZMM i 。

在本章中，我们通常会将 SSE 寄存器中的单个浮点数称为 [xyzw] ，就像在纸面上进行齐次坐标系下的向量/矩阵运算时一样（参见图 4.38）。通常情况下，如何称呼 SSE 寄存器中的元素并不重要，只要你对每个元素的解释保持一致即可。最通用的方法是将 SSE 向量 r 视为包含元素 [r 0 r 1 r 2 r 3] 。大多数 SSE 文档都使用此约定，但有些文档使用 [wxyz] 约定，因此请务必谨慎！

4.10.1.1 __m128 数据类型

为了使 SSE 指令能够对打包浮点数据执行算术运算，该数据必须驻留在 XMM i 寄存器之一中。对于长期存储，打包浮点数据当然可以驻留在内存中，但在用于任何计算之前，必须先将其从 RAM 传输到 SSE 寄存器中，然后将结果传输回 RAM。

为了更轻松地处理 SSE 和 AVX 数据，C 和 C++ 编译器提供了表示 float 压缩数组的特殊数据类型。__m128 类型封装了一个由 4 个 float 组成的压缩数组，用于 SSE 内部函数。（__m256 和 __m512 类型同样表示由 8 个 float 组成的压缩数组）

Figure 4.38. The four components of an SSE register in 32-bit floating-point mode.

4.10.1 The SSE Instruction Set and Its Registers

The SSE instruction set includes a wide variety of operations, with many variants for operating on differently-sized data elements within SSE registers. For the purposes of the present discussion, however, we'll stick to the relatively small subset of instructions that deal with packed 32-bit floating-point data. These instructions are denoted with a `ps` suffix, indicating that we're dealing with *packed* data (p), and that each element is a *single*-precision `float` (s). However most of the upcoming discussions extend intuitively into AVX's 256- and 512-bit modes; see https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions for an overview of AVX.

The SSE registers are named XMM*i*, where *i* is an integer between 0 and 15 (e.g., XMM0, XMM1, and so on). In packed 32-bit floating-point mode, each 128-bit XMM*i* register contains four 32-bit floats. In AVX, the registers are 256 bits wide and are named YMM*i*; in AVX-512, they are 512 bits in width and are named ZMM*i*.

In this chapter, we'll often refer to the individual floats within an SSE register as $\begin{bmatrix} x & y & z & w \end{bmatrix}$, just as they would be when doing vector/matrix math in homogeneous coordinates on paper (see Figure 4.38). It doesn't usually matter what you call the elements of an SSE register, as long as you're consistent about how you *interpret* each element. The most general approach is to think of an SSE vector $\mathbf{r}$ as containing the elements $\begin{bmatrix} r_0 & r_1 & r_2 & r_3 \end{bmatrix}$. Most SSE documentation uses this convention, although some documentation uses a $\begin{bmatrix} w & x & y & z \end{bmatrix}$ convention, so be careful out there!

4.10.1.1 The __m128 Data Type

In order for SSE instructions to perform arithmetic on packed floating-point data, that data must reside in one of the XMM*i* registers. For long-term storage, packed floating-point data can of course reside in memory, but it must be transferred from RAM into an SSE register prior to being used for any calculations, and the results subsequently transferred back to RAM.

To make working with SSE and AVX data easier, C and C++ compilers provide special data types that represent packed arrays of `floats`. The `__m128` type encapsulates a packed array of four `floats` for use with the SSE intrinsics. (The `__m256` and `__m512` types likewise represent packed arrays of eight

和 16 个 float，用于 AVX 内部函数。）__m128 数据类型及其同类可用于声明全局变量、自动变量、函数参数和返回类型，甚至类和结构成员。将自动变量和函数参数声明为 __m128 类型通常会导致编译器将这些值视为 SSE 寄存器的直接代理。但使用 __m128 类型声明全局变量、结构/类成员以及有时的局部变量会导致数据在内存中存储为 16 字节对齐的浮点数组。在 SSE 计算中使用基于内存的 __m128 变量将导致编译器隐式发出指令，用于在对数据执行计算之前将数据从内存加载到 SSE 寄存器中，同样发出指令将计算结果存储回"支持"每个此类变量的内存中。因此，最好检查反汇编以确保在使用 __m128 类型（及其 AVX 相关类型）时没有对 SSE 寄存器进行不必要的加载和存储。

4.10.1.2 SSE 数据的对齐

每当要将用于 XMM i 寄存器的数据存储在内存中时，它都必须是 16 字节（128 位）对齐的。（同样，要用于 AVX 的 256 位 YMM i 寄存器的数据必须是 32 字节（256 位）对齐的，而要用于 512 位 ZMM i 寄存器的数据必须是 64 字节（512 位）对齐的。）编译器确保 __m128 类型的全局变量和局部变量自动对齐。它还会填充结构体和类成员，以便任何 __m128 成员都相对于对象的开头正确对齐，并确保整个结构体或类的对齐等于其成员的最坏情况对齐。这意味着，声明包含至少一个 __m128 成员的结构体或类的全局或局部变量实例时，编译器会自动对其进行 16 字节对齐。

但是，所有此类结构体或类的动态分配实例都需要手动对齐。同样，任何打算与 SSE 指令一起使用的浮点数组也必须正确对齐；您可以通过 C++11 alignas 说明符来确保这一点。有关对齐内存分配的更多信息，请参阅第 6.2.1.3 节。

4.10.1.3 SSE 编译器内部函数

我们可以直接使用 SSE 和 AVX 汇编语言指令，或许可以使用编译器的内联汇编语法。然而，编写这样的代码不仅不可移植，而且非常麻烦！为了简化操作，现代编译器提供了内部函数——一种看起来和

and 16 `floats`, respectively, for use with AVX intrinsics.)

The `__m128` data type and its kin can be used to declare global variables, automatic variables, function arguments and return types, and even class and structure members. Declaring automatic variables and function arguments to be of type `__m128` often results in the compiler treating those values as direct proxies for SSE registers. But using the `__m128` type to declare global variables, structure/class members, and sometimes local variables results in the data being stored as a 16-byte aligned array of `float` in memory. Using a memory-based `__m128` variable in an SSE calculation will cause the compiler to implicitly emit instructions for loading the data from memory into an SSE register prior to performing the calculation on it, and likewise to emit instructions for storing the results of the calculation back into the memory that "backs" each such variable. As such, it's a good idea to check the disassembly to make sure that you're not doing unnecessary loads and stores of SSE registers when using the `__m128` type (and its AVX relatives).

4.10.1.2 Alignment of SSE Data

Whenever data that is intended for use in an XMM*i* register is stored in memory, it must be 16-byte (128-bit) aligned. (Likewise, data intended for use with AVX's 256-bit YMM*i* registers must be 32-byte (256-bit) aligned, and data for use with the 512-bit ZMM*i* registers must be 64-byte (512-bit) aligned.)

The compiler ensures that global and local variables of type `__m128` are aligned automatically. It also pads `struct` and `class` members so that any `__m128` members are aligned properly *relative to* the start of the object, and ensures that the alignment of the entire struct or class is equal to the worst-case alignment of its members. This means that declaring a global or local variable instance of a struct or class that includes at least one `__m128` member will be 16-byte aligned by the compiler automatically.

However, all *dynamically allocated* instances of such a struct or class need to be aligned manually. Likewise, any array of `float` that you intend to use with SSE instructions must be properly aligned; you can ensure this via the C++11 `alignas` specifier. See Section 6.2.1.3 for more information on aligned memory allocations.

4.10.1.3 SSE Compiler Intrinsics

We could work directly with the SSE and AVX assembly language instructions, perhaps using our compiler's inline assembly syntax. However, writing code like this is not only non-portable, it's also a big pain in the butt! To make life easier, modern compilers provide *intrinsics*—special syntax that looks and

其行为类似于常规 C 函数，但实际上会被编译器简化为内联汇编代码。许多内在函数会被转换为一条汇编语言指令，但也有一些是宏，会被转换为一系列指令。

为了使用 SSE 和 AVX 内部函数，您的 .cpp 文件必须在 Visual Studio 中 #include <xmmintrin.h>，或者在使用 Clang 或 gcc 编译时包含 <x86intrin.h>。

4.10.1.4 一些有用的 SSE 内部函数

SSE 内部函数有很多，但为了便于讨论，我们仅从其中五个开始：

- **`__m128 _mm_set_ps(float w, float z, float y, float x);`**
  此内在函数使用提供的四个浮点值初始化 __m128 变量。

- **`__m128 _mm_load_ps(const float* pData);`**
  此内在函数将 C 样式数组中的四个浮点数加载到 __m128 变量中。输入数组必须为 16 字节对齐。

- **`void _mm_store_ps(float* pData, __m128 v);`**
  此内在函数将 __m128 变量的内容存储到四个 float 的 C 样式数组中，该数组必须按 16 字节对齐。

- **`__m128 _mm_add_ps(__m128 a, __m128 b);`**
  此内在函数将变量 a 和 b 中包含的四对 float 并行相加并返回结果。

- **`__m128 _mm_mul_ps(__m128 a, __m128 b);`**
  此内在函数并行将变量 a 和 b 中包含的四对浮点数相乘并返回结果。

您可能已经注意到，参数 x 、 y 、 z 和 w 以相反的顺序传递给 _mm_set_ps() 函数。这种奇怪的约定可能源于 Intel CPU 是 little-endian 的。正如位模式为 0x12345678 的单个浮点值将按地址递增的顺序存储在字节 0x78、0x56、0x34、0x12 中一样，SSE 寄存器的内容在内存中的存储顺序也与这些组件在寄存器中实际出现的顺序相反。换句话说，不仅 SSE 寄存器中每个浮点数的四个字节以 little-endian 顺序存储，而且四个浮点数本身也是如此。所有这些只是命名约定的问题：实际上没有“最高有效位”

behaves like a regular C function, but is actually boiled down to inline assembly code by the compiler. Many intrinsics translate into a single assembly language instruction, although some are *macros* that translate into a sequence of instructions.

In order to use SSE and AVX intrinsics, your .cpp file must `#include <xmmintrin.h>` in Visual Studio, or `<x86intrin.h>` when compiling with Clang or gcc.

4.10.1.4 Some Useful SSE Intrinsics

There are a lot of SSE intrinsics, but for the purposes of our discussion here we can start with only five of them:

- **`__m128 _mm_set_ps(float w, float z, float y, float x);`**
  This intrinsic initializes an `__m128` variable with the four floating-point values provided.

- **`__m128 _mm_load_ps(const float* pData);`**
  This intrinsic loads four `floats` from a C-style array into an `__m128` variable. The input array must be 16-byte aligned.

- **`void _mm_store_ps(float* pData, __m128 v);`**
  This intrinsic stores the contents of an `__m128` variable into a C-style array of four `floats`, which must be 16-byte aligned.

- **`__m128 _mm_add_ps(__m128 a, __m128 b);`**
  This intrinsic adds the four pairs of `floats` contained in the variables `a` and `b` in parallel and returns the result.

- **`__m128 _mm_mul_ps(__m128 a, __m128 b);`**
  This intrinsic multiplies the four pairs of `floats` contained in the variables `a` and `b` in parallel and returns the result.

You may have noticed that the arguments `x`, `y`, `z` and `w` are passed to the `_mm_set_ps()` function in reverse order. This strange convention probably arises from the fact that Intel CPUs are *little-endian*. Just as a single floating-point value with the bit pattern 0x12345678 would be stored in memory as the bytes 0x78, 0x56, 0x34, 0x12 in order of increasing addresses, so too are the contents of an SSE register stored in memory in an order that's opposite to the order those components actually appear within the register. In other words, not only are the four bytes comprising each float in an SSE register stored in little-endian order, but so too are the four floats themselves. All of this is just a question of naming convention: There's really no "most-significant"

或 SSE 寄存器中的“最低有效”浮点数。因此，我们可以将内存中的顺序视为“正确”顺序，并将 _mm_set_ps() 视为“反向”顺序，或者我们可以将 _mm_set_ps( ) 的参数视为“正确”顺序，并将所有内存中的向量视为“反向”。我们将坚持前一种约定，因为这意味着我们能够更自然地读取向量：由成员（vx、vy、vz、vw）组成的同构向量 v 将以 float v[] = { vx, vy, vz, vw } 的形式存储在 C/C++ 数组中，但将以 w、z、y 的形式传递给 _mm_set_ps()，

x.

下面是一小段代码片段，它加载两个四元素浮点向量，将它们相加，然后打印结果。

```
#include <xmmintrin.h>

void TestAddSSE()
{
    alignas(16) float A[4];
    alignas(16) float B[4] = { 2.0f, 4.0f, 6.0f, 8.0f };

    // set a = (1, 2, 3, 4) from literal values, and
    // load b = (2, 4, 6, 8) from a floating-point array
    // just to illustrate the two ways of doing this
    // (remember that _mm_set_ps() is backwards!)
    __m128 a = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f);
    __m128 b = _mm_load_ps(&B[0]);

    // add the vectors
    __m128 r = _mm_add_ps(a, b);

    // store '__m128 a' into a float array for printing
    _mm_store_ps(&A[0], a);

    // store result into a float array for printing
    alignas(16) float R[4];
    _mm_store_ps(&R[0], r);

    // inspect results
    printf("a = %.1f %.1f %.1f %.1f\n",
           A[0], A[1], A[2], A[3]);
    printf("b = %.1f %.1f %.1f %.1f\n",
           B[0], B[1], B[2], B[3]);
    printf("r = %.1f %.1f %.1f %.1f\n",
           R[0], R[1], R[2], R[3]);
}
```


or “least-significant” float within an SSE register. So we can either treat the in-memory order as the “correct” order and consider `_mm_set_ps()` to be “backwards,” or we can treat the arguments of `_mm_set_ps()` as being in the “correct” order and think of all in-memory vectors as being “backwards.” We'll stick with the former convention, since it means that we'll be able to read off our vectors more naturally: A homogeneous vector v consisting of members (v_x, v_y, v_z, v_w) would be stored in a C/C++ array as `float v[] = { vx, vy, vz, vw }`, but would be passed to `_mm_set_ps()` as w, z, y, x.

Here's a small code snippet that loads two four-element floating-point vectors, adds them, and prints the results.

```
#include <xmmintrin.h>

void TestAddSSE()
{
    alignas(16) float A[4];
    alignas(16) float B[4] = { 2.0f, 4.0f, 6.0f, 8.0f };

    // set a = (1, 2, 3, 4) from literal values, and
    // load b = (2, 4, 6, 8) from a floating-point array
    // just to illustrate the two ways of doing this
    // (remember that _mm_set_ps() is backwards!)
    __m128 a = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f);
    __m128 b = _mm_load_ps(&B[0]);

    // add the vectors
    __m128 r = _mm_add_ps(a, b);

    // store '__m128 a' into a float array for printing
    _mm_store_ps(&A[0], a);

    // store result into a float array for printing
    alignas(16) float R[4];
    _mm_store_ps(&R[0], r);

    // inspect results
    printf("a = %.1f %.1f %.1f %.1f\n",
           A[0], A[1], A[2], A[3]);
    printf("b = %.1f %.1f %.1f %.1f\n",
           B[0], B[1], B[2], B[3]);
    printf("r = %.1f %.1f %.1f %.1f\n",
           R[0], R[1], R[2], R[3]);
}
```

4.10.1.5 AltiVec 向量类型

顺便提一下，GNU C/C++ 编译器 gcc（例如，用于编译 PS3 的代码）支持 PowerPC 的 AltiVec 指令集，该指令集支持 SIMD 操作，就像 SSE 在英特尔处理器上所做的那样。128 位矢量类型可以像常规 C/C++ 类型一样声明，但它们前面有关键字 vector 。例如，包含四个 float 的 SIMD 变量将被声明为 vector float 。gcc 还提供了一种将文字 SIMD 值写入源代码的方法。例如，您可以使用如下值初始化矢量浮点数：

```
vector float v = (vector float)(-1.0f, 2.0f, 0.5f, 1.0f);
```

相应的 Visual Studio 代码稍微笨重一些：

```
// use compiler intrinsic to load "literal" value
// (remember _mm_set_ps() is backwards!)
__m128 v = _mm_set_ps(1.0f, 0.5f, 2.0f, -1.0f);
```

我们不会在本章中明确介绍 AltiVec，但一旦您理解了 SSE，学习起来就会非常容易。

4.10.2 使用 SSE 对循环进行矢量化

SIMD 可以将某些类型的计算速度提高四倍，因为一条机器语言指令可以并行对四个浮点值执行算术运算。让我们看看如何使用 SSE 内部函数来实现这一点。

首先，考虑一个简单的循环，它将两个 float 数组成对添加，并将每个结果存储到输出数组中：

```
void AddArrays_ref(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB,
{
    for (int i = 0; i < count; ++i)
    {
        results[i] = dataA[i] + dataB[i];
    }
}
```

我们可以使用 SSE 内部函数显著加快此循环的速度，如下所示：

4.10.1.5 AltiVec `vector` Types

As a quick aside, the GNU C/C++ compiler *gcc* (used to compile code for the PS3, for example) provides support for the PowerPC's *AltiVec* instruction set, which provides support for SIMD operations, much as SSE does on Intel processors. 128-bit vector types can be declared like regular C/C++ types, but they are preceded by the keyword `vector`. For example, a SIMD variable containing four `floats` would be declared `vector float`. gcc also provides a means of writing *literal* SIMD values into your source code. For example, you can initialize a `vector float` with a value like this:

```
vector float v = (vector float)(-1.0f, 2.0f, 0.5f, 1.0f);
```

The corresponding Visual Studio code is a tad more clunky:

```
// use compiler intrinsic to load "literal" value
// (remember _mm_set_ps() is backwards!)
__m128 v = _mm_set_ps(1.0f, 0.5f, 2.0f, -1.0f);
```

We won't cover AltiVec explicitly in this chapter, but once you understand SSE it'll be very easy to learn.

4.10.2 Using SSE to Vectorize a Loop

SIMD offers the potential to speed up certain kinds of calculations by a factor of four, because a single machine language instruction can perform an arithmetic operation on four floating-point values in parallel. Let's see how this can be done using SSE intrinsics.

First, consider a simple loop that adds two arrays of `floats` pairwise, storing each result into an output array:

```
void AddArrays_ref(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB,
{
    for (int i = 0; i < count; ++i)
    {
        results[i] = dataA[i] + dataB[i];
    }
}
```

We can speed up this loop significantly using SSE intrinsics, like this:

```
void AddArrays_sse(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB)
{
    // NOTE: the caller needs to ensure that the size of
    // all 3 arrays are equal, and a multiple of four!
    assert(count % 4 == 0);

    for (int i = 0; i < count; i += 4)
    {
        __m128 a = _mm_load_ps(&dataA[i]);
        __m128 b = _mm_load_ps(&dataB[i]);
        __m128 r = _mm_add_ps(a, b);
        _mm_store_ps(&results[i], r);
    }
}
```

在这个版本中，我们每次循环四个值。我们将四个浮点数块加载到 SSE 寄存器中，并行添加它们，然后将结果存储到结果数组中相应的四个浮点数块中。这称为循环向量化。（在本例中，我们假设三个数组的大小相等，并且大小是 4 的倍数。调用者负责根据需要为每个数组填充一个、两个或三个虚拟值，以满足此要求。）

矢量化可以显著提升速度。本例的速度并没有快到四倍，因为每次迭代都需要以四个一组的方式加载值并存储结果，这带来了开销；但当我测量这些函数在非常大的浮点数组上运行的效果时，非矢量化循环的执行时间大约是矢量化循环的 3.8 倍。

4.10.3 矢量化点积

让我们将矢量化应用到一个更有趣的任务上：计算点积。给定两个四元素向量数组，目标是成对计算它们的点积，并将结果存储到一个浮点型输出数组中。

以下是未使用 SSE 的参考实现。在此实现中，输入数组 a[] 和 b[] 中每个连续的四个 float 值块都被解释为一个同构的四元素向量。

```
void DotArrays_ref(int count,
                   float r[],
                   const float a[],
```



```
void AddArrays_sse(int count,
                   float* results,
                   const float* dataA,
                   const float* dataB)
{
    // NOTE: the caller needs to ensure that the size of
    // all 3 arrays are equal, and a multiple of four!
    assert(count % 4 == 0);

    for (int i = 0; i < count; i += 4)
    {
        __m128 a = _mm_load_ps(&dataA[i]);
        __m128 b = _mm_load_ps(&dataB[i]);
        __m128 r = _mm_add_ps(a, b);
        _mm_store_ps(&results[i], r);
    }
}
```


In this version, we loop over the values four at a time. We load blocks of four floats into SSE registers, add them in parallel, and store the results into a corresponding block of four floats within the result array. This is called *vectorizing* our loop. (In this example, we're assuming that the sizes of the three arrays are equal, and that the size is a multiple of four. The caller is responsible for *padding* the arrays with one, two or three dummy values each, as necessary, in order to meet this requirement.)

Vectorization can lead to a significant speed improvement. This particular example isn't quite four times faster, due to the overhead of having to load the values in groups of four and store the results on each iteration; but when I measured these functions running on very large arrays of floats, the non-vectorized loop took roughly 3.8 times as long to do its work as the vectorized loop.

4.10.3 Vectorized Dot Products

Let's apply vectorization to a somewhat more interesting task: Calculating dot products. Given two arrays of four-element vectors, the goal is to calculate their dot products pairwise, and store the results into an output array of floats.

Here's a reference implementation without using SSE. In this implementation, each contiguous block of four `floats` within the `a[]` and `b[]` input arrays is interpreted as one homogeneous four-element vector.

```
void DotArrays_ref(int count,
                   float r[],
                   const float a[],
```

```
                    const float b[])
{
    for (int i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const int j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

4.10.3.1 第一次尝试（比较慢）

这是首次尝试使用 SSE 内部函数完成此任务：

```
void DotArrays_sse_horizontal(int count,
                              float r[],
                              const float a[],
                              const float b[])
{
    for (int i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const int j = i * 4;

        __m128 va = _mm_load_ps(&a[j]); // ax,ay,az,aw
        __m128 vb = _mm_load_ps(&b[j]); // bx,by,bz,bw

        __m128 v0 = _mm_mul_ps(va, vb);

        // add across the register...
        __m128 v1 = _mm_hadd_ps(v0, v0);
        // (v0w+v0z, v0y+v0x, v0w+v0z, v0y+v0x)
        __m128 vr = _mm_hadd_ps(v1, v1);
        // (v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x,
        //  v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x)

        _mm_store_ss(&r[i], vr); // extract vr.x as a float
    }
}
```

此实现需要一条新指令：_mm_hadd_ps()（水平


```
                    const float b[])
{
    for (int i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const int j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

4.10.3.1 A First Attempt (That's Slow)

Here's a first attempt at using SSE intrinsics for this task:

```
void DotArrays_sse_horizontal(int count,
                              float r[],
                              const float a[],
                              const float b[])
{
    for (int i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const int j = i * 4;

        __m128 va = _mm_load_ps(&a[j]); // ax,ay,az,aw
        __m128 vb = _mm_load_ps(&b[j]); // bx,by,bz,bw

        __m128 v0 = _mm_mul_ps(va, vb);

        // add across the register...
        __m128 v1 = _mm_hadd_ps(v0, v0);
        // (v0w+v0z, v0y+v0x, v0w+v0z, v0y+v0x)
        __m128 vr = _mm_hadd_ps(v1, v1);
        // (v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x,
        //  v0w+v0z+v0y+v0x, v0w+v0z+v0y+v0x)

        _mm_store_ss(&r[i], vr); // extract vr.x as a float
    }
}
```

This implementation required a new instruction: `_mm_hadd_ps()` (*horizontal*

加法）。此内在函数对单个输入寄存器（x，y，z，w）进行操作，并计算两个和：s = x + y 和 t = z + w。它将这两个和存储到目标寄存器的四个槽中，形式为（t，s，t，s）。执行此操作两次，我们可以计算出和 d = x + y + z + w。这称为跨寄存器加法。

跨寄存器加法通常不是一个好主意，因为这是一个非常慢的操作。对 DotArrays_sse() 实现进行性能分析表明，它实际上比参考实现花费的时间略长。在这里使用 SSE 实际上减慢了我们的速度！12

4.10.3.2 更好的方法

实现 SIMD 并行点积计算能力的关键在于找到一种避免跨寄存器加法的方法。这可以实现，但我们必须先转置输入向量。通过将它们按转置顺序存储，我们可以像使用 float 时一样计算点积：我们将 x 分量相乘，然后将结果与 y 分量的乘积相加，然后将结果与 z 分量的乘积相加，最后将结果与 w 分量的乘积相加。

代码如下：

```
void DotArrays_sse(int count,
                   float r[],
                   const float a[],
                   const float b[])
{
  for (int i = 0; i < count; i += 4)
  {
    __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,4,8,12]
    __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[1,5,9,13]
    __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[2,6,10,14]
    __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[3,7,11,15]

    __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,4,8,12]
    __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[1,5,9,13]
    __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[2,6,10,14]
    __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[3,7,11,15]

    __m128 result;
    result = _mm_mul_ps(vaX, vbX);
    result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
    result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
```

12 在 SSE4 中，英特尔引入了内部函数 _mm_dp_ps ()（以及相应的 dpps 指令），它计算点积的延迟比涉及两次调用 _mm_hadd_ps () 的版本略低。但所有水平加法都非常昂贵，应尽可能避免。

add). This intrinsic operates on a single input register (x, y, z, w) and calculates two sums: $s = x + y$ and $t = z + w$. It stores these two sums into the four slots of the destination register as (t, s, t, s). Performing this operation twice allows us to calculate the sum $d = x + y + z + w$. This is called *adding across a register*.

Adding across a register is not usually a good idea because it's a very *slow* operation. Profiling the `DotArrays_sse()` implementation shows that it actually takes a little bit *more* time than the reference implementation. Using SSE here has actually slowed us down![12]

4.10.3.2 A Better Approach

The key to realizing the power of SIMD parallelism for dot products is to figure out a way to avoid having to add across a register. This can be done, but we'll have to *transpose* our input vectors first. By storing them in transposed order, we can calculate our dot product in just the same way that we calculated it when using `floats`: We multiply the x components, then add that result to the product of the y components, then add that result to the product of the z components, and finally add that result to the product of the w components. Here's what the code looks like:

```
void DotArrays_sse(int count,
                   float r[],
                   const float a[],
                   const float b[])
{
  for (int i = 0; i < count; i += 4)
  {
    __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,4,8,12]
    __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[1,5,9,13]
    __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[2,6,10,14]
    __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[3,7,11,15]

    __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,4,8,12]
    __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[1,5,9,13]
    __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[2,6,10,14]
    __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[3,7,11,15]

    __m128 result;
    result = _mm_mul_ps(vaX, vbX);
    result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
    result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
```

[12] With SSE4, Intel introduced the intrinsic `_mm_dp_ps()` (and the corresponding `dpps` instruction) which calculates a dot product with somewhat lower latency than the version involving two invocations of `_mm_hadd_ps()`. But all horizontal adds are very expensive, and should be avoided wherever possible.

```
      result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));

      _mm_store_ps(&r[i], result);
    }
  }
```

MADD指令

有趣的是，乘法后接加法是一种非常常见的运算，它甚至有自己的名字——madd。有些 CPU 提供一条 SIMD 指令来执行 madd 运算。例如，PowerPC AltiVec 的内在函数 vec_madd() 就执行此运算。因此，在 AltiVec 中，我们的 DotArrays() 函数的核心可以稍微简化一些：

```
vector float result = vec_mul(vaX, vbX);
result = vec_madd(vaY, vbY, result));
result = vec_madd(vaZ, vbZ, result));
result = vec_madd(vaW, vbW, result));
```

4.10.3.3 边走边转调

上述实现假设输入数据已被调用者转置。换句话说，假设 a[] 数组包含以下元素：{a0, a4, a8, a12, a1, a5, a9, a13, ...}，b[] 数组亦是如此。如果我们想要操作与参考实现格式相同的输入数据，则必须在函数中进行转置。

方法如下：

```
void DotArrays_sse_transpose(int count,
                             float r[],
                             const float a[],
                             const float b[])
{
  for (int i = 0; i < count; i += 4)
  {
    __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,1,2,3]
    __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[4,5,6,7]
    __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[8,9,10,11]
    __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[12,13,14,15]

    __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,1,2,3]
    __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[4,5,6,7]
    __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[8,9,10,11]
    __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[12,13,14,15]
```



```
      result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));

      _mm_store_ps(&r[i], result);
    }
  }
```

The MADD Instruction

It's interesting to note that a multiply followed by an add is such a common operation that it has its own name—*madd*. Some CPUs provide a single SIMD instruction for performing a *madd* operation. For example, the PowerPC AltiVec intrinsic `vec_madd()` performs this operation. So in AltiVec, the guts of our `DotArrays()` function could be made just a little bit simpler:

```
vector float result = vec_mul(vaX, vbX);
result = vec_madd(vaY, vbY, result));
result = vec_madd(vaZ, vbZ, result));
result = vec_madd(vaW, vbW, result));
```

4.10.3.3 Transpose as We Go

The above implementation *assumes* that the input data has already been transposed by the caller. In other words, the `a[]` array is assumed to contain the components $\{a_0, a_4, a_8, a_{12}, a_1, a_5, a_9, a_{13}, ...\}$ and likewise for the `b[]` array. If we want to operate on input data that's in the same format as it was for the reference implementation, we'll have to do the transposition within our function. Here's how:

```
void DotArrays_sse_transpose(int count,
                             float r[],
                             const float a[],
                             const float b[])
{
  for (int i = 0; i < count; i += 4)
  {
    __m128 vaX = _mm_load_ps(&a[(i+0)*4]); // a[0,1,2,3]
    __m128 vaY = _mm_load_ps(&a[(i+1)*4]); // a[4,5,6,7]
    __m128 vaZ = _mm_load_ps(&a[(i+2)*4]); // a[8,9,10,11]
    __m128 vaW = _mm_load_ps(&a[(i+3)*4]); // a[12,13,14,15]

    __m128 vbX = _mm_load_ps(&b[(i+0)*4]); // b[0,1,2,3]
    __m128 vbY = _mm_load_ps(&b[(i+1)*4]); // b[4,5,6,7]
    __m128 vbZ = _mm_load_ps(&b[(i+2)*4]); // b[8,9,10,11]
    __m128 vbW = _mm_load_ps(&b[(i+3)*4]); // b[12,13,14,15]
```

```
    _MM_TRANSPOSE4_PS(vaX, vaY, vaZ, vaW);
                                      // vaX = a[0,4,8,12]
                                      // vaY = a[1,5,9,13]
                                      // ...
    _MM_TRANSPOSE4_PS(vbX, vbY, vbZ, vbW);
                                      // vbX = b[0,4,8,12]
                                      // vbY = b[1,5,9,13]
                                      // ...

    __m128 result;
    result = _mm_mul_ps(vaX, vbX);
    result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
    result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
    result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));

    _mm_store_ps(&r[i], result);
  }
}
```

两次调用 _MM_TRANSPOSE() 实际上是调用一个略微复杂的宏，该宏使用 shuffle 指令来移动四个输入寄存器的分量。值得庆幸的是，shuffle 操作的开销并不大，因此在计算点积时转置向量并不会引入太多开销。对 DotArrays() 的三种实现进行性能分析后发现，我们的最终版本（即在计算过程中转置向量的版本）比参考实现大约快 3.5 倍。

4.10.3.4 随机播放和转置

对于好奇的读者，_MM_TRANSPOSE() 宏如下所示：

```
#define _MM_TRANSPOSE4_PS(row0, row1, row2, row3)   \
  { __m128 tmp3, tmp2, tmp1, tmp0;                  \
                                                    \
    tmp0   = _mm_shuffle_ps((row0), (row1), 0x44);  \
    tmp2   = _mm_shuffle_ps((row0), (row1), 0xEE);  \
    tmp1   = _mm_shuffle_ps((row2), (row3), 0x44);  \
    tmp3   = _mm_shuffle_ps((row2), (row3), 0xEE);  \
                                                    \
    (row0) = _mm_shuffle_ps(tmp0, tmp1, 0x88);      \
    (row1) = _mm_shuffle_ps(tmp0, tmp1, 0xDD);      \
    (row2) = _mm_shuffle_ps(tmp2, tmp3, 0x88);      \
    (row3) = _mm_shuffle_ps(tmp2, tmp3, 0xDD); }
```

这些疯狂的十六进制数字是按位打包的四元素字段，称为


```
    _MM_TRANSPOSE4_PS(vaX, vaY, vaZ, vaW);
                                      // vaX = a[0,4,8,12]
                                      // vaY = a[1,5,9,13]
                                      // ...
    _MM_TRANSPOSE4_PS(vbX, vbY, vbZ, vbW);
                                      // vbX = b[0,4,8,12]
                                      // vbY = b[1,5,9,13]
                                      // ...

    __m128 result;
    result = _mm_mul_ps(vaX, vbX);
    result = _mm_add_ps(result, _mm_mul_ps(vaY, vbY));
    result = _mm_add_ps(result, _mm_mul_ps(vaZ, vbZ));
    result = _mm_add_ps(result, _mm_mul_ps(vaW, vbW));

    _mm_store_ps(&r[i], result);
  }
}
```

Those two calls to `_MM_TRANSPOSE()` are actually invocations of a somewhat complex macro that uses *shuffle* instructions to move the components of the four input registers around. Thankfully shuffling isn't a particularly expensive operation, so transposing our vectors as we calculate the dot products doesn't introduce too much overhead. Profiling all three implementations of `DotArrays()` shows that our final version (the one that transposes the vectors as it goes) is roughly 3.5 times faster than the reference implementation.

4.10.3.4 Shuffle and Transpose

For the curious reader, here's what the `_MM_TRANSPOSE()` macro looks like:

```
#define _MM_TRANSPOSE4_PS(row0, row1, row2, row3)   \
  { __m128 tmp3, tmp2, tmp1, tmp0;                  \
                                                    \
    tmp0   = _mm_shuffle_ps((row0), (row1), 0x44);  \
    tmp2   = _mm_shuffle_ps((row0), (row1), 0xEE);  \
    tmp1   = _mm_shuffle_ps((row2), (row3), 0x44);  \
    tmp3   = _mm_shuffle_ps((row2), (row3), 0xEE);  \
                                                    \
    (row0) = _mm_shuffle_ps(tmp0, tmp1, 0x88);      \
    (row1) = _mm_shuffle_ps(tmp0, tmp1, 0xDD);      \
    (row2) = _mm_shuffle_ps(tmp2, tmp3, 0x88);      \
    (row3) = _mm_shuffle_ps(tmp2, tmp3, 0xDD); }
```

Those crazy hexadecimal numbers are bit-packed four-element fields called

洗牌掩码。它们告诉 _mm_shuffle() 内部函数如何精确地洗牌组件。这些位封装字段经常引起混淆，可能是因为大多数文档中使用的命名约定。但实际上它非常简单：洗牌掩码由四个整数构成，每个整数代表 SSE 寄存器的一个组件（因此可以取 0 到 3 之间的值）。

```
#define SHUFMASK(p,q,r,s) \
                (p | (q<<2) | (r<<4) | (s<<6))
```

将两个 SSE 寄存器 a 和 b 以及一个混洗掩码传递给 _mm_shuffle_ps() 内部函数会导致 a 和 b 的指定组件出现在输出寄存器 r 中，如下所示：

```
__m128 a = ...;
__m128 b = ...;
__m128 r = _mm_shuffle_ps(a, b,
                          SHUFMASK(p,q,r,s));
// r == ( a[p], a[q], b[r], b[s] )
```

4.10.4 使用 SSE 进行向量矩阵乘法

现在我们了解了如何进行点积运算，我们可以将一个四元素向量与一个 4 × 4 矩阵相乘。为此，我们只需对输入向量和输入矩阵的四行分别进行四次点积运算即可。

我们首先定义一个 Mat44 类，它封装了四个 SSE 向量，分别代表矩阵的四行。我们将使用联合体，以便能够轻松地以 float 类型访问矩阵的各个成员。（之所以能做到这一点，是因为 Mat44 类的实例始终驻留在内存中，而不是直接存储在 SSE 寄存器中。）

```
union Mat44
{
    float  c[4][4]; // components
    __m128 row[4];  // rows
};
```

将向量和矩阵相乘的函数如下所示：

```
__m128 MulVecMat_sse(const __m128& v, const Mat44& M)
{
    // first transpose v
    __m128 vX = _mm_shuffle_ps(v, v, 0x00); // (vx,vx,vx,vx)
```


shuffle masks. They tell the `_mm_shuffle()` intrinsic how exactly to shuffle the components. These bit-packed fields are a common source of confusion, possibly because of the naming conventions used in most documentation. But it's actually pretty simple: A shuffle mask is constructed out of four integers, each of which represents one of the components of an SSE register (and hence can have a value between 0 and 3).

```
#define SHUFMASK(p,q,r,s) \
                (p | (q<<2) | (r<<4) | (s<<6))
```

Passing two SSE registers `a` and `b` along with a shuffle mask to the `_mm_shuffle_ps()` intrinsic results in the specified components of `a` and `b` appearing in the output register `r` as follows:

```
__m128 a = ...;
__m128 b = ...;
__m128 r = _mm_shuffle_ps(a, b,
                          SHUFMASK(p,q,r,s));
// r == ( a[p], a[q], b[r], b[s] )
```

4.10.4 Vector-Matrix Multiplication with SSE

Now that we understand how to perform a dot product, we can multiply a four-element vector with a 4 × 4 matrix. To do it, we simply need to perform four dot products between the input vector and each of the four rows of the input matrix.

We'll start by defining a `Mat44` class that encapsulates four SSE vectors, representing its four rows. We'll use a `union` so that we can easily access the individual members of the matrix as `floats`. (This works because instances of our `Mat44` class will always reside in memory, never directly in SSE registers.)

```
union Mat44
{
    float  c[4][4]; // components
    __m128 row[4];  // rows
};
```

The function to multiply a vector and a matrix looks like this:

```
__m128 MulVecMat_sse(const __m128& v, const Mat44& M)
{
    // first transpose v
    __m128 vX = _mm_shuffle_ps(v, v, 0x00); // (vx,vx,vx,vx)
```

```
    __m128 vY = _mm_shuffle_ps(v, v, 0x55); // (vy,vy,vy,vy)
    __m128 vZ = _mm_shuffle_ps(v, v, 0xAA); // (vz,vz,vz,vz)
    __m128 vW = _mm_shuffle_ps(v, v, 0xFF); // (vw,vw,vw,vw)

    __m128 r =        _mm_mul_ps(vX, M.row[0]);
    r = _mm_add_ps(r, _mm_mul_ps(vY, M.row[1]));
    r = _mm_add_ps(r, _mm_mul_ps(vZ, M.row[2]));
    r = _mm_add_ps(r, _mm_mul_ps(vW, M.row[3]));
    return r;
}
```

这些混洗操作用于将 v 的一个分量（vx、vy、vz 或 vw）复制到 SSE 寄存器的所有四个通道上。这样做的效果是，在对 M 的行（已转置）进行点积运算之前，先对 v 进行转置。（请记住，向量矩阵乘法通常涉及对输入向量和矩阵的列进行点积运算。在这里，我们将 v 转置到四个 SSE 寄存器中，然后对矩阵的行进行逐个分量的运算。）

4.10.5 使用 SSE 进行矩阵-矩阵乘法

一旦我们有了向量和矩阵相乘的函数，用 SSE 内在函数来计算两个 4×4 矩阵的乘积就变得非常简单了。代码如下：

```
void MulMatMat_sse(Mat44& R, const Mat44& A, const Mat44& B)
{
    R.row[0] = MulVecMat_sse(A.row[0], B);
    R.row[1] = MulVecMat_sse(A.row[1], B);
    R.row[2] = MulVecMat_sse(A.row[2], B);
    R.row[3] = MulVecMat_sse(A.row[3], B);
}
```

4.10.6 广义矢量化

由于 SSE 寄存器包含四个浮点值，人们很容易认为它与四元素齐次向量 v 的分量“完美契合”，并认为 SSE 的最佳用途是进行 3D 向量运算。然而，这种看待 SIMD 并行性的方式非常局限。

大多数“批量”操作（即在大型数据集上重复执行单个计算）都可以使用 SIMD 并行性进行矢量化。仔细想想，SIMD 寄存器的组件实际上就像并行的“通道”，可以在其中执行任意处理。使用浮点数


```
    __m128 vY = _mm_shuffle_ps(v, v, 0x55); // (vy,vy,vy,vy)
    __m128 vZ = _mm_shuffle_ps(v, v, 0xAA); // (vz,vz,vz,vz)
    __m128 vW = _mm_shuffle_ps(v, v, 0xFF); // (vw,vw,vw,vw)

    __m128 r =        _mm_mul_ps(vX, M.row[0]);
    r = _mm_add_ps(r, _mm_mul_ps(vY, M.row[1]));
    r = _mm_add_ps(r, _mm_mul_ps(vZ, M.row[2]));
    r = _mm_add_ps(r, _mm_mul_ps(vW, M.row[3]));
    return r;
}
```


The shuffles are used to replicate *one* component of **v** (either v_x, v_y, v_z or v_w) across *all four* lanes of an SSE register. This has the effect of transposing **v** prior to performing the dot product with the rows of **M**, which are already transposed. (Remember that vector-matrix multiplication normally involves taking dot products between an input vector and the *columns* of the matrix. Here, we're transposing **v** into four SSE registers, and then doing our math, component-wise, with the *rows* of the matrix.)

4.10.5 Matrix-Matrix Multiplication with SSE

Multiplying two 4×4 matrices with SSE intrinsics is trivial once we have a function to multiply a vector and a matrix. Here's what the code looks like:


```
void MulMatMat_sse(Mat44& R, const Mat44& A, const Mat44& B)
{
    R.row[0] = MulVecMat_sse(A.row[0], B);
    R.row[1] = MulVecMat_sse(A.row[1], B);
    R.row[2] = MulVecMat_sse(A.row[2], B);
    R.row[3] = MulVecMat_sse(A.row[3], B);
}
```


4.10.6 Generalized Vectorizaton

Because an SSE register contains four floating-point values, it's tempting to think of it as a natural "fit" for the components of a four-element homogeneous vector **v**, and to think that the best use of SSE is for doing 3D vector math. However, this is a very limiting way of thinking about SIMD parallelism.

Most "batched" operations, in which a single computation is performed repeatedly on a large dataset, can be vectorized using SIMD parallelism. If you think about it, the components of a SIMD register really function like parallel "lanes" in which arbitrary processing can be performed. Working with `float`

变量只给我们提供了一条通道，但使用 128 位（四元素）SIMD 变量，我们可以在四条通道上并行执行相同的计算——换句话说，我们可以一次执行四次计算。使用 256 位 AVX 寄存器，我们可以获得八条通道，让我们一次可以执行八次计算。而 AVX-512 则提供了 16 条通道，让我们可以一次进行 16 次计算。

编写矢量化代码最简单的方法是先将其编写为单通道算法（仅使用 float 类型）。一旦它运行正常，我们就可以将其转换为一次操作 N 个元素，使用具有 N 通道容量的 SIMD 寄存器。我们已经看到了这个过程的实际应用：在 4.10.3 节中，我们首先编写了一个循环，每次执行一个点积运算，然后我们转换为使用 SSE，这样就可以一次执行四个点积运算。

以这种方式矢量化代码的一个好处是，您只需对代码进行少量调整，即可享受更广泛 SIMD 硬件的优势。在仅支持 SSE 的机器上，循环每次迭代执行四次操作；在支持 AVX 的机器上，只需将其更改为每次迭代执行八次操作；而在 AVX-512 系统上，每次迭代可以执行 16 次操作。

有趣的是，大多数优化编译器可以自动矢量化某些类型的单通道循环。事实上，在编写上述示例时，我花了一些功夫才强制编译器不矢量化我的单通道代码，以便将其性能与我的 SIMD 实现进行比较！再次强调，在编写优化代码时，查看反汇编代码总是一个好主意——你可能会发现编译器执行的操作比你想象的要多（或少）！

4.10.7 向量预测

让我们看另一个（完全是人为设计的）例子。这个例子不仅会强化广义向量化的思想，还能用来说明另一种有用的技巧：向量预测。

想象一下，我们需要对大量浮点数数组求平方根。我们首先将其写成单车道循环，如下所示：

```
#include <cmath>

void SqrtArray_ref(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    for (unsigned i = 0; i < count; ++i)
    {
```


variables gives us a single lane, but working with 128-bit (four-element) SIMD variables allows us to do that same calculation in parallel across four lanes—in other words, we can perform our calculations four at a time. Working with 256-bit AVX registers gives us eight lanes, allowing us to perform our calculations eight at a time. And AVX-512 gives us 16 lanes, letting us do 16 calculations at a time.

The easiest way to write *vectorized* code is to start out by writing it as a single-lane algorithm (just using `floats`). Once it works, we can convert it to operate *N* elements at a time, using SIMD registers that have an *N*-lane capacity. We've already seen this process in action: In Section 4.10.3, we first wrote a loop that performed a large batch of dot products one at a time, and then we converted to use SSE so that we could perform those dot products four at a time.

One nice side-effect of vectorizing your code in this way is that you can reap the benefits of wider SIMD hardware with little adjustment to your code. On a machine with only SSE support, you perform four operations per iteration of your loop; on a machine that supports AVX, you simply change it to do eight operations per iteration; and on an AVX-512 system, you can do 16 operations per iteration.

Interestingly, most optimizing compilers can *vectorize* some kinds of single-lane loops automatically. In fact, when writing the above examples, it took some doing to force the compiler *not* to vectorize my single-lane code, so that I could compare its performance to my SIMD implementation! Once again, it's always a good idea to look at the disassembly when writing optimized code—you may discover the compiler is doing more (or less) than you thought!

4.10.7 Vector Predication

Let's take a look at another (totally contrived) example. This example will reinforce the ideas of generalized vectorization, but it will also serve to illustrate another useful technique: *vector predication*.

Imagine that we needed to take the square roots of a large array of `floats`. We'd start out by writing it as a single-lane loop, like this:

```
#include <cmath>

void SqrtArray_ref(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    for (unsigned i = 0; i < count; ++i)
    {
```

```
        if (a[i] >= 0.0f)
            r[i] = std::sqrt(a[i]);
        else
            r[i] = 0.0f;
    }
}
```

接下来，让我们将此循环转换为 SSE，一次执行四个平方根：

```
#include <xmmintrin.h>

void SqrtArray_sse_broken(float* __restrict__ r,
                          const float* __restrict__ a,
                          int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f); // all zeros

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);

        __m128 vr;
        if (_mm_cmpge_ps(va, vz)) // ???
            vr = _mm_sqrt_ps(va);
        else
            vr = vz;

        _mm_store_ps(r + i, vr);
    }
}
```

这看起来很简单：我们只需一次遍历输入数组中的四个浮点数，将四个浮点数组加载到 SSE 寄存器中，然后使用 _mm_sqrt_ps() 执行四个并行平方根。

然而，这个循环中有一个小陷阱。我们需要检查输入值是否大于或等于零，因为负数的平方根是虚数（因此会产生 QNaN）。内部函数 _mm_cmpge_ps() 逐个比较两个 SSE 寄存器的值，以查看它们是否大于或等于调用者提供的值向量。然而，这个函数并不返回 bool 值。它怎么会返回呢？因为我们将四个值与另外四个值进行比较，所以其中一些可能通过测试，而另一些则未通过。这意味着我们不能仅仅对结果进行 if 检查。

of `_mm_cmpge_ps()`.[13]

13 单通道参考实现中的 if 检查也会阻止编译器自动矢量化循环。


```
        if (a[i] >= 0.0f)
            r[i] = std::sqrt(a[i]);
        else
            r[i] = 0.0f;
    }
}
```

Next, let's convert this loop into SSE, performing four square roots at a time:

```
#include <xmmintrin.h>

void SqrtArray_sse_broken(float* __restrict__ r,
                          const float* __restrict__ a,
                          int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f); // all zeros

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);

        __m128 vr;
        if (_mm_cmpge_ps(va, vz)) // ???
            vr = _mm_sqrt_ps(va);
        else
            vr = vz;

        _mm_store_ps(r + i, vr);
    }
}
```

This seems simple enough: We simply stride through the input array four `floats` at a time, load groups of four floats into an SSE register, and then perform four parallel square roots with `_mm_sqrt_ps()`.

However, there's one small gotcha in this loop. We need to check whether the input values are greater than or equal to zero, because the square root of a negative number is imaginary (and will therefore produce QNaN). The intrinsic `_mm_cmpge_ps()` compares the values of two SSE registers, componentwise, to see if they are greater than or equal to a vector of values supplied by the caller. However, this function doesn't return a `bool`. How can it? We're comparing four values to four other values, so some of them may pass the test while others fail it. That means we can't just do an `if` check against the results of `_mm_cmpge_ps()`.[13]

[13]The if check in the single-lane reference implementation also prevents the compiler from automatically vectorizing the loop.

这是否预示着我们的矢量化实现将面临厄运？谢天谢地，并非如此。所有 SSE 比较函数（例如 _mm_cmpge_ps()）都会生成一个四分量结果，并存储在 SSE 寄存器中。但结果并非包含四个浮点值，而是由四个 32 位掩码组成。如果输入寄存器中的相应分量通过测试，则每个掩码全为二进制 1（0xFFFFFFFFU）；如果该分量未通过测试，则每个掩码全为二进制 0（0x0U）。

我们可以将 SSE 比较内在函数的结果用作位掩码，以便在两个可能的结果之间进行选择。在我们的示例中，当输入值通过测试（大于或等于零）时，我们希望选择该输入值的平方根；当输入值未通过测试（为负数）时，我们希望选择零值。这称为 预测 ，由于我们将其应用于 SIMD 向量，因此也称为 向量预测 。

在 4.2.6.2 节中，我们了解了如何使用按位 AND、OR 和 NOT 运算符对浮点值进行预测。以下是该示例的摘录：

```
    // ...

    // select quotient when mask is all ones, or default
    // value d when mask is all zeros (NOTE: this won't
    // work as written -- you'd need to use a union to
    // interpret the floats as unsigned for masking)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

这里没有什么不同，我们只需要使用这些位运算符的 SSE 版本：

```
#include <xmmintrin.h>

void SqrtArray_sse(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f);

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);
```


Does this spell doom for our vectorized implementation? Thankfully not. All SSE comparison intrinsics like `_mm_cmpge_ps()` produce a four-component result, stored in an SSE register. But instead of containing four floating-point values, the result consists of four 32-bit masks. Each mask contains all binary 1s (0xFFFFFFFFU) if the corresponding component in the input register passed the test, and all binary 0s (0x0U) if that component failed the test.

We can use the results of an SSE comparison intrinsic by applying it as a bit mask in order to *select* between one of two possible results. In our example, when the input value passes the test (is greater than or equal to zero), we want to select the square root of that input value; when it fails the test (is negative), we want to select a value of zero. This is called *predication*, and because we're applying it to SIMD vectors, it's called *vector predication*.

In Section 4.2.6.2, we saw how to do predication with floating-point values, using bitwise AND, OR and NOT operators. Here's an excerpt from that example:

```
    // ...

    // select quotient when mask is all ones, or default
    // value d when mask is all zeros (NOTE: this won't
    // work as written -- you'd need to use a union to
    // interpret the floats as unsigned for masking)
    const float result = (q & mask) | (d & ~mask);
    return result;
}
```

It's no different here, we just need to use SSE versions of these bitwise operators:

```
#include <xmmintrin.h>

void SqrtArray_sse(float* __restrict__ r,
                   const float* __restrict__ a,
                   int count)
{
    assert(count % 4 == 0);
    __m128 vz = _mm_set1_ps(0.0f);

    for (int i = 0; i < count; i += 4)
    {
        __m128 va = _mm_load_ps(a + i);
```

```
        // always do the quotient, but it may end
        // up producing QNaN in some or all lanes
        __m128 vq = _mm_sqrt_ps(va);

        // now select between vq and vz, depending
        // on whether the input was greater than
        // or equal to zero or not
        __m128 mask = _mm_cmpge_ps(va, zero);

        // (vq & mask) | (vz & ~mask)
        __m128 qmask = _mm_and_ps(mask, vq);
        __m128 znotmask = _mm_andnot_ps(mask, vz);
        __m128 vr = _mm_or_ps(qmask, znotmask);

        _mm_store_ps(r + i, vr);
    }
}
```

将这种向量预测操作封装在一个函数中非常方便，也很常见，这个函数通常被称为 向量选择 。PowerPC 的 AltiVec ISA 为此提供了一个名为 vec_sel() 的内在函数。它的工作原理如下：

```
// pseudocode illustrating how AltiVec's vec_sel() intrinsic
// works
vector float vec_sel(vector float falseVec,
                     vector float trueVec,
                     vector bool mask)
{
    vector float r;
    for (each lane i)
    {
        if (mask[i] == 0)
            r[i] = falseVec[i];
        else
            r[i] = trueVec[i];
    }
    return r;
}
```

SSE2 没有提供向量选择指令，但值得庆幸的是，SSE4 中引入了一条指令 - 它由内在的 _mm_blendv_ps() 发出。

让我们看看如何自己实现一个向量选择操作。我们可以这样写：


```
        // always do the quotient, but it may end
        // up producing QNaN in some or all lanes
        __m128 vq = _mm_sqrt_ps(va);

        // now select between vq and vz, depending
        // on whether the input was greater than
        // or equal to zero or not
        __m128 mask = _mm_cmpge_ps(va, zero);

        // (vq & mask) | (vz & ~mask)
        __m128 qmask = _mm_and_ps(mask, vq);
        __m128 znotmask = _mm_andnot_ps(mask, vz);
        __m128 vr = _mm_or_ps(qmask, znotmask);

        _mm_store_ps(r + i, vr);
    }
}
```

It's convenient and commonplace to encapsulate this *vector predication* operation in a function, which is typically called *vector select*. PowerPC's AltiVec ISA provides an intrinsic called `vec_sel()` for this purpose. It works like this:

```
// pseudocode illustrating how AltiVec's vec_sel() intrinsic
// works
vector float vec_sel(vector float falseVec,
                     vector float trueVec,
                     vector bool mask)
{
    vector float r;
    for (each lane i)
    {
        if (mask[i] == 0)
            r[i] = falseVec[i];
        else
            r[i] = trueVec[i];
    }
    return r;
}
```

SSE2 provided no vector select instruction, but thankfully one was introduced in SSE4—it is emitted by the intrinsic `_mm_blendv_ps()`.

Let's take a look at how we might implement a vector select operation ourselves. We can write it like this:

```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
    // (b & mask) | (a & ~mask)
    __m128 bmask = _mm_and_ps(mask, b);
    __m128 anotmask = _mm_andnot_ps(mask, a);
    return _mm_or_ps(bmask, anotmask);
}
```

或者如果我们够勇敢的话，我们可以使用异或来完成同样的事情：

```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
    // (((a ^ b) & mask) ^ a)
    __m128 diff = _mm_xor_ps(a, b);
    return _mm_xor_ps(a, _mm_and_ps(mask, diff));
}
```

看看你能不能弄清楚它是怎么运作的。这里有两个提示：异或运算符计算两个值之间的按位差。连续两次异或运算不会改变输入值（`(a ^ b) ^ b == a`）。

4.11 GPGPU编程简介

我们在上一节中提到，如果目标 CPU 包含 SIMD 矢量处理单元，并且源代码满足某些要求（例如不涉及复杂的分支），大多数优化编译器可以自动对部分代码进行矢量化。矢量化也是通用 GPU 编程 (GPGPU) 的支柱之一。在本节中，我们将简要介绍 GPU 在硬件架构方面与 CPU 的区别，以及 SIMD 并行和矢量化的概念如何帮助程序员编写能够在 GPU 上并行处理大量数据的计算着色器。

4.11.1 数据并行计算

GPU 是一种专用协处理器，专门设计用于加速那些需要高度数据并行性的计算。它通过将 SIMD 并行性（矢量化 ALU）与 MIMD 并行性（采用一种抢占式多线程）相结合来实现这一点。NVIDIA 创造了“单指令多线程 (SIMT)”一词来描述这种 SIMD/MIMD 混合设计。


```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
    // (b & mask) | (a & ~mask)
    __m128 bmask = _mm_and_ps(mask, b);
    __m128 anotmask = _mm_andnot_ps(mask, a);
    return _mm_or_ps(bmask, anotmask);
}
```

Or if we're feeling brave, we can accomplish the same thing with exclusive OR:

```
__m128 _mm_select_ps(const __m128 a,
                     const __m128 b,
                     const __m128 mask)
{
    // (((a ^ b) & mask) ^ a)
    __m128 diff = _mm_xor_ps(a, b);
    return _mm_xor_ps(a, _mm_and_ps(mask, diff));
}
```

See if you can figure out how this works. Here are two hints: The exclusive OR operator calculates the *bitwise difference* between two values. Two XORs in a row leave the input value unchanged (`(a ^ b) ^ b == a`).

4.11 Introduction to GPGPU Programming

We said in the preceding section that most optimizing compilers can *vectorize* some code automatically, if the target CPU includes a SIMD vector processing unit, and if the source code meets certain requirements (such as not involving complex branching). Vectorization is also one of the pillars of *general-purpose GPU* programming (GPGPU). In this section, we'll take a brief introductory look at how a GPU differs from a CPU in terms of its hardware architecture, and how the concepts of SIMD parallelism and vectorization enable programmers to write *compute shaders* that are capable of processing large amounts of data in parallel on a GPU.

4.11.1 Data-Parallel Computations

A GPU is a specialized coprocessor designed specifically to accelerate those computations that involve a high degree of *data parallelism*. It does so by combining SIMD parallelism (vectorized ALUs) with MIMD parallelism (by employing a form of preemptive multithreading). NVIDIA coined the term *single instruction multiple thread* (SIMT) to describe this SIMD/MIMD hybrid design,

尽管该设计并非 NVIDIA GPU 独有——尽管 GPU 设计的细节因供应商和产品线而异，但所有 GPU 都在其设计中采用了 SIMT 并行性的一般原则。

GPU 专为在超大规模数据集上执行数据并行计算而设计。为了使计算任务能够很好地在 GPU 上执行，对数据集中任何一个元素执行的计算必须独立于对其他元素的计算结果。

换句话说，必须能够按任意顺序处理元素。
我们从 4.10.3 节开始介绍的 SIMD 矢量化的简单示例都是数据并行计算的例子。回想一下这个函数，它处理两个可能非常大的输入向量数组，并生成一个包含这些向量标量点积的输出数组：

```
void DotArrays_ref(unsigned count,
                   float r[],
                   const float a[],
                   const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

此循环每次迭代执行的计算都独立于其他迭代执行的计算。这意味着我们可以自由地按照我们认为合适的顺序执行计算。此外，此特性使我们能够应用 SSE 或 AVX 内部函数来矢量化循环——我们不再需要一次执行一个计算，而是可以使用 SIMD 并行性同时执行 4、8 或 16 个计算，从而分别将迭代次数减少 4 倍、8 倍或 16 倍。

现在想象一下将 SIMD 并行化推向极致。如果我们有一个拥有 1024 个通道的 SIMD VPU，会怎么样？在这种情况下，我们可以将总迭代次数除以 1024——当输入数组包含 1024 个或更少的元素时，我们实际上可以在一次迭代中执行整个循环！

although the design is not unique to NVIDIA GPUs—although the specifics of GPU designs vary from vendor to vendor and from product line to product line in significant ways, all GPUs employ the general principles of SIMT parallelism in their designs.

GPUs are designed specifically to perform *data-parallel* computations on very large datasets. In order for a computational task to be well-suited to execution on a GPU, the computations performed on any one element of the dataset must be *independent* of the results of computations on other elements. In other words, it must be possible to process the elements in any order.

The simple examples of SIMD vectorization that we presented starting in Section 4.10.3 are all examples of data-parallel computations. Recall this function, which processes two potentially very large arrays of input vectors and produces an output array containing the scalar dot products of those vectors:

```
void DotArrays_ref(unsigned count,
                   float r[],
                   const float a[],
                   const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        // treat each block of four floats as a
        // single four-element vector
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

The computation performed by each iteration of this loop is independent of the computations performed by the other iterations. That means that we are free to perform the computations in any order we see fit. Moreover, this property is what allows us to apply SSE or AVX intrinsics to vectorize the loop—instead of performing the computations one-at-a-time, we can use SIMD parallelism to perform four, eight or 16 computations simultaneously, thereby reducing the iteration count by a factor of four, eight or 16, respectively.

Now imagine taking SIMD parallelization to an extreme. What if we had a SIMD VPU with 1024 lanes? In that case, we could divide the total number of iterations by 1024—and when the input arrays contain 1024 elements or fewer, we could literally execute the entire loop in a single iteration!

粗略地说，这就是 GPU 的工作方式。然而，它使用的并非每个通道宽度为 1024 的 SIMD。GPU 的 SIMD 单元通常宽度为 8 或 16，但它们每次以 32 或 64 个元素为一批处理工作负载。更重要的是，GPU 包含许多这样的 SIMD 单元。因此，大型工作负载可以并行地在这些 SIMD 单元之间进行调度，从而使 GPU 能够并行处理数千个数据元素。

当像素着色器（也称为片段着色器）应用于数百万像素，或顶点着色器应用于数十万甚至数百万个 3D 网格顶点，每帧以 30 或 60 FPS 的速度运行时，数据并行计算正是医生所要求的。但现代 GPU 将其计算能力开放给程序员，使我们能够编写通用计算着色器。只要我们希望在大型数据集上执行的计算具有彼此高度独立的属性，它们在 GPU 上的执行效率就可能高于在 CPU 上。

4.11.2 计算内核

在 4.10.3 节中，我们了解到，为了将 DotArrays_ref() 函数中的循环向量化，我们必须重写代码。该函数的向量化版本利用了 SSE 或 AVX 的内在函数；标量数据类型被替换为向量类型，例如 SSE 的 __m128 或 AltiVec 的向量浮点数；并且循环被硬编码为每次迭代 4 个、8 个或 16 个元素。

在编写 GPGPU 计算着色器时，我们采用了不同的策略。我们不再将循环硬编码为以固定大小的批次运行，而是使用标量数据类型将其保留为“单通道”计算。然后，我们将循环主体提取到一个称为 内核 的单独函数中。上面的例子在内核中看起来如下：

```
void DotKernel(unsigned i,
               float r[],
               const float a[],
               const float b[])
{
    // treat each block of four floats as a
    // single four-element vector
    const unsigned j = i * 4;

    r[i] = a[j+0]*b[j+0]  // ax*bx
         + a[j+1]*b[j+1]  // ay*by
         + a[j+2]*b[j+2]  // az*bz
         + a[j+3]*b[j+3]; // aw*bw
}
```


This is, roughly speaking, what a GPU does. However, it doesn't use SIMDs that are *actually* 1024 lanes wide each. A GPU's *SIMD units* are typically eight or 16 lanes wide, but they process workloads in batches of 32 or 64 elements at a time. What's more, a GPU contains *many* such SIMD units. So a large workload can be dispatched across these SIMD units in parallel, resulting in the GPU being capable of processing literally thousands of data elements in parallel.

Data-parallel computations are just what the doctor ordered when applying a pixel shader (also known as a fragment shader) to millions of pixels, or a vertex shader to hundreds of thousands or even millions of 3D mesh vertices, every frame at 30 or 60 FPS. But modern GPUs expose their computational power to programmers, allowing us to write general-purpose *compute shaders*. As long as the computations we wish to perform on a large dataset have the property of being largely independent of one another, they can probably be executed by a GPU more efficiently than they could be on a CPU.

4.11.2 Compute Kernels

In Section 4.10.3, we saw that in order to vectorize a loop like the one in the `DotArrays_ref()` function, we must rewrite the code. The vectorized version of the function makes use of SSE or AVX intrinsics; our *scalar* data types are replaced by *vector* types such as SSE's `__m128` or AltiVec's `vector float`; and the loop is hard-coded to iterate four, eight or 16 elements at a time.

When writing a GPGPU compute shader, we take a different tack. Instead of hard-coding the loop to operate in fixed-size batches, we leave the loop as a "single-lane" computation using scalar data types. Then, we extract the body of the loop into a separate function known as a *kernel*. Here's what the example above would look like as a kernel:

```
void DotKernel(unsigned i,
               float r[],
               const float a[],
               const float b[])
{
    // treat each block of four floats as a
    // single four-element vector
    const unsigned j = i * 4;

    r[i] = a[j+0]*b[j+0]  // ax*bx
         + a[j+1]*b[j+1]  // ay*by
         + a[j+2]*b[j+2]  // az*bz
         + a[j+3]*b[j+3]; // aw*bw
}
```

```
void DotArrays_gpgpu1(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        DotKernel(i, r, a, b);
    }
}
```

DotKernel() 函数现在的形式适合转换为计算着色器。它只处理输入数据的一个元素，并生成单个输出元素。这类似于像素/片段着色器接收单个输入像素/片段颜色并将其转换为单个输出颜色，或者类似于顶点着色器接收单个输入顶点并生成单个输出顶点。GPU 有效地为我们运行 for 循环，为数据集的每个元素调用一次核函数。

GPGPU 计算内核通常使用一种特殊的着色语言编写，这种语言可以编译成 GPU 可理解的机器码。着色语言的语法通常与 C 语言非常接近，因此将 C 或 C++ 循环转换为 GPU 计算内核通常并非难事。着色语言的例子包括 DirectX 的 HLSL（高级着色器语言）、OpenGL 的 GLSL、NVIDIA 的 Cg 和 CUDA C 语言以及 OpenCL。

某些着色语言要求您将内核代码移至与 C++ 应用程序代码分开的特殊源文件中。然而，OpenCL 和 CUDA C 是 C++ 语言本身的扩展。因此，它们允许程序员将计算内核编写为常规 C/C++ 函数，只需进行少量语法调整，并以相对简单的语法在 GPU 上调用这些内核。

作为一个具体的例子，这是我们用 CUDA C 编写的 DotKernel() 函数：

```
__global__ void DotKernel_CUDA(int count,
                               float* r,
                               const float* a,
                               const float* b)
{
    // CUDA provides a magic "thread index" to each
    // invocation of the kernel -- this serves as
    // our loop index i
```



```
void DotArrays_gpgpu1(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    for (unsigned i = 0; i < count; ++i)
    {
        DotKernel(i, r, a, b);
    }
}
```

The `DotKernel()` function is now in a form that's suitable for conversion into a *compute shader*. It processes just one element of the input data, and produces a single output element. This is analogous to how a pixel/fragment shader receives a single input pixel/fragment color and transforms it into a single output color, or to how a vertex shader receives a single input vertex and produces a single output vertex. The GPU effectively runs the `for` loop for us, calling our kernel function once for each element of our dataset.

GPGPU compute kernels are typically written in a special *shading language* which can be compiled into machine code that's understandable by the GPU. Shading languages are usually very close to C in syntax, so converting a C or C++ loop into a GPU compute kernel isn't usually a particularly difficult undertaking. Some examples of shading languages include DirectX's HLSL (high-level shader language), OpenGL's GLSL, NVIDIA's Cg and CUDA C languages, and OpenCL.

Some shading languages require you to move your kernel code into special source files, separate from your C++ application code. OpenCL and CUDA C, however, are extensions to the C++ language itself. As such, they permit programmers to write compute kernels as regular C/C++ functions, with only minor syntactic adjustments, and to invoke those kernels on the GPU with relatively simple syntax.

As a concrete example, here's our `DotKernel()` function written in CUDA C:

```
__global__ void DotKernel_CUDA(int count,
                               float* r,
                               const float* a,
                               const float* b)
{
    // CUDA provides a magic "thread index" to each
    // invocation of the kernel -- this serves as
    // our loop index i
```

```
    size_t i = threadIdx.x;

    // make sure the index is valid
    if (i < count)
    {
        // treat each block of four floats as a
        // single four-element vector
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

您会注意到，循环索引 i 取自内核函数本身中名为 threadIdx 的变量。线程索引是编译器提供的“魔法”输入，就像 this 指针在 C++ 类成员函数中“神奇地”指向当前实例一样。我们将在下一节中详细讨论线程索引。

4.11.3 执行内核

既然我们已经编写了一个计算内核，让我们看看如何在 GPU 上执行它。具体细节因着色语言而异，但关键概念大致相同。例如，以下是如何在 CUDA C 中启动计算内核：

```
void DotArrays_gpgpu2(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    // allocate "managed" buffers that are visible
    // to both CPU and GPU
    int *cr, *ca, *cb;
    cudaMallocManaged(&cr, count * sizeof(float));
    cudaMallocManaged(&ca, count * sizeof(float) * 4);
    cudaMallocManaged(&cb, count * sizeof(float) * 4);

    // transfer the data into GPU-visible memory
    memcpy(ca, a, count * sizeof(float) * 4);
    memcpy(cb, b, count * sizeof(float) * 4);
```



```
    size_t i = threadIdx.x;

    // make sure the index is valid
    if (i < count)
    {
        // treat each block of four floats as a
        // single four-element vector
        const unsigned j = i * 4;

        r[i] = a[j+0]*b[j+0]  // ax*bx
             + a[j+1]*b[j+1]  // ay*by
             + a[j+2]*b[j+2]  // az*bz
             + a[j+3]*b[j+3]; // aw*bw
    }
}
```

You'll notice that the loop index `i` is taken from a variable called `threadIdx` within the kernel function itself. The thread index is a "magic" input provided by the compiler, in much the same way that the `this` pointer "magically" points to the current instance within a C++ class member function. We'll talk more about the thread index in the next section.

4.11.3 Executing a Kernel

Given that we've written a compute kernel, let's see how to execute it on the GPU. The details differ from shading language to shading language, but the key concepts are roughly equivalent. For example, here's how we'd kick off our compute kernel in CUDA C:

```
void DotArrays_gpgpu2(unsigned count,
                      float r[],
                      const float a[],
                      const float b[])
{
    // allocate "managed" buffers that are visible
    // to both CPU and GPU
    int *cr, *ca, *cb;
    cudaMallocManaged(&cr, count * sizeof(float));
    cudaMallocManaged(&ca, count * sizeof(float) * 4);
    cudaMallocManaged(&cb, count * sizeof(float) * 4);

    // transfer the data into GPU-visible memory
    memcpy(ca, a, count * sizeof(float) * 4);
    memcpy(cb, b, count * sizeof(float) * 4);
```

```
    // run the kernel on the GPU
    DotKernel_CUDA <<<1, count>>> (cr, ca, cb, count);

    // wait for GPU to finish
    cudaDeviceSynchronize();

    // return results and clean up
    memcpy(r, cr, count * sizeof(float));
    cudaFree(cr);
    cudaFree(ca);
    cudaFree(cb);
}
```

需要一些设置代码来将输入和输出缓冲区分配为 CPU 和 GPU 可见的“托管”内存，并将输入数据复制到其中。然后，CUDA 特有的三重尖括号符号（<<<G, N>>>）通过向驱动程序提交请求，在 GPU 上执行计算内核。调用 cudaDeviceSynchronize() 会强制 CPU 等待 GPU 完成其工作（这与 pthread_join() 强制一个线程等待另一个线程完成的方式非常相似）。

最后，我们释放 GPU 可见的数据缓冲区。

让我们仔细看看 <<<G,N>>> 尖括号表示法。尖括号内的第二个参数 N 允许我们指定输入数据的维度。这对应于我们希望 GPU 执行的循环迭代次数。它实际上可以是一维、二维或三维的量，允许我们处理一维、二维或三维输入数组。但是，就像在 C/C++ 中一样，多维数组实际上只是以特殊方式索引的一维数组。例如，在 C/C++ 中，像 [row][column] 这样的二维数组访问实际上等同于 [row*numColumns + column] 这样的一维数组访问。同样的原理也适用于多维 GPU 缓冲区。

尖括号中的第一个参数 G 告诉驱动程序在运行此计算内核时要使用多少个线程组（在 NVIDIA 术语中称为线程块）。具有单个线程组的计算作业被限制在 GPU 上的单个计算单元上运行。计算单元本质上是 GPU 中的一个核心——一个能够执行指令流的硬件组件。为 G 传递一个较大的数字可以让驱动程序将工作负载划分到多个计算单元上。

4.11.4 GPU线程和线程组

G 参数告诉 GPU 驱动程序将工作划分到多少个线程组中。正如你所料，一个线程组由一定数量的


```
    // run the kernel on the GPU
    DotKernel_CUDA <<<1, count>>> (cr, ca, cb, count);

    // wait for GPU to finish
    cudaDeviceSynchronize();

    // return results and clean up
    memcpy(r, cr, count * sizeof(float));
    cudaFree(cr);
    cudaFree(ca);
    cudaFree(cb);
}
```

A bit of set-up code is required to allocate the input and output buffers as “managed” memory that is visible to both CPU and GPU, and to copy the input data into them. The CUDA-specific triple angled brackets notation (`<<<G,N>>>`) then executes the compute kernel on the GPU, by submitting a request to the driver. The call to `cudaDeviceSynchronize()` forces the CPU to wait until the GPU has done its work (in much the same way that `pthread_join()` forces one thread to wait for the completion of another). Finally, we free the GPU-visible data buffers.

Let’s have a closer look at the `<<<G,N>>>` angled bracket notation. The second argument `N` within the angled brackets allows us to specify the *dimensions* of our input data. This corresponds to the number of *iterations* of our loop that we want the GPU to perform. It can actually be a one-, two- or three-dimensional quantity, allowing us to process one-, two- or three-dimensional input arrays. However, just like in C/C++, a multidimensional array is really just a one-dimensional array that’s indexed in a special way. For example, in C/C++ a two-dimensional array access written like `[row][column]` is really equivalent to a one-dimensional array access `[row*numColumns + column]`. The same principle applies to multidimensional GPU buffers.

The first argument `G` in the angled brackets tells the driver how many *thread groups* (known as *thread blocks* in NVIDIA terminology) to use when running this compute kernel. A compute job with a single thread group is constrained to run on a single *compute unit* on the GPU. A compute unit is esssentially a *core* within the GPU—a hardware component that is capable of executing an instruction stream. Passing a larger number for `G` allows the driver to divide the workload up across more than one compute unit.

4.11.4 GPU Threads and Thread Groups

The `G` argument tells the GPU driver into how many *thread groups* to divide our work. As you might expect, a thread group is comprised of some number

图 4.39。典型的 GPU 由多个计算单元 (CU) 组成，每个计算单元就像一个精简的 CPU 核心。一个 CU 包含一个指令提取/解码单元、一个 L1 缓存、可能还有一块本地数据存储 RAM (LDS)、一个标量 ALU 以及多个用于执行矢量化代码的 SIMD 单元。

GPU 线程。但是在 GPU 的上下文中，“线程”一词到底是什么意思呢?

每个 GPU 内核都会被编译成由一系列 GPU 机器语言指令组成的指令流，其方式与 C/C++ 函数被编译成 CPU 指令流的方式大致相同。因此，GPU“线程”等同于 CPU“线程”，因为这两种线程都代表可由一个或多个核心执行的机器语言指令流。然而，GPU 执行线程的方式与 CPU 执行线程的方式略有不同。因此，“线程”一词在用于 GPU 计算内核时的含义与在 CPU 上运行程序时的含义略有不同。

为了理解术语上的差异，让我们简单了解一下 GPU 的架构。我们在第 4.11.1 节中提到，GPU 由多个计算单元组成，每个计算单元包含一定数量的 SIMD 单元。我们可以将计算单元想象成一个精简的 CPU 核心：它包含一个指令提取/解码单元，可能还有一些内存缓存，一个常规的“标量” ALU，通常还有四个左右的 SIMD 单元，这些单元的功能与支持 SIMD 的 CPU 中的矢量处理单元 (VPU) 大致相同。该架构如图 4.39 所示。

CU 中的 SIMD 单元在不同的 GPU 上具有不同的通道宽度，但为了便于讨论，我们假设我们正在使用 AMD

Figure 4.39. A typical GPU is comprised of multiple compute units (CU), each of which acts like a stripped down CPU core. A CU contains an instruction fetch/decode unit, an LI cache, possibly a block of local data storage RAM (LDS), a scalar ALU, and a number of SIMD units for executing vectorized code.

of *GPU threads*. But what exactly does the term “thread” mean in the context of a GPU?

Every GPU kernel is compiled into an *instruction stream* consisting of a sequence of GPU machine language instructions, in much the same way that a C/C++ function is compiled down into a stream of CPU instructions. So a GPU “thread” is equivalent to a CPU “thread,” in the sense that both kinds of threads represent a stream of machine language instructions that can be executed by one or more cores. However, a GPU executes its threads in a manner that is somewhat different from the way in which a CPU executes its threads. As a result, the term “thread” has a subtly different meaning when applied to GPU compute kernels than it has when applied to running programs on a CPU.

In order to understand this difference in terminology, let’s take a brief glimpse at the architecture of a GPU. We said in Section 4.11.1 that a GPU is comprised of multiple *compute units*, each of which contains some number of *SIMD units*. We can think of a compute unit like a stripped down CPU core: It contains an instruction fetch/decode unit, possibly some memory caches, a regular “scalar” ALU, and usually somewhere in the neighborhood of four SIMD units, which serve much the same function as the vector processing unit (VPU) in a SIMD-enabled CPU. This architecture is illustrated in Figure 4.39.

The SIMD units in a CU have different lane widths on different GPUs, but for the sake of this discussion let’s assume we’re working on an AMD

RadeonTM Graphics Core Next (GCN) 架构，其中 SIMD 为 16 条通道。计算单元 (CU) 不支持推测执行或乱序执行——它只是读取指令流并逐条执行，使用 SIMD 单元将每条指令并行应用于 14 到 16 个输入数据元素。

要在 CU 上执行计算内核，驱动程序首先将输入数据缓冲区细分为每个包含 64 个元素的数据块。对于每个包含 64 个元素的数据块，都会在一个 CU 上调用计算内核。这种调用称为波前 (wavefront)（在 NVIDIA 中也称为 warp）。执行波前时，CU 会逐一获取并解码内核的指令。每条指令都使用 SIMD 单元以锁步方式应用于 16 个数据元素。在内部，SIMD 单元由四级流水线组成，因此需要四个时钟周期才能完成。因此，CU 不会让该流水线的各个阶段每四个时钟周期有三个空闲时间，而是将同一条指令再应用三次，分别应用于另外三个包含 16 个数据元素的块。这就是为什么波前包含 64 个数据元素的原因，即使 CU 中的 SIMD 只有 16 个通道宽。

由于 CU 执行指令流的方式有些特殊，“GPU 线程”这个术语实际上指的是单个 SIMD 通道。因此，您可以将 GPU 线程视为我们开始时使用的原始非矢量化循环的一次迭代，然后再将其主体转换为计算内核。或者，您可以将 GPU 线程视为内核函数的一次调用，对单个输入数据进行操作并生成单个输出数据。GPU 实际上并行运行多个 GPU 线程（即，它实际上每个波前运行一次内核，但一次处理 64 个数据元素）只是一个实现细节。通过使程序员不必考虑如何在特定 GPU 上矢量化计算的细节，可以以可移植的方式编写计算内核（以及图形着色器）。

4.11.4.1 从 SIMD 到 SIMT

引入“单指令多线程 (SIMT)”这一术语是为了强调 GPU 不仅使用 SIMD 并行性，还使用一种抢占式多线程技术在波前之间进行时间切片。让我们简单了解一下这样做的原因。

SIMD 单元通过将着色器程序中的每条指令同时应用于 64 个数据元素来运行波前，本质上是同步进行的。（为了便于讨论，我们可以忽略波前以 16 个元素为一个子组进行处理的事实。）然而，任何一条指令

14 GPU 上的计算单元确实包含标量 ALU，因此可以以“单通道”方式执行一些指令，一次对单个输入数据进行操作。

Radeon™ Graphics Core Next (GCN) architecture, in which the SIMDs are 16 lanes wide. The CU is not capable of speculative or out-of-order execution—it simply reads a stream of instructions and executes them one by one, using the SIMD units to apply each instruction[14] to 16 elements of input data in parallel.

To execute a compute kernel on a CU, the driver first subdivides the input data buffers into blocks consisting of 64 elements each. For each of these 64-element blocks of data, the compute kernel is invoked on one CU. Such an invocation is called a *wavefront* (also known as a *warp* in NVIDIA speak). When executing a wavefront, the CU fetches and decodes the instructions of the kernel one by one. Each instruction is applied to 16 data elements in *lock step* using a SIMD unit. Internally, the SIMD unit consists of a four-*stage* pipeline, so it takes four clock cycles to complete. So rather than allow the stages of this pipeline to sit idle for three clock cycles out of every four, the CU applies the same instruction three more times, to three more blocks of 16 data elements. This is why a wavefront consists of 64 data elements, even though the SIMDs in the CU are only 16 lanes wide.

Because of this somewhat peculiar way in which a CU executes an instruction stream, the term “*GPU thread*” actually refers to a single SIMD *lane*. You can think of a GPU thread, therefore, as a single iteration of the original non-vectorized loop that we started with, before we converted its body into a compute kernel. Alternatively, you can think of a GPU thread as a single invocation of the kernel function, operating on a single input datum and producing a single output datum. The fact that the GPU actually runs multiple GPU threads in parallel (i.e., the fact that it really runs the kernel once per wavefront, but processes 64 data elements at a time) is just an implementation detail. By insulating the programmer from having to think about the details of how the computation is vectorized on any particular GPU, compute kernels (and graphics shaders as well) can be written in a portable manner.

4.11.4.1 From SIMD to SIMT

The term *single instruction multiple thread* (SIMT) was introduced to underscore the fact that a GPU doesn't just use SIMD parallelism—it also uses a form of preemptive multithreading to time-slice between wavefronts. Let's take a brief look at why this is done.

A SIMD unit runs a wavefront by applying each instruction in the shader program to 64 data elements at a time, essentially in lock-step. (We can ignore the fact that the wavefront is processed in subgroups of 16 elements each for the purposes of the present discussion.) However, any one instruction in the

[14] The compute units on a GPU do contain a scalar ALU and therefore can perform some instructions in “single lane” fashion, operating on a single input datum at a time.

图 4.40. 每当一个波前停滞时（例如由于内存访问延迟），SIMD 单元上下文就会切换到另一个波前来填充延迟槽。

程序可能最终必须访问内存，这会在 SIMD 单元等待内存控制器响应时引入较大的停顿。

为了填补这些较大的延迟槽，SIMD 单元会在多个波前（取自单个着色器程序，也可能取自多个不相关的着色器程序）之间进行时间切片。每当一个波前停滞时，SIMD 单元上下文就会切换到另一个波前，从而使该单元保持忙碌状态（只要有可运行的波前可以切换）。图 4.40 展示了此策略。

您可能已经想到，GPU 中的 SIMD 单元需要以非常高的频率执行上下文切换。在 CPU 上进行上下文切换时，传出线程的寄存器状态会保存到内存中，以免丢失；然后，传入线程的寄存器状态会从内存读取到 CPU 的寄存器中，以便线程可以从中断处继续执行。然而，在 GPU 上，每次发生上下文切换时保存每个 Wavefront 的 SIMD 寄存器状态会花费太多时间。

为了消除上下文切换期间保存寄存器的开销，每个 SIMD 单元都包含一个非常大的寄存器文件。该寄存器文件中的物理寄存器数量比任何一个 Wavefront 可用的逻辑寄存器数量大很多倍（通常大约是其十倍）。

Figure 4.40. Whenever one wavefront stalls, for example due to memory access latency, the SIMD unit context switches to another wavefront to fill the delay slot.

program might end up having to access memory, and that introduces a large stall while the SIMD unit waits for the memory controller to respond.

To fill these large delay slots, a SIMD unit time-slices between multiple wavefronts (taken from a single shader program, or potentially from many unrelated shader programs). Whenever one wavefront stalls, the SIMD unit context switches to another wavefront, thereby keeping the unit busy (as long as there are runnable wavefronts to switch to). This strategy is illustrated in Figure 4.40.

As you might well imagine, the SIMD units in a GPU need to perform context switches at a very high frequency. During a context switch on a CPU, the state of the outgoing thread's registers are saved to memory so they won't be lost, and then the state of the incoming thread's registers are read from memory into the CPU's registers so that it can continue executing where it left off. However, on a GPU it would take too much time to save the state of each wavefront's SIMD registers every time a context switch occurred.

To eliminate the cost of saving registers during context switches, each SIMD unit contains a very large register file. The number of *physical* registers in this register file is many times larger than the number of *logical* registers available to any one wavefront (typically on the order of ten times larger).

这意味着最多十个 Wavefront 的逻辑寄存器的内容可以始终保存在这些物理寄存器中。这反过来又意味着可以在 Wavefront 之间执行上下文切换，而无需保存或恢复任何寄存器。

4.11.5 进一步阅读

显然，GPGPU 编程是一个庞大的话题，本书也一如既往地只是略微涉及了一些皮毛。想要了解更多关于 GPGPU 和图形着色器编程的信息，请查看以下在线教程和资源：

- CUDA 编程简介：https://developer.nvidia.com/how-to-cuda-c-cpp
- OpenCL 学习资源：https://developer.nvidia.com/opencl
- HLSL 编程指南和参考手册：https://msdn.micro soft.com/en-us/library/bb5 09561(v=VS.85).aspx
- OpenGL 着色语言简介：https://www.khronos。org/opengl/wiki/OpenGL_Shading_Language
- AMD RadeonTM GCN 架构白皮书：https://www.amd. com/Documents/GCN_Architecture_whitepaper.pdf

This means that the contents of the logical registers for up to ten wavefronts can be maintained at all times in these physical registers. And that in turn implies that context switches can be performed between wavefronts without saving or restoring any registers whatsoever.

4.11.5 Further Reading

Obviously GPGPU programming is a huge topic, and as always we've only just scratched the surface in this book. For more information on GPGPU and graphics shader programming, check out the following online tutorials and resources:

- Introduction to CUDA programming: https://developer.nvidia.com/how-to-cuda-c-cpp
- OpenCL learning resources: https://developer.nvidia.com/opencl
- HLSL programming guide and reference manual: https://msdn.micro soft.com/en-us/library/bb509561(v=VS.85).aspx
- Introduction to the OpenGL shading language: https://www.khronos. org/opengl/wiki/OpenGL_Shading_Language
- AMD Radeon™ GCN architecture whitepaper: https://www.amd. com/Documents/GCN_Architecture_whitepaper.pdf

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

5
游戏中的 3D 数学

游戏是在某种计算机上实时模拟的虚拟世界的数学模型。因此，数学渗透到游戏行业的一切工作中。游戏程序员几乎运用所有数学分支，从三角学到代数、统计学到微积分。然而，作为一名游戏程序员，迄今为止最常用的数学是三维向量和矩阵数学（即三维线性代数）。

即使是数学这个分支，也非常广泛和深奥，所以我们不可能指望在一章之内就将其讲透彻。相反，我将尝试概述一个典型的游戏程序员所需的数学工具。在此过程中，我会提供一些技巧和窍门，帮助你理清所有那些相当令人困惑的概念和规则。要想深入了解游戏的3D数学，我强烈推荐Eric Lengyel的专著[32]。Christer Ericson的专著[14]中关于实时碰撞检测的第三章也是一本很棒的资源。

5.1 在二维空间中求解三维问题

我们将在下一章学习的许多数学运算在二维和三维空间中同样有效。这是一个好消息，因为

5
3D Math for Games

A game is a mathematical model of a virtual world simulated in real time on a computer of some kind. Therefore, mathematics pervades everything we do in the game industry. Game programmers make use of virtually all branches of mathematics, from trigonometry to algebra to statistics to calculus. However, by far the most prevalent kind of mathematics you'll be doing as a game programmer is 3D vector and matrix math (i.e., 3D *linear algebra*).

Even this one branch of mathematics is very broad and very deep, so we cannot hope to cover it in any great depth in a single chapter. Instead, I will attempt to provide an overview of the mathematical tools needed by a typical game programmer. Along the way, I'll offer some tips and tricks, which should help you keep all of the rather confusing concepts and rules straight in your head. For an excellent in-depth coverage of 3D math for games, I highly recommend Eric Lengyel's book [32] on the topic. Chapter 3 of Christer Ericson's book [14] on real-time collision detection is also an excellent resource.

5.1 Solving 3D Problems in 2D

Many of the mathematical operations we're going to learn about in the following chapter work equally well in 2D and 3D. This is very good news, because

这意味着您有时可以通过在二维中思考和绘图来解决三维矢量问题（这相当容易！）遗憾的是，二维和三维之间的这种等价性并不总是成立。某些运算（如叉积）仅在三维中定义，而某些问题只有考虑三维才有意义。尽管如此，从考虑手头问题的简化二维版本开始几乎永远不会有坏处。一旦您理解了二维的解决方案，您就可以思考如何将问题扩展到三维。在某些情况下，您会高兴地发现您的二维结果在三维中也有效。在其他情况下，您将能够找到一个问题确实是二维的坐标系。在本书中，只要二维和三维之间的区别不重要，我们就会使用二维图。

5.2 点和向量

大多数现代 3D 游戏都是由虚拟世界中的三维物体组成的。游戏引擎需要追踪所有这些物体的位置、方向和比例，在游戏世界中为它们设置动画，并将它们转换到屏幕空间，以便能够在屏幕上渲染。游戏中的 3D 物体几乎总是由三角形组成，三角形的顶点用点表示。因此，在我们学习如何在游戏引擎中表示整个物体之前，让我们先来了解一下点以及它密切相关的“表亲”——矢量。

图 5.1. 以笛卡尔坐标表示的点。

5.2.1 点和笛卡尔坐标

从技术角度来说，点是 n 维空间中的一个位置。（在游戏中，n 通常等于 2 或 3。）笛卡尔坐标系是迄今为止游戏程序员最常用的坐标系。它使用两个或三个相互垂直的轴来指定二维或三维空间中的位置。因此，点 P 可以用一对或三个实数（P x , P y）或（P x , P y , P z）表示（参见图 5.1）。

图 5.2. 圆柱坐标中表示的点。

当然，笛卡尔坐标系并不是我们唯一的选择。其他一些常见的坐标系包括：

- 圆柱坐标系。该系统采用垂直“高度”轴 h 、从垂直方向向外延伸的径向轴以及偏航角 θ 。在圆柱坐标系中，点 P 用三个数字（Ph , Pr , P θ）表示。如图 5.2 所示。

it means you can *sometimes* solve a 3D vector problem by thinking and drawing pictures in 2D (which is considerably easier to do!) Sadly, this equivalence between 2D and 3D does not hold all the time. Some operations, like the cross product, are only defined in 3D, and some problems only make sense when all three dimensions are considered. Nonetheless, it almost never hurts to start by thinking about a simplified two-dimensional version of the problem at hand. Once you understand the solution in 2D, you can think about how the problem extends into three dimensions. In some cases, you'll happily discover that your 2D result works in 3D as well. In others, you'll be able to find a coordinate system in which the problem really *is* two-dimensional. In this book, we'll employ two-dimensional diagrams wherever the distinction between 2D and 3D is not relevant.

5.2 Points and Vectors

The majority of modern 3D games are made up of three-dimensional objects in a virtual world. A game engine needs to keep track of the positions, orientations and scales of all these objects, animate them in the game world, and transform them into screen space so they can be rendered on screen. In games, 3D objects are almost always made up of triangles, the vertices of which are represented by points. So, before we learn how to represent whole objects in a game engine, let's first take a look at the point and its closely related cousin, the vector.

Figure 5.1. A point represented in Cartesian coordinates.

5.2.1 Points and Cartesian Coordinates

Technically speaking, a *point* is a location in n-dimensional space. (In games, n is usually equal to 2 or 3.) The Cartesian coordinate system is by far the most common coordinate system employed by game programmers. It uses two or three mutually perpendicular axes to specify a position in 2D or 3D space. So, a point **P** is represented by a pair or triple of real numbers, (P_x, P_y) or (P_x, P_y, P_z) (see Figure 5.1).

Of course, the Cartesian coordinate system is not our only choice. Some other common systems include:

Figure 5.2. A point represented in cylindrical coordinates.

- *Cylindrical coordinates*. This system employs a vertical "height" axis h, a radial axis r emanating out from the vertical, and a yaw angle *theta* (θ). In cylindrical coordinates, a point **P** is represented by the triple of numbers (P_h, P_r, P_θ). This is illustrated in Figure 5.2.

• 球面坐标系。该系统采用俯仰角φ、偏航角θ和径向测量值r。因此，点可以用三个数字（P r ，P φ ，P θ ）表示。如图5.3所示。

图 5.3. 球坐标中表示的点。

笛卡尔坐标系是迄今为止游戏编程中使用最广泛的坐标系。但是，请务必记住选择最能体现当前问题的坐标系。例如，在 Midway Home Entertainment 开发的游戏《Crank the Weasel》中，主角 Crank 在装饰艺术风格的城市中奔跑捡拾战利品。我希望让战利品围绕 Crank 的身体呈螺旋状旋转，越来越靠近他，直至消失。我用圆柱坐标系表示战利品相对于 Crank 角色当前位置的位置。为了实现螺旋动画，我只需赋予战利品一个沿 θ 的恒定角速度、一个沿其径向轴 r 向内运动的微小恒定线速度以及一个沿 h 轴向上运动的非常微小的恒定线速度，这样战利品就会逐渐上升到 Crank 裤子口袋的高度。这个极其简单的动画看起来非常棒，而且使用圆柱坐标系建模比使用笛卡尔坐标系容易得多。

5.2.2 左手坐标系与右手坐标系

在三维笛卡尔坐标系中，我们有两种方式来排列三个相互垂直的坐标轴：右手坐标系 (RH) 和左手坐标系 (LH)。在右手坐标系中，当你用右手的手指绕 z 轴弯曲，拇指指向 z 轴正方向时，你的手指会从 x 轴指向 y 轴。在左手坐标系中，用左手弯曲手指时，情况也是如此。

左手坐标系和右手坐标系之间的唯一区别是

图 5.4 左手和右手笛卡尔坐标系。

- *Spherical coordinates*. This system employs a pitch angle phi (ϕ), a yaw angle theta (θ) and a radial measurement r. Points are therefore represented by the triple of numbers (P_r, P_ϕ, P_θ). This is illustrated in Figure 5.3.

Figure 5.3. A point represented in spherical coordinates.

Cartesian coordinates are by far the most widely used coordinate system in game programming. However, always remember to select the coordinate system that best maps to the problem at hand. For example, in the game *Crank the Weasel* by Midway Home Entertainment, the main character Crank runs around an art-deco city picking up loot. I wanted to make the items of loot swirl around Crank's body in a spiral, getting closer and closer to him until they disappeared. I represented the position of the loot in cylindrical coordinates relative to the Crank character's current position. To implement the spiral animation, I simply gave the loot a constant angular speed in θ, a small constant linear speed inward along its radial axis r and a very slight constant linear speed upward along the h-axis so the loot would gradually rise up to the level of Crank's pants pockets. This extremely simple animation looked great, and it was much easier to model using cylindrical coordinates than it would have been using a Cartesian system.

5.2.2 Left-Handed versus Right-Handed Coordinate Systems

In three-dimensional Cartesian coordinates, we have two choices when arranging our three mutually perpendicular axes: right-handed (RH) and left-handed (LH). In a right-handed coordinate system, when you curl the fingers of your right hand around the z-axis with the thumb pointing toward positive z coordinates, your fingers point from the x-axis toward the y-axis. In a left-handed coordinate system the same thing is true using your left hand.

The only difference between a left-handed coordinate system and a right-

Figure 5.4. Left- and right-handed Cartesian coordinate systems.

手性坐标系是指三个轴中任意一个指向的方向。例如，如果 y 轴指向上方，x 轴指向右侧，则在右手系中，z 轴朝向我们（向页面外），而在左手系中，z 轴远离我们（向页面内）。左手和右手笛卡尔坐标系如图 5.4 所示。

左手坐标系和右手坐标系之间的转换非常简单。我们只需翻转任意一个轴的方向，其他两个轴保持不变即可。需要记住的是，在左手坐标系和右手坐标系之间，数学规则并不会改变。改变的只是我们对数字的解读——我们对数字如何映射到三维空间的心理图像。左手和右手的惯例仅适用于可视化，而不适用于底层数学。（实际上，在物理模拟中处理叉积时，惯用手性确实很重要，因为叉积实际上不是向量——它是一个称为伪向量的特殊数学对象。我们将在 5.2.4.9 节中更深入地讨论伪向量。）

数值表示和视觉表示之间的映射完全取决于我们数学家和程序员。我们可以选择让 y 轴指向上方，z 轴指向前方，x 轴指向左侧（RH）或右侧（LH）。或者，我们可以选择让 z 轴指向上方。或者，x 轴指向上方或下方。重要的是，我们决定一个映射，然后始终坚持下去。

话虽如此，某些约定在某些应用中确实比其他约定更有效。例如，3D 图形程序员通常使用左手坐标系，y 轴向上，x 轴向右，正 z 轴指向远离观察者的方向（即虚拟相机指向的方向）。当使用这种特定的坐标系将 3D 图形渲染到 2D 屏幕上时，z 坐标的增加对应于场景深度的增加（即与虚拟相机的距离增加）。正如我们将在后续章节中看到的，这正是使用 αz 缓冲方案进行深度遮挡时所需要的。

5.2.3 向量

向量是 n 维空间中既有大小又有方向的量。向量可以理解为一条有向线段，从称为尾的点延伸到称为头的点。与此相对，标量（即普通的实值数）表□□示大小，但没有方向。通常，标量用斜体表示（例如 v ），而向量用粗体表示（例如 v ）。

三维向量可以用三个标量（x , y , z）来表示，就像一个点

handed coordinate system is the direction in which one of the three axes is pointing. For example, if the y-axis points upward and x points to the right, then z comes toward us (out of the page) in a right-handed system, and away from us (into the page) in a left-handed system. Left- and right-handed Cartesian coordinate systems are depicted in Figure 5.4.

It is easy to convert from left-handed to right-handed coordinates and vice versa. We simply flip the direction of any one axis, leaving the other two axes alone. It's important to remember that the rules of mathematics do not change between left-handed and right-handed coordinate systems. Only our *interpretation* of the numbers—our mental image of how the numbers map into 3D space—changes. Left-handed and right-handed conventions apply to visualization only, not to the underlying mathematics. (Actually, handedness does matter when dealing with cross products in physical simulations, because a cross product is not actually a vector—it's a special mathematical object known as a *pseudovector*. We'll discuss pseudovectors in a little more depth in Section 5.2.4.9.)

The mapping between the numerical representation and the visual representation is entirely up to us as mathematicians and programmers. We could choose to have the y-axis pointing up, with z forward and x to the left (RH) or right (LH). Or we could choose to have the z-axis point up. Or the x-axis could point up instead—or down. All that matters is that we decide upon a mapping, and then stick with it consistently.

That being said, some conventions do tend to work better than others for certain applications. For example, 3D graphics programmers typically work with a left-handed coordinate system, with the y-axis pointing up, x to the right and positive z pointing away from the viewer (i.e., in the direction the virtual camera is pointing). When 3D graphics are rendered onto a 2D screen using this particular coordinate system, increasing z-coordinates correspond to increasing *depth* into the scene (i.e., increasing distance away from the virtual camera). As we will see in subsequent chapters, this is exactly what is required when using a z-buffering scheme for depth occlusion.

5.2.3 Vectors

A *vector* is a quantity that has both a *magnitude* and a *direction* in n-dimensional space. A vector can be visualized as a *directed line segment* extending from a point called the *tail* to a point called the *head*. Contrast this to a *scalar* (i.e., an ordinary real-valued number), which represents a magnitude but has no direction. Usually scalars are written in italics (e.g., v) while vectors are written in boldface (e.g., $\mathbf{v}$).

A 3D vector can be represented by a triple of scalars (x, y, z), just as a point

可以。点和向量之间的区别实际上非常微妙。从技术上讲，向量只是相对于某个已知点的偏移量。向量可以在三维空间中移动到任意位置——只要其大小和方向不变，它就是同一个向量。

只要我们将向量的尾部固定在坐标系的原点，向量就可以用来表示点。这样的向量有时被称为位置向量或半径向量。为了便于理解，我们可以将任意三个标量解释为点或向量，前提是位置向量的尾部必须固定在所选坐标系的原点。这意味着点和向量在数学上处理方式略有不同。有人可能会说点是绝对的，而向量是相对的。

绝大多数游戏程序员使用术语“向量”来指代点（位置向量）和严格线性代数意义上的向量（纯方向向量）。大多数 3D 数学库也以这种方式使用术语“向量”。在本书中，当区别很重要时，我们将使用术语“方向向量”或仅使用“方向”。请务必始终牢记点和方向之间的区别（即使您的数学库没有意识到）。正如我们将在 5.3.6.1 节中看到的那样，在将方向转换为齐次坐标以便使用 4×4 矩阵进行操作时，方向需要与点区别对待，因此混淆这两种类型的向量可能会导致代码中出现错误。

5.2.3.1 笛卡尔基向量

通常，定义三个正交的单位向量（即相互垂直且每个向量的长度都等于一的向量）来对应三个笛卡尔主轴是很有用的。沿 x 轴的单位向量通常称为 i ，沿 y 轴的单位向量称为 j ，沿 z 轴的单位向量称为 k 。

向量 i 、 j 和 k 有时被称为笛卡尔基向量。
任何点或向量都可以表示为标量（实数）乘以这些单位基向量之和。例如：

$$(5, 3, -2) = 5\mathbf{i} + 3\mathbf{j} - 2\mathbf{k}.$$

5.2.4 向量运算

大多数对标量进行的数学运算也适用于向量。此外，还有一些新的运算仅适用于向量。

can be. The distinction between points and vectors is actually quite subtle. Technically, a vector is just an offset *relative to* some known point. A vector can be moved anywhere in 3D space—as long as its magnitude and direction don't change, it is the same vector.

A vector can be used to represent a point, provided that we fix the tail of the vector to the origin of our coordinate system. Such a vector is sometimes called a *position vector* or *radius vector*. For our purposes, we can interpret any triple of scalars as either a point or a vector, provided that we remember that a *position vector* is constrained such that its tail remains at the origin of the chosen coordinate system. This implies that points and vectors are treated in subtly different ways mathematically. One might say that points are *absolute*, while vectors are *relative*.

The vast majority of game programmers use the term "vector" to refer both to points (position vectors) and to vectors in the strict linear algebra sense (purely directional vectors). Most 3D math libraries also use the term "vector" in this way. In this book, we'll use the term "direction vector" or just "direction" when the distinction is important. Be careful to always keep the difference between points and directions clear in your mind (even if your math library doesn't). As we'll see in Section 5.3.6.1, directions need to be treated differently from points when converting them into homogeneous coordinates for manipulation with 4×4 matrices, so getting the two types of vector mixed up can and will lead to bugs in your code.

5.2.3.1 Cartesian Basis Vectors

It is often useful to define three *orthogonal unit vectors* (i.e., vectors that are mutually perpendicular and each with a length equal to one), corresponding to the three principal Cartesian axes. The unit vector along the x-axis is typically called $\mathbf{i}$, the y-axis unit vector is called $\mathbf{j}$, and the z-axis unit vector is called $\mathbf{k}$. The vectors $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are sometimes called Cartesian *basis vectors*.

Any point or vector can be expressed as a sum of scalars (real numbers) multiplied by these unit basis vectors. For example,

$$(5, 3, -2) = 5\mathbf{i} + 3\mathbf{j} - 2\mathbf{k}.$$

5.2.4 Vector Operations

Most of the mathematical operations that you can perform on scalars can be applied to vectors as well. There are also some new operations that apply only to vectors.

图 5.5. 向量与标量 2 的乘积。

5.2.4.1 与标量相乘

向量 a 与标量 s 的乘法是通过将 a 的各个分量乘以 s 来实现的：

$$s\mathbf{a} = (sa_x, sa_y, sa_z).$$

与标量相乘会缩放向量的大小，同时保持其方向不变，如图 5.5 所示。与 -1 相乘会翻转向量的方向（头部变成尾部，反之亦然）。

缩放因子在每个轴上可以不同。我们称之为非均匀缩放，它可以表示为缩放向量 s 与待缩放向量（我们用 □ 运算符表示）的分量乘积。从技术上讲，两个向量之间的这种特殊乘积被称为 Hadamard 积。它在游戏行业中很少使用——事实上，非均匀缩放是它在游戏中仅有的常见用途之一：

$$\mathbf{s} \otimes \mathbf{a} = (s_x a_x, s_y a_y, s_z a_z). \tag{5.1}$$

正如我们将在 5.3.7.3 节中看到的那样，缩放向量 s 实际上只是 3×3 对角缩放矩阵 S 的一种简洁表示。因此，公式 (5.1) 的另一种写法如下：

$$\mathbf{aS} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} s_x a_x & s_y a_y & s_z a_z \end{bmatrix}.$$

我们将在第 5.3 节更深入地探讨矩阵。

5.2.4.2 加法和减法

两个向量 a 和 b 的加法定义为：一个向量的分量等于 a 和 b 分量之和。这可以通过以下方式可视化：

Figure 5.5. Multiplication of a vector by the scalar 2.

5.2.4.1 Multiplication by a Scalar

Multiplication of a vector **a** by a scalar s is accomplished by multiplying the individual components of **a** by s:

$$s\mathbf{a} = (sa_x, sa_y, sa_z).$$

Multiplication by a scalar has the effect of scaling the magnitude of the vector, while leaving its direction unchanged, as shown in Figure 5.5. Multiplication by -1 flips the direction of the vector (the head becomes the tail and vice versa).

The scale factor can be different along each axis. We call this *nonuniform scale*, and it can be represented as the *component-wise product* of a scaling vector **s** and the vector in question, which we'll denote with the $\otimes$ operator. Technically speaking, this special kind of product between two vectors is known as the *Hadamard product*. It is rarely used in the game industry—in fact, nonuniform scaling is one of its *only* commonplace uses in games:

$$\mathbf{s} \otimes \mathbf{a} = (s_x a_x, s_y a_y, s_z a_z). \tag{5.1}$$

As we'll see in Section 5.3.7.3, a scaling vector **s** is really just a compact way to represent a 3×3 diagonal scaling matrix **S**. So another way to write Equation (5.1) is as follows:

$$\mathbf{aS} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} s_x a_x & s_y a_y & s_z a_z \end{bmatrix}.$$

We'll explore matrices in more depth in Section 5.3.

5.2.4.2 Addition and Subtraction

The addition of two vectors **a** and **b** is defined as the vector whose components are the sums of the *components* of **a** and **b**. This can be visualized by placing

图 5.6. 向量加法和减法。

图 5.7. 矢量的大小（为便于说明，以 2D 形式显示）。

将向量 a 的头部放到向量 b 的尾部——和就是从向量 a 的尾部到向量 b 的头部（参见图 5.6）：

$$\mathbf{a} + \mathbf{b} = \left[(a_x + b_x), (a_y + b_y), (a_z + b_z)\right].$$

向量减法 a − b 只不过是 a 和 − b 的加法（即将 b 乘以 − 1 的结果，也就是翻转的结果）。这对应于一个向量，其分量是 a 的分量与 b 的分量之差：

$$\mathbf{a} - \mathbf{b} = \left[(a_x - b_x), (a_y - b_y), (a_z - b_z)\right].$$

向量加法和减法如图 5.6 所示。

添加和减去点和方向

你可以自由地加减方向向量。然而，从技术角度来说，点之间不能互相加法——你只能将一个方向向量加到一个点上，其结果为另一个点。同样，你可以取两个点之间的差，从而得到一个方向向量。这些操作总结如下：

- 方向 + 方向 = 方向
- 方向 – 方向 = 方向
- 点 + 方向 = 点
- 点 – 点 = 方向
- 点 + 点 = 无意义 5.2.4.3 幅度

向量的幅值是一个标量，表示向量在二维或三维空间中的长度。它用竖线表示

Figure 5.6. Vector addition and subtraction.

Figure 5.7. Magnitude of a vector (shown in 2D for ease of illustration).

the head of vector **a** onto the tail of vector **b**—the sum is then the vector from the tail of **a** to the head of **b** (see also Figure 5.6):

$$\mathbf{a} + \mathbf{b} = \left[(a_x + b_x), (a_y + b_y), (a_z + b_z)\right].$$

Vector subtraction $\mathbf{a} - \mathbf{b}$ is nothing more than addition of **a** and $-\mathbf{b}$ (i.e., the result of scaling **b** by -1, which flips it around). This corresponds to the vector whose components are the difference between the components of **a** and the components of **b**:

$$\mathbf{a} - \mathbf{b} = \left[(a_x - b_x), (a_y - b_y), (a_z - b_z)\right].$$

Vector addition and subtraction are depicted in Figure 5.6.

Adding and Subtracting Points and Directions

You can add and subtract direction vectors freely. However, technically speaking, points cannot be added to one another—you can only add a direction vector to a point, the result of which is another point. Likewise, you can take the difference between two points, resulting in a direction vector. These operations are summarized below:

- direction + direction = direction
- direction – direction = direction
- point + direction = point
- point – point = direction
- point + point = *nonsense*

5.2.4.3 Magnitude

The magnitude of a vector is a scalar representing the length of the vector as it would be measured in 2D or 3D space. It is denoted by placing vertical bars

围绕向量的粗体符号。我们可以使用勾股定理来计算向量的大小，如图 5.7 所示：

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

5.2.4.4 矢量运算的实际应用

图 5.8. 给定角色在当前帧中的位置和速度，可以使用简单的矢量加法来找到角色在下一帧中的位置。

信不信由你，我们已经可以仅凭迄今为止所学的向量运算来解决各种现实世界的游戏问题。当尝试解决问题时，我们可以使用加法、减法、缩放和幅度等运算从已知内容中生成新数据。例如，如果我们有 AI 角色 P 1 的当前位置向量，以及表示她当前速度的向量 v，我们可以通过将速度向量按帧时间间隔 Δ t 缩放，然后将其添加到当前位置，找到她在下一帧的位置 P 2。如图 5.8 所示，得到的向量方程为 P 2 = P 1 + v Δ t。（这称为显式欧拉积分——它实际上仅在速度恒定时有效，但你明白我的意思。）再举一个例子，假设我们有两个球体，我们想知道它们是否相交。已知两个球体的圆心 C 1 和 C 2，只需将这两个点相减，即可求出它们之间的方向向量 d = C 2 − C 1 。该向量 d = | d | 的大小决定了两个球体圆心之间的距离。如果该距离小于两个球体半径之和，则它们相交；否则，则不相交。如图 5.9 所示。

在大多数计算机上计算平方根的开销很大，因此游戏程序员应该始终使用平方幅度，只要这样做是有效的

图 5.9 球体与球体相交测试仅涉及向量减法、向量幅度和浮点比较运算。

around the vector's boldface symbol. We can use the Pythagorean theorem to calculate a vector's magnitude, as shown in Figure 5.7:

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

5.2.4.4 Vector Operations in Action

Figure 5.8. Simple vector addition can be used to find a character's position in the next frame, given her position and velocity in the current frame.

Believe it or not, we can already solve all sorts of real-world game problems given just the vector operations we've learned thus far. When trying to solve a problem, we can use operations like addition, subtraction, scaling and magnitude to generate new data out of the things we already know. For example, if we have the current position vector of an AI character $\mathbf{P}_1$, and a vector $\mathbf{v}$ representing her current velocity, we can find her position on the next frame $\mathbf{P}_2$ by scaling the velocity vector by the frame time interval Δt, and then adding it to the current position. As shown in Figure 5.8, the resulting vector equation is $\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{v}\,\Delta t$. (This is known as *explicit Euler integration*—it's actually only valid when the velocity is constant, but you get the idea.)

As another example, let's say we have two spheres, and we want to know whether they intersect. Given that we know the center points of the two spheres, $\mathbf{C}_1$ and $\mathbf{C}_2$, we can find a direction vector between them by simply subtracting the points, $\mathbf{d} = \mathbf{C}_2 - \mathbf{C}_1$. The magnitude of this vector $d = |\mathbf{d}|$ determines how far apart the spheres' centers are. If this distance is less than the sum of the spheres' radii, they are intersecting; otherwise they're not. This is shown in Figure 5.9.

Square roots are expensive to calculate on most computers, so game programmers should always use the *squared magnitude* whenever it is valid to do

Figure 5.9. A sphere-sphere intersection test involves only vector subtraction, vector magnitude and floating-point comparison operations.

so:

$$|\mathbf{a}|^2 = (a_x^2 + a_y^2 + a_z^2).$$

在比较两个向量的相对长度（“向量 a 是否比向量 b 长？”）或将向量的幅值与其他标量（平方）进行比较时，使用幅值的平方是有效的。因此，在我们的球体相交测试中，我们应该计算 d 2 = | d | 2 ，并将其与半径的平方和 (r 1 + r 2) 2 进行比较，以获得最快的速度。编写高性能软件时，如无必要，切勿开平方！

5.2.4.5 归一化和单位向量

单位向量是量级（长度）为 1 的向量。单位向量在 3D 数学和游戏编程中非常有用，具体原因我们将在下文中讨论。

给定一个长度为 v = | v | 的任意向量 v ，我们可以将其转换为一个单位向量 u ，该向量指向与 v 相同的方向，但长度为单位。为此，我们只需将 v 乘以其幅值的倒数即可。我们称之为归一化：

$$\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{v}\mathbf{v}.$$

5.2.4.6 法向量

如果一个向量垂直于某个表面，则称该向量为该表面的法线。法线向量在游戏和计算机图形学中非常有用。例如，一个平面可以由一个点和一个法线向量定义。在 3D 图形中，光照计算大量使用法线向量来定义表面相对于入射光线方向的方向。

法向量通常具有单位长度，但并非必须如此。请注意不要将“归一化”与“法向量”混淆。归一化向量是任何单位长度的向量。法向量是任何垂直于表面的向量，无论其长度是否为单位。

5.2.4.7 点积和投影

向量可以相乘，但与标量不同的是，向量乘法有很多种。在游戏编程中，我们最常使用以下两种乘法：

- 点积（又称标量积或内积），以及
- 叉积（又称向量积或外积）。

so:

$$|\mathbf{a}|^2 = (a_x^2 + a_y^2 + a_z^2).$$

Using the squared magnitude is valid when comparing the *relative lengths* of two vectors (“is vector **a** longer than vector **b**?”), or when comparing a vector’s magnitude to some other (squared) scalar quantity. So in our sphere-sphere intersection test, we should calculate $d^2 = |\mathbf{d}|^2$ and compare this to the squared sum of the radii, $(r_1 + r_2)^2$ for maximum speed. When writing high-performance software, never take a square root when you don’t have to!

5.2.4.5 Normalization and Unit Vectors

A *unit vector* is a vector with a magnitude (length) of one. Unit vectors are very useful in 3D mathematics and game programming, for reasons we’ll see below.

Given an arbitrary vector **v** of length $v = |\mathbf{v}|$, we can convert it to a unit vector **u** that points in the same direction as **v**, but has unit length. To do this, we simply multiply **v** by the reciprocal of its magnitude. We call this *normalization:*

$$\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{v}\mathbf{v}.$$

5.2.4.6 Normal Vectors

A vector is said to be *normal* to a surface if it is *perpendicular* to that surface. Normal vectors are highly useful in games and computer graphics. For example, a *plane* can be defined by a point and a normal vector. And in 3D graphics, lighting calculations make heavy use of normal vectors to define the direction of surfaces relative to the direction of the light rays impinging upon them.

Normal vectors are usually of unit length, but they do not need to be. Be careful not to confuse the term “normalization” with the term “normal vector.” A normalized vector is any vector of unit length. A normal vector is any vector that is perpendicular to a surface, whether or not it is of unit length.

5.2.4.7 Dot Product and Projection

Vectors can be multiplied, but unlike scalars there are a number of different kinds of vector multiplication. In game programming, we most often work with the following two kinds of multiplication:

- the *dot product* (a.k.a. scalar product or inner product), and
- the *cross product* (a.k.a. vector product or outer product).

两个向量的点积产生一个标量；它的定义是将两个向量的各个分量的乘积相加：

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = d \quad \text{(a scalar)}.$$

点积也可以写成两个向量的大小与它们之间角度的余弦的乘积：

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta.$$

点积是可交换的（即两个向量的顺序可以反转），并且对于加法是可分配的：

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a};$$
$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}.$$

点积与标量乘法的组合如下：

$$s\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \cdot s\mathbf{b} = s(\mathbf{a} \cdot \mathbf{b}).$$

矢量投影

如果 u 是单位向量（|u|=1），则点积（a · u）表示向量 a 在由 u 方向定义的无限线上的投影长度，如图 5.10 所示。此投影概念在二维和三维空间中同样适用，并且对于解决各种三维问题非常有用。

图 5.10.使用点积的矢量投影。

量级作为点积

向量的平方可以通过以下点积来求得：

The dot product of two vectors yields a scalar; it is defined by adding the products of the individual components of the two vectors:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = d \quad \text{(a scalar)}.$$

The dot product can also be written as the product of the magnitudes of the two vectors and the cosine of the angle between them:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta.$$

The dot product is *commutative* (i.e., the order of the two vectors can be reversed) and *distributive* over addition:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a};$$
$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}.$$

And the dot product combines with scalar multiplication as follows:

$$s\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \cdot s\mathbf{b} = s(\mathbf{a} \cdot \mathbf{b}).$$

Vector Projection

If $\mathbf{u}$ is a unit vector ($|\mathbf{u}| = 1$), then the dot product ($\mathbf{a} \cdot \mathbf{u}$) represents the length of the *projection* of vector $\mathbf{a}$ onto the infinite line defined by the direction of $\mathbf{u}$, as shown in Figure 5.10. This projection concept works equally well in 2D or 3D and is highly useful for solving a wide variety of three-dimensional problems.

Figure 5.10. Vector projection using the dot product.

Magnitude as a Dot Product

The squared magnitude of a vector can be found by taking the dot product of

该向量与其自身相乘。然后，只需开平方，就可以轻松求出其幅值：

$$|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a};$$
$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}.$$

这是可行的，因为零度的余弦是 1，所以 $|a||a|\cos\theta = |a||a| = |\mathbf{a}|^2$.

点积测试

点积非常适合用来测试两个向量是共线还是垂直，或者它们指向的方向大致相同还是大致相反。对于任意两个向量 a 和 b ，游戏程序员经常使用以下测试，如图 5.11 所示：

- 共线。（a · b）= |a||b|=ab（即，它们之间的角度恰好为0度 - 当a和b为单位向量时，该点积等于+1）。
- 共线但相反。（a · b）= −ab（即，它们之间的角度为180度 - 当a和b为单位向量时，该点积等于−1）。
- 垂直。（a · b）= 0（即它们之间的角度为90度）。
- 同一方向。（a · b）>0（即它们之间的角度小于90度）。
- 相反的方向。（a · b）<0（即它们之间的角度大于90度）。

点积的其他一些应用

点积可用于游戏编程中的各种用途。例如，假设我们想知道敌人是在玩家角色的前面还是后面。我们可以通过简单的向量减法（v = E − P）找到从玩家位置 P 到敌人位置 E 的向量。假设我们有一个向量 f 指向玩家所面向的方向。（正如我们将在 5.3.10.3 节中看到的，向量 f 可以直接从玩家的模型到世界矩阵中提取。）点积 d = v · f 可用于测试敌人是在玩家前面还是后面——当敌人在前面时，它是正数；当敌人在后面时，它是负数。

点积也可以用来计算平面上方或下方点的高度（例如，在编写登月游戏时，这可能很有用）。我们可以用两个矢量来定义一个平面：一个位于平面任意位置的点 Q，以及一个垂直于平面的单位矢量 n（即法线）。

that vector with itself. Its magnitude is then easily found by taking the square root:

$$|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a};$$
$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}.$$

This works because the cosine of zero degrees is 1, so $|\mathbf{a}|\,|\mathbf{a}|\cos\theta = |\mathbf{a}|\,|\mathbf{a}| = |\mathbf{a}|^2$.

Dot Product Tests

Dot products are great for testing if two vectors are collinear or perpendicular, or whether they point in roughly the same or roughly opposite directions. For any two arbitrary vectors **a** and **b**, game programmers often use the following tests, as shown in Figure 5.11:

- *Collinear*. $(\mathbf{a} \cdot \mathbf{b}) = |\mathbf{a}|\,|\mathbf{b}| = ab$ (i.e., the angle between them is exactly 0 degrees—this dot product equals $+1$ when **a** and **b** are *unit vectors*).
- *Collinear but opposite*. $(\mathbf{a} \cdot \mathbf{b}) = -ab$ (i.e., the angle between them is 180 degrees—this dot product equals -1 when **a** and **b** are unit vectors).
- *Perpendicular*. $(\mathbf{a} \cdot \mathbf{b}) = 0$ (i.e., the angle between them is 90 degrees).
- *Same direction*. $(\mathbf{a} \cdot \mathbf{b}) > 0$ (i.e., the angle between them is less than 90 degrees).
- *Opposite directions*. $(\mathbf{a} \cdot \mathbf{b}) < 0$ (i.e., the angle between them is greater than 90 degrees).

Some Other Applications of the Dot Product

Dot products can be used for all sorts of things in game programming. For example, let's say we want to find out whether an enemy is in front of the player character or behind him. We can find a vector from the player's position **P** to the enemy's position **E** by simple vector subtraction ($\mathbf{v} = \mathbf{E} - \mathbf{P}$). Let's assume we have a vector **f** pointing in the direction that the player is *facing*. (As we'll see in Section 5.3.10.3, the vector **f** can be extracted directly from the player's model-to-world matrix.) The dot product $d = \mathbf{v} \cdot \mathbf{f}$ can be used to test whether the enemy is in front of or behind the player—it will be positive when the enemy is in front and negative when the enemy is behind.

The dot product can also be used to find the height of a point above or below a plane (which might be useful when writing a moon-landing game for example). We can define a plane with two vector quantities: a point **Q** lying anywhere on the plane, and a unit vector **n** that is perpendicular (i.e., normal)

图 5.11 一些常见的点积测试。

图 5.12 点积可用于查找平面上方或下方点的高度。

到平面。为了求点 P 在平面上方的高度 h，我们首先计算从平面上任意一点（Q 就很好了）到待求点 P 的一个向量。因此我们有 v = P − Q 。向量 v 与单位长度法向量 n 的点积就是 v 在 n 定义的直线上的投影。但这恰恰是我们要求的高度。因此，

$$h = \mathbf{v} \cdot \mathbf{n} = (\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}. \tag{5.2}$$

如图 5.12 所示。

5.2.4.8 叉积

两个向量的叉积（也称为外积或向量积）会得到另一个与被乘向量垂直的向量，如图 5.13 所示。叉积运算仅在

Figure 5.11. Some common dot product tests.

Figure 5.12. The dot product can be used to find the height of a point above or below a plane.

to the plane. To find the height h of a point $\mathbf{P}$ above the plane, we first calculate a vector from any point on the plane ($\mathbf{Q}$ will do nicely) to the point in question $\mathbf{P}$. So we have $\mathbf{v} = \mathbf{P} - \mathbf{Q}$. The dot product of vector $\mathbf{v}$ with the unit-length normal vector $\mathbf{n}$ is just the projection of $\mathbf{v}$ onto the line defined by $\mathbf{n}$. But that is exactly the height we're looking for. Therefore,

$$h = \mathbf{v} \cdot \mathbf{n} = (\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}. \tag{5.2}$$

This is illustrated in Figure 5.12.

5.2.4.8 Cross Product

The *cross product* (also known as the *outer product* or *vector product*) of two vectors yields another *vector* that is *perpendicular* to the two vectors being multiplied, as shown in Figure 5.13. The cross product operation is only defined in

图 5.14. 平行四边形的面积表示为叉积的大小。

三个维度：

$$\begin{aligned}\mathbf{a}\times\mathbf{b} &= \left[(a_yb_z-a_zb_y),\,(a_zb_x-a_xb_z),\,(a_xb_y-a_yb_x)\right]\\ &= (a_yb_z-a_zb_y)\mathbf{i}+(a_zb_x-a_xb_z)\mathbf{j}+(a_xb_y-a_yb_x)\mathbf{k}.\end{aligned}$$

数字 5.13. 这叉 产品向量 **a** 和 **b**（右撇子）。

叉积的大小

叉积向量的幅值等于两个向量幅值与它们夹角的正弦值的乘积。（这与点积的定义类似，只是用正弦值代替了余弦值。）

$$|\mathbf{a}\times\mathbf{b}| = |\mathbf{a}|\,|\mathbf{b}|\sin\theta.$$

如图 5.14 所示，叉积 | a × b | 的大小等于边长为 a 和 b 的平行四边形的面积。由于三角形是平行四边形的一半，因此，顶点分别由位置向量 V 1 、V 2 和 V 3 指定的三角形的面积可以计算为其任意两条边的叉积大小的一半：

$$\text{三角形} = 1 \qquad \left|(\mathbf{V}_2-\mathbf{V}_1)\times(\mathbf{V}_3-\mathbf{V}_1)\right|.$$

叉积的方向

使用右手坐标系时，可以使用右手定则来确定叉积的方向。只需将手指弯曲，使其指向旋转向量 a 使其移动到向量 b 上方的方向，叉积（a × b）的方向就会指向拇指的方向。

请注意，在使用左手坐标系时，叉积遵循左手定则。这意味着叉积的方向会根据坐标系的选择而变化。这看起来可能有点奇怪。

Figure 5.14. Area of a parallelogram expressed as the magnitude of a cross product.

three dimensions:

$$\begin{aligned}\mathbf{a}\times\mathbf{b} &= \left[(a_yb_z-a_zb_y),\,(a_zb_x-a_xb_z),\,(a_xb_y-a_yb_x)\right]\\ &= (a_yb_z-a_zb_y)\mathbf{i}+(a_zb_x-a_xb_z)\mathbf{j}+(a_xb_y-a_yb_x)\mathbf{k}.\end{aligned}$$

Figure 5.13. The cross product of vectors **a** and **b** (right-handed).

Magnitude of the Cross Product

The magnitude of the cross product vector is the product of the magnitudes of the two vectors and the sine of the angle between them. (This is similar to the definition of the dot product, but it replaces the cosine with the sine.)

$$|\mathbf{a}\times\mathbf{b}| = |\mathbf{a}|\,|\mathbf{b}|\sin\theta.$$

The magnitude of the cross product $|\mathbf{a}\times\mathbf{b}|$ is equal to the area of the parallelogram whose sides are **a** and **b**, as shown in Figure 5.14. Since a triangle is one half of a parallelogram, the area of a triangle whose vertices are specified by the position vectors $\mathbf{V}_1$, $\mathbf{V}_2$ and $\mathbf{V}_3$ can be calculated as one half of the magnitude of the cross product of any two of its sides:

$$A_{\text{triangle}} = \tfrac{1}{2}\left|(\mathbf{V}_2-\mathbf{V}_1)\times(\mathbf{V}_3-\mathbf{V}_1)\right|.$$

Direction of the Cross Product

When using a right-handed coordinate system, you can use the *right-hand rule* to determine the direction of the cross product. Simply cup your fingers such that they point in the direction you'd rotate vector **a** to move it on top of vector **b**, and the cross product $(\mathbf{a}\times\mathbf{b})$ will be in the direction of your thumb.

Note that the cross product is defined by the *left-hand rule* when using a left-handed coordinate system. This means that the direction of the cross product changes depending on the choice of coordinate system. This might seem odd

起初，但请记住，坐标系的惯用手性并不影响我们进行的数学计算——它只会改变我们对数字在三维空间中的可视化效果。当从右手系转换为左手系或反之亦然时，所有点和向量的数值表示保持不变，但其中一个轴会翻转。因此，我们对一切的可视化都会沿着翻转的轴进行镜像。所以，如果叉积恰好与我们正在翻转的轴（例如 z 轴）对齐，那么当轴翻转时，它需要翻转。如果不是，则必须更改叉积本身的数学定义，以使叉积的 z 坐标在新坐标系中为负值。我不会为这一切而失眠。只要记住：在可视化叉积时，在右手坐标系中使用右手定则，在左手坐标系中使用左手定则。

叉积的性质

叉积不具有交换性（即顺序很重要）：

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}.$$

然而，它是反交换的：

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

叉积对于加法来说是可分配的：

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c}).$$

它与标量乘法的组合如下：

$$(s\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (s\mathbf{b}) = s(\mathbf{a} \times \mathbf{b}).$$

笛卡尔基向量通过叉积相互关联，如下所示：

$$\begin{aligned} \mathbf{i} \times \mathbf{j} &= -(\mathbf{j} \times \mathbf{i}) = \mathbf{k} \\ \mathbf{j} \times \mathbf{k} &= -(\mathbf{k} \times \mathbf{j}) = \mathbf{i} \\ \mathbf{k} \times \mathbf{i} &= -(\mathbf{i} \times \mathbf{k}) = \mathbf{j} \end{aligned}$$

这三个叉积定义了绕笛卡尔坐标轴正向旋转的方向。正向旋转的方向是从 x 到 y（绕 z 轴），从 y 到 z（绕 x 轴），以及从 z 到 x（绕 y 轴）。注意绕 y 轴的旋转是如何按字母顺序“反转”的，即从 z 到 x（而不是从 x 到 z）。正如我们将要看到的

at first, but remember that the handedness of a coordinate system does *not* affect the mathematical calculations we carry out—it only changes our *visualization* of what the numbers look like in 3D space. When converting from a right-handed system to a left-handed system or vice versa, the numerical representations of all the points and vectors stay the same, but one axis flips. Our visualization of everything is therefore mirrored along that flipped axis. So if a cross product just happens to align with the axis we're flipping (e.g., the z-axis), it needs to flip when the axis flips. If it didn't, the mathematical definition of the cross product itself would have to be changed so that the z-coordinate of the cross product comes out negative in the new coordinate system. I wouldn't lose too much sleep over all of this. Just remember: when *visualizing* a cross product, use the right-hand rule in a right-handed coordinate system and the left-hand rule in a left-handed coordinate system.

Properties of the Cross Product

The cross product is *not commutative* (i.e., order matters):

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}.$$

However, it is *anti-commutative:*

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

The cross product is distributive over addition:

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c}).$$

And it combines with scalar multiplication as follows:

$$(s\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (s\mathbf{b}) = s(\mathbf{a} \times \mathbf{b}).$$

The Cartesian basis vectors are related by cross products as follows:

$$\begin{aligned} \mathbf{i} \times \mathbf{j} &= -(\mathbf{j} \times \mathbf{i}) = \mathbf{k} \\ \mathbf{j} \times \mathbf{k} &= -(\mathbf{k} \times \mathbf{j}) = \mathbf{i} \\ \mathbf{k} \times \mathbf{i} &= -(\mathbf{i} \times \mathbf{k}) = \mathbf{j} \end{aligned}$$

These three cross products define the direction of *positive rotations* about the Cartesian axes. The positive rotations go from x to y (about z), from y to z (about x) and from z to x (about y). Notice how the rotation about the y-axis "reversed" alphabetically, in that it goes from z to x (not from x to z). As we'll

见下文，这给了我们一个提示，为什么绕 y 轴旋转的矩阵与绕 x 轴和 z 轴旋转的矩阵相比看起来是反转的。

叉积的实际应用

叉积在游戏中有许多应用。其最常见的用途之一是用于找到与另外两个向量垂直的向量。正如我们将在 5.3.10.2 节中看到的，如果我们知道一个物体的局部单位基向量（ilocal、jlocal 和 klocal），我们可以轻松找到一个表示物体方向的矩阵。假设我们只知道物体的 klocal 向量 - 即物体面向的方向。如果我们假设物体没有围绕 klocal 的滚动，那么我们可以通过计算 klocal（我们已经知道）和世界空间向上向量 jworld（等于 [0 1 0]）的叉积来找到 ilocal。我们这样做如下：ilocal = normalize（jworld×klocal）。然后我们可以通过将 ilocal 和 k local 相交来找到 jlocal，如下所示：jlocal = klocal×ilocal。

可以使用非常类似的技术来查找三角形或其他平面表面的法向量。给定平面上的三个点 P 1 ,) P 2 和 P 3 ，法向量就是 n = normalize ((P 2 − P 1) × (P 3 − P 1) 。

叉积也用于物理模拟。当力作用于物体时，当且仅当力施加在物体中心以外时，物体才会产生旋转运动。这种旋转力被称为扭矩，其计算方法如下：给定力 F ，以及从质心到力施加点的矢量 r ，则扭矩 N = r × F 。

5.2.4.9 伪向量和外代数

我们在 5.2.2 节中提到，叉积实际上并不产生向量，而是产生一种称为伪向量的特殊数学对象。向量和伪向量之间的区别非常微妙。事实上，在执行我们在游戏编程中通常遇到的变换（平移、旋转和缩放）时，你根本无法分辨它们之间的区别。只有当你反射坐标系时（就像从左手坐标系移动到右手坐标系时发生的情况），伪向量的特殊性质才会显现出来。在反射下，向量会变换为其镜像，这可能与你所期望的一样。但当伪向量被反射时，它会变换为其镜像并且也会改变方向。

位置及其所有导数（线速度、加速度、加加速度）均用真矢量（也称为极矢量或逆变矢量）表示。角速度和磁场用伪矢量表示。

see below, this gives us a hint as to why the matrix for rotation about the y-axis looks *inverted* when compared to the matrices for rotation about the x- and z-axes.

The Cross Product in Action

The cross product has a number of applications in games. One of its most common uses is for finding a vector that is perpendicular to two other vectors. As we'll see in Section 5.3.10.2, if we know an object's local unit basis vectors, ($\mathbf{i}_{\text{local}}$, $\mathbf{j}_{\text{local}}$ and $\mathbf{k}_{\text{local}}$), we can easily find a matrix representing the object's orientation. Let's assume that all we know is the object's $\mathbf{k}_{\text{local}}$ vector—i.e., the direction in which the object is facing. If we assume that the object has no roll about $\mathbf{k}_{\text{local}}$, then we can find $\mathbf{i}_{\text{local}}$ by taking the cross product between $\mathbf{k}_{\text{local}}$ (which we already know) and the world-space up vector $\mathbf{j}_{\text{world}}$ (which equals $[0\ 1\ 0]$). We do so as follows: $\mathbf{i}_{\text{local}} = \text{normalize}(\mathbf{j}_{\text{world}} \times \mathbf{k}_{\text{local}})$. We can then find $\mathbf{j}_{\text{local}}$ by simply crossing $\mathbf{i}_{\text{local}}$ and $\mathbf{k}_{\text{local}}$ as follows: $\mathbf{j}_{\text{local}} = \mathbf{k}_{\text{local}} \times \mathbf{i}_{\text{local}}$.

A very similar technique can be used to find a unit vector normal to the surface of a triangle or some other plane. Given three points on the plane, $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$, the normal vector is just $\mathbf{n} = \text{normalize}\big((\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)\big)$.

Cross products are also used in physics simulations. When a force is applied to an object, it will give rise to rotational motion if and only if it is applied off-center. This rotational force is known as a *torque*, and it is calculated as follows. Given a force $\mathbf{F}$, and a vector $\mathbf{r}$ from the center of mass to the point at which the force is applied, the torque $\mathbf{N} = \mathbf{r} \times \mathbf{F}$.

5.2.4.9 Pseudovectors and Exterior Algebra

We mentioned in Section 5.2.2 that the cross product doesn't actually produce a vector—it produces a special kind of mathematical object known as a *pseudovector*. The difference between a vector and a pseudovector is pretty subtle. In fact, you can't tell the difference between them at all when performing the kinds of transformations we normally encounter in game programming—translation, rotation and scaling. It's only when you *reflect* the coordinate system (as happens when you move from a left-handed coordinate system to a right-handed system) that the special nature of pseudovectors becomes apparent. Under reflection, a vector transforms into its mirror image, as you'd probably expect. But when a pseudovector is reflected, it transforms into its mirror image and also *changes direction*.

Positions and all of the derivatives thereof (linear velocity, acceleration, jerk) are represented by true vectors (also known as *polar vectors* or *contravariant vectors*). Angular velocities and magnetic fields are represented by pseu-

图 5.15 在外代数（Grassman 代数）中，单个楔积产生伪向量或双向量，两个楔积产生伪标量或三向量。

伪向量（也称为轴向量、协变向量、双向量或2叶向量）。三角形的表面法线（使用叉积计算）也是一个伪向量。

值得注意的是，叉积（A × B）、标量三重积（A ·（B × C））和矩阵的行列式都是相互关联的，而伪向量是这一切的核心。数学家们提出了一套代数规则，称为外代数或格拉斯曼代数，它描述了向量和伪向量的工作原理，并使我们能够计算平行四边形（二维）的面积、平行六面体的体积（三维）等等，尤其是在高维空间中。

我们不会在这里深入讨论所有细节，但格拉斯曼代数的基本思想是引入一种特殊的向量积，称为楔积，记为 A ∧ B 。成对楔积会产生一个伪向量，它等价于叉积，也表示由两个向量构成的平行四边形的带符号面积（其中符号告诉我们是从 A 旋转到 B 还是从 B 旋转到 A）。连续进行两次楔积，即 A ∧ B ∧ C ，

Figure 5.15. In the exterior algebra (Grassman algebra), a single wedge product yields a pseudovector or bivector, and two wedge products yield a pseudoscalar or trivector.

dovectors (also known as *axial vectors*, *covariant vectors*, *bivectors* or *2-blades*). The surface normal of a triangle (which is calculated using a cross product) is also a pseudovector.

It's pretty interesting to note that the cross product ($\mathbf{A} \times \mathbf{B}$), the scalar triple product ($\mathbf{A} \cdot (\mathbf{B} \times \mathbf{C})$) and the determinant of a matrix are all inter-related, and pseudovectors lie at the heart of it all. Mathematicians have come up with a set of algebraic rules, called an *exterior algebra* or *Grassman algebra*, which describe how vectors and pseudovectors work and allow us to calculate areas of parallelograms (in 2D), volumes of parallelepipeds (in 3D), and so on in higher dimensions.

We won't get into all the details here, but the basic idea of Grassman algebra is to introduce a special kind of vector product known as the *wedge product*, denoted $\mathbf{A} \wedge \mathbf{B}$. A pairwise wedge product yields a pseudovector and is equivalent to a cross product, which also represents the *signed area* of the parallelogram formed by the two vectors (where the sign tells us whether we're rotating from $\mathbf{A}$ to $\mathbf{B}$ or vice versa). Doing two wedge products in a row, $\mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$,

等价于标量三重积 A ·（B × C），并产生另一个称为伪标量（也称为三向量或三叶）的奇特数学对象，它可以解释为由三个向量构成的平行六面体的有符号体积（参见图 5.15）。这也延伸到更高的维度。

这一切对我们游戏程序员来说意味着什么？意义不大。我们真正需要记住的是，代码中的一些向量实际上是伪向量，这样我们才能在改变惯用手性的时候正确地对它们进行变换。当然，如果你真的想成为极客，你可以和朋友们讨论外代数和楔积，并解释叉积实际上不是向量，以此来打动他们。这可能会让你在下次社交活动中看起来很酷……也可能不会。

有关更多信息，请参阅http://en.wikipedia.org/wiki/Pseudovector、http://en.wikipedia.org/wiki/Exterior_algebra 和 http://www.terathon.com/gdc12_lengyel.pdf 。

5.2.5 点和向量的线性插值

在游戏中，我们经常需要找到一个位于两个已知向量中间的向量。例如，如果我们想以每秒 30 帧的速度，在两秒内让一个物体从 A 点平滑地移动到 B 点，我们就需要在 A 点和 B 点之间找到 60 个中间位置。

线性插值是一种简单的数学运算，用于在两个已知点之间找到一个中间点。该运算的名称通常缩写为 LERP。该运算定义如下，其中 β 的取值范围为 0 到 1（含 0 和 1）：

$$\begin{aligned} \mathrm{L} = \mathrm{LERP}(\mathrm{A}, \mathrm{B}, \beta) &= (1-\beta)\,\mathrm{A} + \beta\,\mathrm{B} \\ &= \left[(1-\beta)A_x + \beta B_x, \quad (1-\beta)A_y + \beta B_y, \quad (1-\beta)A_z + \beta B_z\right] \end{aligned}$$

从几何学上讲，L = LERP (A , B , β) 表示位于点 A 到点 B 的线段上 β % 位置的点的位置向量，如图 5.16 所示。从数学上讲，LERP 函数只是两个输入向量的加权平均值，权重分别为 (1 − β) 和 β 。注意，权重之和始终为 1，这是任何加权平均值的一般要求。

5.3 矩阵

矩阵是由 m × n 个标量组成的矩形阵列。矩阵是表示平移、旋转和缩放等线性变换的便捷方式。

is equivalent to the scalar triple product $\mathbf{A} \cdot (\mathbf{B} \times \mathbf{C})$ and produces another strange mathematical object known as a *pseudoscalar* (also known as a *trivector* or a *3-blade*), which can be interpreted as the *signed volume* of the parallelepiped formed by the three vectors (see Figure 5.15). This extends into higher dimensions as well.

What does all this mean for us as game programmers? Not too much. All we really need to keep in mind is that some vectors in our code are actually pseudovectors, so that we can transform them properly when changing handedness, for example. Of course if you really want to geek out, you can impress your friends by talking about *exterior algebras* and *wedge products* and explaining how cross products aren't really vectors. Which might make you look cool at your next social engagement …or not.

For more information, see http://en.wikipedia.org/wiki/Pseudovector, http://en.wikipedia.org/wiki/Exterior_algebra, and http://www.terathon.com/gdc12_lengyel.pdf.

5.2.5 Linear Interpolation of Points and Vectors

In games, we often need to find a vector that is midway between two known vectors. For example, if we want to smoothly animate an object from point **A** to point **B** over the course of two seconds at 30 frames per second, we would need to find 60 intermediate positions between **A** and **B**.

A *linear interpolation* is a simple mathematical operation that finds an intermediate point between two known points. The name of this operation is often shortened to LERP. The operation is defined as follows, where β ranges from 0 to 1 inclusive:

$$\begin{aligned} \mathbf{L} = \mathrm{LERP}(\mathbf{A}, \mathbf{B}, \beta) &= (1-\beta)\mathbf{A} + \beta\mathbf{B} \\ &= \left[(1-\beta)A_x + \beta B_x, \quad (1-\beta)A_y + \beta B_y, \quad (1-\beta)A_z + \beta B_z\right] \end{aligned}$$

Geometrically, $\mathbf{L} = \mathrm{LERP}(\mathbf{A}, \mathbf{B}, \beta)$ is the position vector of a point that lies β percent of the way along the line segment from point **A** to point **B**, as shown in Figure 5.16. Mathematically, the LERP function is just a *weighted average* of the two input vectors, with weights $(1-\beta)$ and β, respectively. Notice that the weights always add to 1, which is a general requirement for any weighted average.

5.3 Matrices

A *matrix* is a rectangular array of $m \times n$ scalars. Matrices are a convenient way of representing linear transformations such as translation, rotation and scale.

图 5.16. 点 A 和 B 之间的线性插值 (LERP)，其中 β = 0.4。

矩阵 M 通常写成方括号中标量 M rc 的网格形式，其中下标 r 和 c 分别代表元素的行和列索引。例如，如果 M 是一个 3 × 3 矩阵，则可以写成如下形式：

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}.$$

我们可以将 3×3 矩阵的行和/或列视为 3D 向量。
当一个 3 × 3 矩阵的所有行向量和列向量均为单位幅值时，我们称之为特殊正交矩阵。这种矩阵也称为各向同性矩阵或标准正交矩阵。这类矩阵表示纯旋转。

在特定约束条件下，4 × 4 矩阵可以表示任意三维变换，包括平移、旋转和缩放。这些变换被称为变换矩阵，它们对于游戏工程师来说非常有用。矩阵表示的变换通过矩阵乘法应用于点或向量。我们将在下文探讨其工作原理。

仿射矩阵是一个 4 × 4 的变换矩阵，它保留了直线的平行度和相对距离比，但不一定保留绝对长度和角度。仿射矩阵是以下操作的任意组合：旋转、平移、缩放和/或剪切。

5.3.1 矩阵乘法

两个矩阵 A 和 B 的乘积 P 记作 P = AB 。如果 A 和 B 是变换矩阵，则乘积 P 是另一个执行两个原始变换的变换矩阵。例如，如果 A 是缩放矩阵，B 是旋转矩阵，则矩阵 P 会同时缩放和旋转应用该变换的点或向量。这在游戏编程中特别有用，因为我们可以预先计算一个执行一系列变换的矩阵，然后将所有这些变换高效地应用于大量向量。

Figure 5.16. Linear interpolation (LERP) between points **A** and **B**, with $\beta = 0.4$.

A matrix **M** is usually written as a grid of scalars M_{rc} enclosed in square brackets, where the subscripts r and c represent the row and column indices of the entry, respectively. For example, if **M** is a 3×3 matrix, it could be written as follows:

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}.$$

We can think of the rows and/or columns of a 3×3 matrix as 3D vectors. When all of the row and column vectors of a 3×3 matrix are of unit magnitude, we call it a *special orthogonal* matrix. This is also known as an *isotropic* matrix, or an *orthonormal* matrix. Such matrices represent pure rotations.

Under certain constraints, a 4×4 matrix can represent arbitrary 3D *transformations*, including *translations*, *rotations*, and changes in *scale*. These are called *transformation matrices*, and they are the kinds of matrices that will be most useful to us as game engineers. The transformations represented by a matrix are applied to a point or vector via matrix multiplication. We'll investigate how this works below.

An *affine* matrix is a 4×4 transformation matrix that preserves parallelism of lines and relative distance ratios, but not necessarily absolute lengths and angles. An affine matrix is any combination of the following operations: rotation, translation, scale and/or shear.

5.3.1 Matrix Multiplication

The product **P** of two matrices **A** and **B** is written $\mathbf{P} = \mathbf{AB}$. If **A** and **B** are transformation matrices, then the product **P** is another transformation matrix that performs *both* of the original transformations. For example, if **A** is a scale matrix and **B** is a rotation, the matrix **P** would both scale *and* rotate the points or vectors to which it is applied. This is particularly useful in game programming, because we can precalculate a single matrix that performs a whole sequence of transformations and then apply all of those transformations to a large number of vectors efficiently.

要计算矩阵积，我们只需对 n A × m A 矩阵 A 的行和 n B × m B 矩阵 B 的列进行点积即可。每个点积都成为结果矩阵 P 的一个分量。只要内部维度相等（即 m A = n B），这两个矩阵就可以相乘。例如，如果 A 和 B 都是 3 × 3 矩阵，则 P = AB 可以表示如下：

$$\begin{aligned}\mathbf{P} &= \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} \\ &= \begin{bmatrix} \text{A 行 1 · B 列 1} & \text{A 行 1 · B 列 2} & \text{A 行 1 · B 列 3} \\ \text{A 行 2 · B 列 1} & \text{A 行 2 · B 列 2} & \text{A 行 2 · B 列 3} \\ \text{A 行 3 · B 列 1} & \text{A 行 3 · B 列 2} & \text{A 行 3 · B 列 3} \end{bmatrix}.\end{aligned}$$

矩阵乘法不满足交换律。换句话说，矩阵乘法的执行顺序很重要：

$$\mathbf{AB} \neq \mathbf{BA}$$

我们将在第 5.3.2 节中了解为什么这很重要。
矩阵乘法通常称为连接，因为 n 个变换矩阵的乘积是一个按照矩阵相乘的顺序连接或链接原始变换序列的矩阵。

5.3.2 将点和向量表示为矩阵

点和向量可以表示为行矩阵（1 × n）或列矩阵（n × 1），其中 n 是我们所处理空间的维度（通常为 2 或 3）。例如，向量 v = (3, 4, − 1) 可以写成

$$\mathbf{v}_1 = \begin{bmatrix} 3 & 4 & -1 \end{bmatrix}$$

or as

$$\mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ -1 \end{bmatrix} = \mathbf{v}_1^{\mathrm{T}}.$$

这里，上标 T 代表矩阵转置（参见第 5.3.5 节）。
列向量和行向量的选择完全是任意的，但它确实会影响矩阵乘法的书写顺序。这是因为在矩阵相乘时，两个矩阵的内部维度必须相等，所以

To calculate a matrix product, we simply take dot products between the rows of the $n_{\mathrm{A}} \times m_{\mathrm{A}}$ matrix $\mathbf{A}$ and the columns of the $n_{\mathrm{B}} \times m_{\mathrm{B}}$ matrix $\mathbf{B}$. Each dot product becomes one component of the resulting matrix $\mathbf{P}$. The two matrices can be multiplied as long as the *inner dimensions* are equal (i.e., $m_{\mathrm{A}} = n_{\mathrm{B}}$). For example, if $\mathbf{A}$ and $\mathbf{B}$ are 3×3 matrices, then $\mathbf{P} = \mathbf{AB}$ may be expressed as follows:

$$\begin{aligned}\mathbf{P} &= \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{\mathrm{row1}} \cdot \mathbf{B}_{\mathrm{col1}} & \mathbf{A}_{\mathrm{row1}} \cdot \mathbf{B}_{\mathrm{col2}} & \mathbf{A}_{\mathrm{row1}} \cdot \mathbf{B}_{\mathrm{col3}} \\ \mathbf{A}_{\mathrm{row2}} \cdot \mathbf{B}_{\mathrm{col1}} & \mathbf{A}_{\mathrm{row2}} \cdot \mathbf{B}_{\mathrm{col2}} & \mathbf{A}_{\mathrm{row2}} \cdot \mathbf{B}_{\mathrm{col3}} \\ \mathbf{A}_{\mathrm{row3}} \cdot \mathbf{B}_{\mathrm{col1}} & \mathbf{A}_{\mathrm{row3}} \cdot \mathbf{B}_{\mathrm{col2}} & \mathbf{A}_{\mathrm{row3}} \cdot \mathbf{B}_{\mathrm{col3}} \end{bmatrix}.\end{aligned}$$

Matrix multiplication is not commutative. In other words, the order in which matrix multiplication is done matters:

$$\mathbf{AB} \neq \mathbf{BA}$$

We'll see exactly why this matters in Section 5.3.2.

Matrix multiplication is often called *concatenation*, because the product of n transformation matrices is a matrix that concatenates, or chains together, the original sequence of transformations in the order the matrices were multiplied.

5.3.2 Representing Points and Vectors as Matrices

Points and vectors can be represented as *row matrices* $(1 \times n)$ or *column matrices* $(n \times 1)$, where n is the dimension of the space we're working with (usually 2 or 3). For example, the vector $\mathbf{v} = (3, 4, -1)$ can be written either as

$$\mathbf{v}_1 = \begin{bmatrix} 3 & 4 & -1 \end{bmatrix}$$

or as

$$\mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ -1 \end{bmatrix} = \mathbf{v}_1^{\mathrm{T}}.$$

Here, the superscripted T represents matrix *transposition* (see Section 5.3.5).

The choice between column and row vectors is a completely arbitrary one, but it does affect the order in which matrix multiplications are written. This happens because when multiplying matrices, the inner dimensions of the two matrices must be equal, so

- 将 1 × n 行向量乘以 n × n 矩阵，该向量必须出现在矩阵的左侧（v′ 1 × n = v 1 × n M n × n），而
- 要将 n × n 矩阵与 n × 1 列向量相乘，该向量必须出现在矩阵的右侧（v′ n×1=Mn×nvn×1）。

如果将多个变换矩阵 A 、 B 和 C 依次应用于向量 v ，则当使用行向量时，变换从左到右“读取”，而当使用列向量时，变换则从右到左“读取”。记住这一点最简单的方法是首先应用最接近向量的矩阵。如下方括号所示：

行向量：从左到右阅读；v′ T = (CT (BT (AT v T))) 列向量：从右到左阅读。

在本书中，我们将采用行向量约定，因为对于英语使用者来说，从左到右的变换顺序最直观易读。不过，请务必仔细检查你的游戏引擎以及你可能阅读的其他书籍、论文或网页所使用的约定。通常，你可以通过查看向量矩阵乘法的书写方式来判断：向量写在矩阵的左侧（对于行向量），还是右侧（对于列向量）。使用列向量时，你需要对本书中介绍的所有矩阵进行转置。

5.3.3 单位矩阵

单位矩阵是指与任何其他矩阵相乘都会得到完全相同矩阵的矩阵。它通常用符号 I 表示。单位矩阵始终是一个方阵，对角线上元素为 1，其余元素为 0：

$$\mathbf{I}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix};$$
$$\mathbf{AI} = \mathbf{IA} \equiv \mathbf{A}.$$

5.3.4 矩阵求逆

矩阵 A 的逆矩阵是另一个矩阵（记为 A − 1 ），它抵消了矩阵 A 的影响。例如，如果 A 将物体绕 z 轴旋转 37 度，那么 A − 1 也会绕 z 轴旋转 − 37 度。同样，如果 A 将物体缩放到原始大小的两倍，那么 A − 1 会将物体缩放到原始大小的一半。当矩阵乘以其自身的逆矩阵时，结果始终是单位矩阵，因此 A (A − 1) ≡ (A − 1) A ≡ I 。并非所有矩阵都具有

- to multiply a $1 \times n$ row vector by an $n \times n$ matrix, the vector must appear to the *left* of the matrix ($\mathbf{v}'_{1\times n} = \mathbf{v}_{1\times n}\,\mathbf{M}_{n\times n}$), whereas
- to multiply an $n \times n$ matrix by an $n \times 1$ column vector, the vector must appear to the *right* of the matrix ($\mathbf{v}'_{n\times 1} = \mathbf{M}_{n\times n}\,\mathbf{v}_{n\times 1}$).

If multiple transformation matrices **A**, **B** and **C** are applied in order to a vector **v**, the transformations "read" from *left to right* when using *row* vectors, but from *right to left* when using *column* vectors. The easiest way to remember this is to realize that the matrix *closest* to the vector is applied first. This is illustrated by the parentheses below:

$$\mathbf{v}' = (((\mathbf{v}\mathbf{A})\mathbf{B})\mathbf{C}) \qquad \text{Row vectors: read left-to-right;}$$
$$\mathbf{v}'^{\mathrm{T}} = (\mathbf{C}^{\mathrm{T}}(\mathbf{B}^{\mathrm{T}}(\mathbf{A}^{\mathrm{T}}\mathbf{v}^{\mathrm{T}}))) \qquad \text{Column vectors: read right-to-left.}$$

In this book we'll adopt the *row vector convention*, because the left-to-right order of transformations is most intuitive to read for English-speaking people. That said, be very careful to check which convention is used by your game engine, and by other books, papers or web pages you may read. You can usually tell by seeing whether vector-matrix multiplications are written with the vector on the left (for row vectors) or the right (for column vectors) of the matrix. When using column vectors, you'll need to *transpose* all the matrices shown in this book.

5.3.3 The Identity Matrix

The *identity matrix* is a matrix that, when multiplied by any other matrix, yields the very same matrix. It is usually represented by the symbol **I**. The identity matrix is always a square matrix with 1's along the diagonal and 0's everywhere else:

$$\mathbf{I}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix};$$
$$\mathbf{AI} = \mathbf{IA} \equiv \mathbf{A}.$$

5.3.4 Matrix Inversion

The *inverse* of a matrix **A** is another matrix (denoted $\mathbf{A}^{-1}$) that *undoes* the effects of matrix **A**. So, for example, if **A** rotates objects by 37 degrees about the z-axis, then $\mathbf{A}^{-1}$ will rotate by -37 degrees about the z-axis. Likewise, if **A** scales objects to be twice their original size, then $\mathbf{A}^{-1}$ scales objects to be half-sized. When a matrix is multiplied by its own inverse, the result is *always* the identity matrix, so $\mathbf{A}(\mathbf{A}^{-1}) \equiv (\mathbf{A}^{-1})\mathbf{A} \equiv \mathbf{I}$. Not all matrices have

逆矩阵。然而，所有仿射矩阵（纯旋转、平移、缩放和剪切的组合）都有逆矩阵。如果存在逆矩阵，可以使用高斯消元法或上下（LU）分解来求逆矩阵。

由于我们接下来会经常处理矩阵乘法，因此需要注意的是，串联矩阵序列的逆可以写成各个矩阵逆的反向串联。例如：

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}.$$

5.3.5 转置

矩阵 M 的转置记为 MT 。它是通过将原矩阵的元素沿其对角线反射得到的。换句话说，原矩阵的行成为转置矩阵的列，反之亦然：

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}.$$

转置之所以有用，有很多原因。首先，正交（纯旋转）矩阵的逆恰好等于其转置——这是一个好消息，因为转置矩阵比求逆通常要便宜得多。在将数据从一个数学库移动到另一个数学库时，转置也很重要，因为有些库使用列向量，而有些库则需要行向量。基于行向量的库使用的矩阵将相对于采用列向量约定的库使用的矩阵进行转置。

与逆矩阵一样，串联矩阵序列的转置可以重写为各个矩阵转置的反向串联。例如：

$$(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}.$$

当我们考虑如何将变换矩阵应用于点和向量时，这将被证明是有用的。

5.3.6 齐次坐标

你可能还记得高中代数课上讲过，2×2 矩阵可以表示二维旋转。要将向量 r 旋转 φ 度（其中正向旋转为逆时针），我们可以写成

$$\begin{bmatrix} r'_x & r'_y \end{bmatrix} = \begin{bmatrix} r_x & r_y \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}.$$

inverses. However, all *affine* matrices (combinations of pure rotations, translations, scales and shears) do have inverses. Gaussian elimination or lower-upper (LU) decomposition can be used to find the inverse, if one exists.

Since we'll be dealing with matrix multiplication a lot, it's important to note here that the inverse of a sequence of concatenated matrices can be written as the *reverse concatenation* of the individual matrices' inverses. For example,

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}.$$

5.3.5 Transposition

The *transpose* of a matrix $\mathbf{M}$ is denoted $\mathbf{M}^{\mathrm{T}}$. It is obtained by reflecting the entries of the original matrix across its diagonal. In other words, the rows of the original matrix become the columns of the transposed matrix, and vice versa:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}.$$

The transpose is useful for a number of reasons. For one thing, the inverse of an orthonormal (pure rotation) matrix is exactly equal to its transpose—which is good news, because it's much cheaper to transpose a matrix than it is to find its inverse in general. Transposition can also be important when moving data from one math library to another, because some libraries use column vectors while others expect row vectors. The matrices used by a row-vector–based library will be *transposed* relative to those used by a library that employs the column vector convention.

As with the inverse, the transpose of a sequence of concatenated matrices can be rewritten as the reverse concatenation of the individual matrices' transposes. For example,

$$(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}.$$

This will prove useful when we consider how to apply transformation matrices to points and vectors.

5.3.6 Homogeneous Coordinates

You may recall from high-school algebra that a 2×2 matrix can represent a rotation in two dimensions. To rotate a vector $\mathbf{r}$ through an angle of ϕ degrees (where positive rotations are counterclockwise), we can write

$$\begin{bmatrix} r'_x & r'_y \end{bmatrix} = \begin{bmatrix} r_x & r_y \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}.$$

三维空间中的旋转可以用 3 × 3 矩阵表示，这或许并不奇怪。上面的二维示例实际上只是绕 z 轴的三维旋转，因此我们可以写成

$$\begin{bmatrix} r'_x & r'_y & r'_z \end{bmatrix} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

问题自然而然地出现了：一个 3 × 3 的矩阵能用来表示平移吗？很遗憾，答案是否定的。将点 r 平移 t 的结果，需要将 t 的分量分别添加到 r 的分量中：

$$\mathbf{r} + \mathbf{t} = \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) \end{bmatrix}.$$

矩阵乘法涉及矩阵元素的乘法和加法，因此使用乘法进行平移的想法似乎很有前景。但不幸的是，没有办法将 t 的分量排列在一个 3 × 3 矩阵中，使得它与列向量 r 相乘的结果如下：

$(r_x + t_x)$.

好消息是，如果我们使用 4×4 矩阵，我们可以获得这样的总和。这样的矩阵会是什么样子？我们知道我们不想要任何旋转效应，所以上面的 3 × 3 应该包含一个单位矩阵。如果我们将 t 的分量排列在矩阵的最底行，并将 r 向量的第四个元素（通常称为 w ）设置为 1，那么对向量 r 与矩阵第 1 列进行点积将得到 (1 · rx) + (0 · ry) + (0 · rz) + (tx · 1)，这正是我们想要的。如果矩阵的右下角包含 1，而第四列的其余部分包含零，那么结果向量的 w 分量中也将有一个 1。这是最终的 4 × 4 平移矩阵的样子：

$$\begin{aligned} \mathbf{r} + \mathbf{t} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \\ &= \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) & 1 \end{bmatrix}. \end{aligned}$$

当一个点或向量以这种方式从三维扩展到四维时，我们称它为齐次坐标系。齐次坐标系中的点的 w 始终等于 1。游戏引擎进行的大部分三维矩阵运算都是使用 4 × 4 矩阵，其中包含四元素点和向量，并以齐次坐标系表示。

It's probably no surprise that rotations in three dimensions can be represented by a 3×3 matrix. The two-dimensional example above is really just a three-dimensional rotation about the z-axis, so we can write

$$\begin{bmatrix} r'_x & r'_y & r'_z \end{bmatrix} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The question naturally arises: Can a 3×3 matrix be used to represent *translations*? Sadly, the answer is no. The result of translating a point **r** by a translation **t** requires adding the components of **t** to the components of **r** individually:

$$\mathbf{r} + \mathbf{t} = \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) \end{bmatrix}.$$

Matrix multiplication involves multiplication and addition of matrix elements, so the idea of using multiplication for translation seems promising. But, unfortunately, there is no way to arrange the components of **t** within a 3×3 matrix such that the result of multiplying it with the column vector **r** yields sums like $(r_x + t_x)$.

The good news is that we *can* obtain sums like this if we use a 4×4 matrix. What would such a matrix look like? Well, we know that we don't want any rotational effects, so the upper 3×3 should contain an identity matrix. If we arrange the components of **t** across the bottom-most row of the matrix and set the fourth element of the **r** vector (usually called w) equal to 1, then taking the dot product of the vector **r** with column 1 of the matrix will yield $(1 \cdot r_x) + (0 \cdot r_y) + (0 \cdot r_z) + (t_x \cdot 1)$, which is exactly what we want. If the bottom right-hand corner of the matrix contains a 1 and the rest of the fourth column contains zeros, then the resulting vector will also have a 1 in its w component. Here's what the final 4×4 translation matrix looks like:

$$\begin{aligned} \mathbf{r} + \mathbf{t} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \\ &= \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) & 1 \end{bmatrix}. \end{aligned}$$

When a point or vector is extended from three dimensions to four in this manner, we say that it has been written in *homogeneous coordinates*. A point in homogeneous coordinates always has $w = 1$. Most of the 3D matrix math done by game engines is performed using 4×4 matrices with four-element points and vectors written in homogeneous coordinates.

5.3.6.1 变换方向向量

从数学上讲，点（位置向量）和方向向量的处理方式略有不同。当用矩阵变换一个点时，矩阵的平移、旋转和缩放都会应用到该点上。但是，当用矩阵变换一个方向时，矩阵的平移效应会被忽略。这是因为方向向量本身并不平移——对方向进行平移会改变其大小，这通常不是我们想要的。

在齐次坐标系中，我们通过定义点的 w 分量等于 1，而方向向量的 w 分量等于 0 来实现这一点。在下面的例子中，请注意向量 v 的 w = 0 分量如何与矩阵中的 t 向量相乘，从而消除了最终结果中的平移：

$$\begin{bmatrix}\mathbf{v} & 0\end{bmatrix}\begin{bmatrix}\mathbf{U} & 0\\ \mathbf{t} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{v}\mathbf{U}+0t) & 0\end{bmatrix} = \begin{bmatrix}\mathbf{v}\mathbf{U} & 0\end{bmatrix}.$$

从技术上讲，齐次（四维）坐标中的点可以通过将 x 、 y 和 z 分量除以 w 分量来转换为非齐次（三维）坐标：

$$\begin{bmatrix}x & y & z & w\end{bmatrix} \equiv \begin{bmatrix}\frac{x}{w} & \frac{y}{w} & \frac{z}{w}\end{bmatrix}.$$

这解释了为什么我们将点的 w 分量设为 1，将向量的 w 分量设为 0。除以 w = 1 对点的坐标没有影响，但将纯方向向量的分量除以 w = 0 则会导致无穷大。四维空间中的无穷远点可以旋转，但不能平移，因为无论我们尝试应用什么平移，该点都将保持在无穷远。因此，实际上，三维空间中的纯方向向量的作用类似于四维均匀空间中的无穷远点。

5.3.7 原子变换矩阵

任何仿射变换矩阵都可以通过简单地连接一系列 4×4 矩阵来创建，这些矩阵分别表示纯平移、纯旋转、纯缩放操作和/或纯剪切。这些原子变换的构建块如下所示。（我们将在讨论中省略剪切操作，因为它在游戏中很少使用。）

请注意，所有仿射 4×4 变换矩阵都可以分为四个分量：

$$\mathbf{M}_{\text{affine}} = \begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix}.$$

5.3.6.1 Transforming Direction Vectors

Mathematically, points (position vectors) and direction vectors are treated in subtly different ways. When transforming a point by a matrix, the translation, rotation and scale of the matrix are all applied to the point. But when transforming a direction by a matrix, the *translational* effects of the matrix are ignored. This is because direction vectors have no translation per se—applying a translation to a direction would alter its magnitude, which is usually not what we want.

In homogeneous coordinates, we achieve this by defining points to have their w components equal to one, while direction vectors have their w components equal to *zero*. In the example below, notice how the $w = 0$ component of the vector $\mathbf{v}$ multiplies with the $\mathbf{t}$ vector in the matrix, thereby eliminating translation in the final result:

$$\begin{bmatrix}\mathbf{v} & 0\end{bmatrix}\begin{bmatrix}\mathbf{U} & 0\\ \mathbf{t} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{v}\mathbf{U}+0t) & 0\end{bmatrix} = \begin{bmatrix}\mathbf{v}\mathbf{U} & 0\end{bmatrix}.$$

Technically, a point in homogeneous (four-dimensional) coordinates can be converted into non-homogeneous (three-dimensional) coordinates by dividing the x, y and z components by the w component:

$$\begin{bmatrix}x & y & z & w\end{bmatrix} \equiv \begin{bmatrix}\frac{x}{w} & \frac{y}{w} & \frac{z}{w}\end{bmatrix}.$$

This sheds some light on why we set a point's w component to one and a vector's w component to zero. Dividing by $w = 1$ has no effect on the coordinates of a point, but dividing a pure direction vector's components by $w = 0$ would yield infinity. A point at infinity in 4D can be rotated but not translated, because no matter what translation we try to apply, the point will remain at infinity. So in effect, a pure direction vector in three-dimensional space acts like a point at infinity in four-dimensional homogeneous space.

5.3.7 Atomic Transformation Matrices

Any affine transformation matrix can be created by simply concatenating a sequence of 4×4 matrices representing pure translations, pure rotations, pure scale operations and/or pure shears. These atomic transformation building blocks are presented below. (We'll omit shear from these discussions, as it tends to be used only rarely in games.)

Notice that all affine 4×4 transformation matrices can be partitioned into four components:

$$\mathbf{M}_{\text{affine}} = \begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix}.$$

- 上面的 3 × 3 矩阵 U，表示旋转和/或缩放，
- 1 × 3 平移向量 t，[0
- 一个 3 × 1 的零向量 0 = 0 0 T, and
- 矩阵右下角的标量 1。

当一个点与一个已经这样划分的矩阵相乘时，结果如下：

$$\begin{bmatrix}\mathbf{r}'_{1\times3} & 1\end{bmatrix} = \begin{bmatrix}\mathbf{r}_{1\times3} & 1\end{bmatrix}\begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{r}\mathbf{U}+\mathbf{t}) & 1\end{bmatrix}.$$

5.3.7.1 翻译

以下矩阵通过向量 t 平移一个点：

$$\begin{aligned}\mathbf{r}+\mathbf{t} &= \begin{bmatrix}r_x & r_y & r_z & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ t_x & t_y & t_z & 1\end{bmatrix} \\ &= \begin{bmatrix}(r_x+t_x) & (r_y+t_y) & (r_z+t_z) & 1\end{bmatrix},\end{aligned} \tag{5.3}$$

或者以分区简写形式：

$$\begin{bmatrix}\mathbf{r} & 1\end{bmatrix}\begin{bmatrix}\mathbf{I} & 0\\ \mathbf{t} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{r}+\mathbf{t}) & 1\end{bmatrix}.$$

要反转纯平移矩阵，只需对向量 t 取反（即对 tx 取反，
t_y and t_z).

5.3.7.2 旋转

所有 4 × 4 纯旋转矩阵都有以下形式

$$\begin{bmatrix}\mathbf{r} & 1\end{bmatrix}\begin{bmatrix}\mathbf{R} & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}\mathbf{r}\mathbf{R} & 1\end{bmatrix}.$$

t 向量为零，上部 3 × 3 矩阵 R 包含旋转角度的余弦和正弦，以弧度为单位。

以下矩阵表示绕 x 轴旋转角度 φ 。

$$\mathrm{rotate}_x(\mathbf{r},\phi) = \begin{bmatrix}r_x & r_y & r_z & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\phi & \sin\phi & 0\\ 0 & -\sin\phi & \cos\phi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}. \tag{5.4}$$

- the upper 3 × 3 matrix **U**, which represents the rotation and/or scale,
- a 1 × 3 translation vector **t**,
- a 3 × 1 vector of zeros $0 = \begin{bmatrix}0 & 0 & 0\end{bmatrix}^{\mathrm{T}}$, and
- a scalar 1 in the bottom-right corner of the matrix.

When a point is multiplied by a matrix that has been partitioned like this, the result is as follows:

$$\begin{bmatrix}\mathbf{r}'_{1\times3} & 1\end{bmatrix} = \begin{bmatrix}\mathbf{r}_{1\times3} & 1\end{bmatrix}\begin{bmatrix}\mathbf{U}_{3\times3} & 0_{3\times1}\\ \mathbf{t}_{1\times3} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{r}\mathbf{U}+\mathbf{t}) & 1\end{bmatrix}.$$

5.3.7.1 Translation

The following matrix translates a point by the vector **t**:

$$\begin{aligned}\mathbf{r}+\mathbf{t} &= \begin{bmatrix}r_x & r_y & r_z & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ t_x & t_y & t_z & 1\end{bmatrix} \\ &= \begin{bmatrix}(r_x+t_x) & (r_y+t_y) & (r_z+t_z) & 1\end{bmatrix},\end{aligned} \tag{5.3}$$

or in partitioned shorthand:

$$\begin{bmatrix}\mathbf{r} & 1\end{bmatrix}\begin{bmatrix}\mathbf{I} & 0\\ \mathbf{t} & 1\end{bmatrix} = \begin{bmatrix}(\mathbf{r}+\mathbf{t}) & 1\end{bmatrix}.$$

To invert a pure translation matrix, simply negate the vector **t** (i.e., negate t_x, t_y and t_z).

5.3.7.2 Rotation

All 4 × 4 pure rotation matrices have the form

$$\begin{bmatrix}\mathbf{r} & 1\end{bmatrix}\begin{bmatrix}\mathbf{R} & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}\mathbf{r}\mathbf{R} & 1\end{bmatrix}.$$

The **t** vector is zero, and the upper 3 × 3 matrix **R** contains cosines and sines of the rotation angle, measured in radians.

The following matrix represents rotation about the x-axis by an angle ϕ.

$$\mathrm{rotate}_x(\mathbf{r},\phi) = \begin{bmatrix}r_x & r_y & r_z & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\phi & \sin\phi & 0\\ 0 & -\sin\phi & \cos\phi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}. \tag{5.4}$$

下面的矩阵表示绕 y 轴旋转角度 θ 。（请注意，这个矩阵相对于另外两个矩阵进行了转置——正负正弦项都沿对角线反射。）

$$\text{rotate}_y(\mathbf{r},\theta) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.5}$$

The following matrix represents rotation about the z-axis by an angle γ:

$$\text{rotate}_z(\mathbf{r},\gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.6}$$

以下是关于这些矩阵的一些观察结果：

- 上部 3 × 3 中的 1 始终出现在我们旋转的轴上，而正弦和余弦项则偏离轴。
- 正向旋转：从 x 到 y（绕 z 轴），从 y 到 z（绕 x 轴），以及从 z 到 x（绕 y 轴）。z 轴到 x 轴的旋转是“环绕”的，这就是为什么绕 y 轴的旋转矩阵相对于其他两个轴是转置的。

  （使用右手或左手规则来记住这一点。）
- 纯旋转的逆运算就是它的转置运算。这是因为反转旋转等同于旋转负角度。你可能还记得 cos（－θ）= cos（θ），而 sin（－θ）= － sin（θ），因此对角度取负会导致两个正弦项实际上交换位置，而余弦项保持不变。

5.3.7.3 规模

以下矩阵将点 r 沿 x 轴缩放 sx 倍，沿 y 轴缩放 sy 倍，沿 z 轴缩放 sz 倍：

$$\begin{aligned} \mathbf{rS} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} s_x r_x & s_y r_y & s_z r_z & 1 \end{bmatrix}, \end{aligned} \tag{5.7}$$

或者以分区简写形式：

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3\times3} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rS}_{3\times3} & 1 \end{bmatrix}.$$

以下是关于此类矩阵的一些观察结果：

The matrix below represents rotation about the y-axis by an angle θ. (Notice that this one is *transposed* relative to the other two—the positive and negative sine terms have been reflected across the diagonal.)

$$\text{rotate}_y(\mathbf{r},\theta) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.5}$$

The following matrix represents rotation about the z-axis by an angle γ:

$$\text{rotate}_z(\mathbf{r},\gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.6}$$

Here are a few observations about these matrices:

- The 1 within the upper 3×3 always appears on the axis we're rotating about, while the sine and cosine terms are off-axis.
- Positive rotations go from x to y (about z), from y to z (about x) and from z to x (about y). The z to x rotation "wraps around," which is why the rotation matrix about the y-axis is transposed relative to the other two. (Use the right-hand or left-hand rule to remember this.)
- The inverse of a pure rotation is just its transpose. This works because inverting a rotation is equivalent to rotating by the negative angle. You may recall that $\cos(-\theta) = \cos(\theta)$ while $\sin(-\theta) = -\sin(\theta)$, so negating the angle causes the two sine terms to effectively switch places, while the cosine terms stay put.

5.3.7.3 Scale

The following matrix scales the point $\mathbf{r}$ by a factor of s_x along the x-axis, s_y along the y-axis and s_z along the z-axis:

$$\begin{aligned} \mathbf{rS} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} s_x r_x & s_y r_y & s_z r_z & 1 \end{bmatrix}, \end{aligned} \tag{5.7}$$

or in partitioned shorthand:

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3\times3} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rS}_{3\times3} & 1 \end{bmatrix}.$$

Here are some observations about this kind of matrix:

• 要反转缩放矩阵，只需用 sx 、 sy 和 sz 的倒数替换（即 1/ sx 、 1/ sy 和 1/ sz ）。

• 当三个轴上的缩放因子相同（sx = sy = sz）时，我们称之为均匀缩放。球体在均匀缩放下仍为球体，而在非均匀缩放下则变为椭圆体。为了使边界球检查的数学运算简单快捷，许多游戏引擎施加了限制，即只能对可渲染几何体或碰撞图元应用均匀缩放。

• 当均匀缩放矩阵 S u 和旋转矩阵 R 连接时，乘法的顺序并不重要（即 S u R = RS u ）。这仅适用于均匀缩放！

4×3矩阵 5.3.8

4 × 4 仿射矩阵的最右列始终包含向量 [0 0 0 1] T 。因此，游戏程序员经常会省略第四列以节省内存。在游戏数学库中，你会经常遇到 4 × 3 仿射矩阵。

5.3.9 坐标空间

我们已经了解了如何使用 4 × 4 矩阵对点和方向向量进行变换。我们可以将这个想法扩展到刚体，因为刚体可以被认为是点的无限集合。对刚体应用变换就像对对象内的每个点应用相同的变换。例如，在计算机图形学中，一个对象通常由三角形网格表示，每个三角形网格有三个顶点，每个顶点由点表示。在这种情况下，可以通过依次对其所有顶点应用变换矩阵来变换对象。

图 5.17. 点 P 相对于不同坐标轴的位置矢量。

- To invert a scaling matrix, simply substitute s_x, s_y and s_z with their reciprocals (i.e., $1/s_x$, $1/s_y$ and $1/s_z$).
- When the scale factor along all three axes is the same ($s_x = s_y = s_z$), we call this *uniform scale*. Spheres remain spheres under uniform scale, whereas under nonuniform scale they become ellipsoids. To keep the mathematics of bounding sphere checks simple and fast, many game engines impose the restriction that only uniform scale may be applied to renderable geometry or collision primitives.
- When a uniform scale matrix $\mathbf{S}_u$ and a rotation matrix $\mathbf{R}$ are concatenated, the order of multiplication is unimportant (i.e., $\mathbf{S}_u\mathbf{R} = \mathbf{R}\mathbf{S}_u$). This only works for *uniform* scale!

5.3.8 4 × 3 Matrices

The rightmost column of an affine 4 × 4 matrix always contains the vector $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^{\mathrm{T}}$. As such, game programmers often omit the fourth column to save memory. You'll encounter 4 × 3 affine matrices frequently in game math libraries.

5.3.9 Coordinate Spaces

We've seen how to apply transformations to points and direction vectors using 4 × 4 matrices. We can extend this idea to rigid objects by realizing that such an object can be thought of as an infinite collection of points. Applying a transformation to a rigid object is like applying that same transformation to every point within the object. For example, in computer graphics an object is usually represented by a mesh of triangles, each of which has three vertices represented by points. In this case, the object can be transformed by applying a transformation matrix to all of its vertices in turn.

Figure 5.17. Position vectors for the point P relative to different coordinate axes.

我们上面说过，点是一个向量，其尾部固定在某个坐标系的原点。换句话说，一个点（位置向量）总是相对于一组坐标轴来表示。每当我们选择一组新的坐标轴时，表示点的三个数字的数值都会发生变化。在图 5.17 中，我们看到一个点 P 由两个不同的位置向量表示——向量 PA 表示 P 相对于“A”轴的位置，而向量 PB 表示同一点相对于另一组坐标轴“B”的位置。

在物理学中，一组坐标轴代表一个参考系，因此我们有时将一组坐标轴称为坐标系（或简称为 框架 ）。游戏行业人士也使用术语“坐标空间”（或简称为 空间 ）来指代一组坐标轴。在接下来的章节中，我们将介绍游戏和计算机图形学中最常用的几个坐标空间。

5.3.9.1 模型空间

在 Maya 或 3DStudioMAX 等工具中创建三角形网格时，三角形顶点的位置是相对于笛卡尔坐标系指定的，我们称之为模型空间（也称为对象空间或局部空间）。模型空间原点通常位于对象的中心位置，例如其质心，或者位于人形或动物角色双脚之间的地面上。

大多数游戏对象都具有固有的方向性。例如，飞机的机头、尾翼和机翼分别对应前、上和左/右方向。模型空间的轴通常与模型上的这些自然方向对齐，并赋予它们直观的名称来指示其方向性，如图 5.18 所示。

- 正面。这个名称指的是指向物体自然运动方向或朝向的轴。本书中，我们将使用符号 F 来表示

图 5.18. 飞机模型空间前轴、左轴和上轴基向量的一种可能选择。

We said above that a point is a vector whose tail is fixed to the origin of some coordinate system. This is another way of saying that a point (position vector) is always expressed *relative* to a set of coordinate axes. The triplet of numbers representing a point changes numerically whenever we select a new set of coordinate axes. In Figure 5.17, we see a point P represented by two different position vectors—the vector $\mathbf{P}_A$ gives the position of **P** relative to the “A” axes, while the vector $\mathbf{P}_B$ gives the position of that same point relative to a different set of axes “B.”

In physics, a set of coordinate axes represents a frame of reference, so we sometimes refer to a set of axes as a *coordinate frame* (or just a *frame*). People in the game industry also use the term *coordinate space* (or simply *space*) to refer to a set of coordinate axes. In the following sections, we'll look at a few of the most common coordinate spaces used in games and computer graphics.

5.3.9.1 Model Space

When a triangle mesh is created in a tool such as Maya or 3DStudioMAX, the positions of the triangles' vertices are specified relative to a Cartesian coordinate system, which we call *model space* (also known as *object space* or *local space*). The model-space origin is usually placed at a central location within the object, such as at its center of mass, or on the ground between the feet of a humanoid or animal character.

Most game objects have an inherent directionality. For example, an airplane has a nose, a tail fin and wings that correspond to the front, up and left/right directions. The model-space axes are usually aligned to these natural directions on the model, and they're given intuitive names to indicate their directionality as illustrated in Figure 5.18.

- *Front*. This name is given to the axis that points in the direction that the object naturally travels or faces. In this book, we'll use the symbol **F** to

Figure 5.18. One possible choice of the model-space front, left and up axis basis vectors for an airplane.

指的是沿前轴的单位基向量。

- 向上 。指向物体顶部的轴。沿此轴的单位基向量记为 U 。

- 左 或 右 。指向对象左侧或右侧的轴被称为“左”或“右”。具体名称取决于游戏引擎使用的是左手坐标系还是右手坐标系。沿该轴的单位基向量将根据情况分别表示为 L 或 R 。

（前、上、左）标签和（x、y、z）轴之间的映射完全是任意的。使用右手轴时，一种常见的选择是将标签“前”分配给正 z 轴，将标签“左”分配给正 x 轴，将标签“上”分配给正 y 轴（或用单位基向量表示，F = k、L = i 和 U = j）。但是，同样常见的是 + x 为前，+ z 为右（F = i、R = k、U = j）。我也使用过 z 轴垂直方向的引擎。唯一真正的要求是，在整个引擎中始终坚持一个约定。

举个例子，直观的轴名可以减少混淆，例如欧拉角（俯仰角、偏航角、滚转角），它们通常用于描述飞机的方向。由于俯仰角、偏航角和滚转角的方向是任意的，因此无法用 (i, j, k) 基向量来定义它们。然而，我们可以用 (L, U, F) 基向量来定义俯仰角、偏航角和滚转角，因为它们的方向是明确定义的。具体来说，

- 俯仰是围绕 L 或 R 旋转，
- 偏航是围绕 U 的旋转，并且
- 滚动是围绕 F 的旋转。

5.3.9.2 世界空间

世界空间是一个固定的坐标空间，游戏世界中所有物体的位置、方向和比例都在此空间中表达。这个坐标空间将所有单个物体连接在一起，形成一个完整的虚拟世界。

世界空间原点的位置是任意的，但它通常位于可玩游戏空间的中心附近，以尽量减少 (x, y, z) 坐标过大时可能出现的浮点精度下降。同样，x、y 和 z 轴的方向也是任意的，尽管我遇到的大多数引擎都使用 y 轴向上或 αz 轴向上的约定。y 轴向上的约定可能是大多数数学教科书中二维约定的延伸，其中 y 轴向上，x 轴向右。z 轴向上的约定也很常见。

refer to a unit basis vector along the front axis.

- *Up*. This name is given to the axis that points towards the top of the object. The unit basis vector along this axis will be denoted **U**.
- *Left or right*. The name “left” or “right” is given to the axis that points toward the left or right side of the object. Which name is chosen depends on whether your game engine uses left-handed or right-handed coordinates. The unit basis vector along this axis will be denoted **L** or **R**, as appropriate.

The mapping between the $(front, up, left)$ labels and the (x, y, z) axes is completely arbitrary. A common choice when working with right-handed axes is to assign the label *front* to the positive z-axis, the label *left* to the positive x-axis and the label *up* to the positive y-axis (or in terms of unit basis vectors, $\mathbf{F} = \mathbf{k}$, $\mathbf{L} = \mathbf{i}$ and $\mathbf{U} = \mathbf{j}$). However, it's equally common for $+x$ to be *front* and $+z$ to be right ($\mathbf{F} = \mathbf{i}$, $\mathbf{R} = \mathbf{k}$, $\mathbf{U} = \mathbf{j}$). I've also worked with engines in which the z-axis is oriented vertically. The only real requirement is that you stick to one convention consistently throughout your engine.

As an example of how intuitive axis names can reduce confusion, consider *Euler angles* $(pitch, yaw, roll)$, which are often used to describe an aircraft's orientation. It's not possible to define pitch, yaw, and roll angles in terms of the $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ basis vectors because their orientation is arbitrary. However, we *can* define pitch, yaw and roll in terms of the $(\mathbf{L}, \mathbf{U}, \mathbf{F})$ basis vectors, because their orientations are clearly defined. Specifically,

- *pitch* is rotation about **L** or **R**,
- *yaw* is rotation about **U**, and
- *roll* is rotation about **F**.

5.3.9.2 World Space

World space is a fixed coordinate space, in which the positions, orientations and scales of all objects in the game world are expressed. This coordinate space ties all the individual objects together into a cohesive virtual world.

The location of the world-space origin is arbitrary, but it is often placed near the center of the playable game space to minimize the reduction in floating-point precision that can occur when (x, y, z) coordinates grow very large. Likewise, the orientation of the x-, y- and z-axes is arbitrary, although most of the engines I've encountered use either a y-up or a z-up convention. The y-up convention was probably an extension of the two-dimensional convention found in most mathematics textbooks, where the y-axis is shown going up and the x-axis going to the right. The z-up convention is also common,

图 5.19。一架里尔喷气式飞机，其左翼尖在模型空间中的位置为 (5, 0, 0)。如果将喷气式飞机绕世界空间 y 轴旋转 90 度，并将其模型空间原点平移到世界空间中的 (− 25, 50, 8)，那么其左翼尖在世界空间坐标系中的位置最终将为 (− 25, 50, 3)。

因为它允许自上而下的正交视图使游戏世界看起来像传统的二维 xy 图。

举个例子，假设我们的飞机的左翼尖在模型空间中的位置是（5，0，0）。（在我们的游戏中，前矢量对应于模型空间中的正 z 轴，y 轴朝上，如图 5.18 所示。）现在想象一下，喷气式飞机面向世界空间中的正 x 轴，其模型空间原点位于某个任意位置，例如（-25，50，8）。因为飞机的 F 矢量（对应于模型空间中的 +z）面向世界空间中的 + x 轴，所以我们知道喷气式飞机已经绕世界 y 轴旋转了 90 度。因此，如果飞机位于世界空间原点，则其左翼尖在世界空间中的位置为（0，0，-5）。但由于飞机的原点已平移至 (− 25, 50, 8) ，因此飞机左翼尖在世界空间中的最终位置为 (− 25, 50, [8 − 5]) = (− 25, 50, 3) 。如图 5.19 所示。

当然，我们可以在我们友好的天空中部署不止一架里尔喷气式飞机。在这种情况下，所有飞机左翼尖在模型空间中的坐标都是 (5, 0, 0)。但在世界空间中，左翼尖会拥有各种各样有趣的坐标，具体取决于每架飞机的方向和平移。

5.3.9.3 视图空间

视图空间（也称为相机空间）是一个固定在相机上的坐标系。视图空间原点位于相机的焦点处。同样，任何轴方向方案都是可行的。然而，y 向上约定（z 轴沿相机朝向的方向（左手）增加）是典型的约定，因为它允许 z 坐标表示屏幕的深度。其他引擎和 API（例如 OpenGL）将视图空间定义为右手坐标系，在这种情况下，相机朝向负 z 轴，z 坐标表示负深度。

Figure 5.19. A Lear jet whose left wingtip is at (5, 0, 0) in model space. If the jet is rotated by 90 degrees about the world-space y-axis, and its model-space origin translated to $(-25, 50, 8)$ in world space, then its left wingtip would end up at $(-25, 50, 3)$ when expressed in world-space coordinates.

because it allows a top-down orthographic view of the game world to look like a traditional two-dimensional xy-plot.

As an example, let's say that our aircraft's left wingtip is at $(5, 0, 0)$ in model space. (In our game, front vectors correspond to the positive z-axis in model space with y up, as shown in Figure 5.18.) Now imagine that the jet is facing down the positive x-axis in world space, with its model-space origin at some arbitrary location, such as $(-25, 50, 8)$. Because the **F** vector of the airplane, which corresponds to $+z$ in model space, is facing down the $+x$-axis in world space, we know that the jet has been rotated by 90 degrees about the world y-axis. So, if the aircraft were sitting at the world-space origin, its left wingtip would be at $(0, 0, -5)$ in world space. But because the aircraft's origin has been translated to $(-25, 50, 8)$, the final position of the jet's left wingtip in world space is $(-25, 50, [8 - 5]) = (-25, 50, 3)$. This is illustrated in Figure 5.19.

We could of course populate our friendly skies with more than one Lear jet. In that case, all of their left wingtips would have coordinates of $(5, 0, 0)$ in model space. But in world space, the left wingtips would have all sorts of interesting coordinates, depending on the orientation and translation of each aircraft.

5.3.9.3 View Space

View space (also known as *camera space*) is a coordinate frame fixed to the camera. The view space origin is placed at the focal point of the camera. Again, any axis orientation scheme is possible. However, a y-up convention with z increasing in the direction the camera is facing (left-handed) is typical because it allows z coordinates to represent depths into the screen. Other engines and APIs, such as OpenGL, define view space to be right-handed, in which case the camera faces towards *negative* z, and z coordinates represent negative depths.

图 5.20 视图空间（也称为相机空间）的左手和右手示例。

图 5.20 说明了视图空间的两种可能的定义。

5.3.10 基础变更

在游戏和计算机图形学中，将对象的位置、方向和比例从一个坐标系转换到另一个坐标系通常非常有用。我们称此操作为 基变换 。

5.3.10.1 坐标空间层次

坐标系是相对的。也就是说，如果要量化三维空间中一组轴的位置、方向和比例，必须相对于另一组轴指定这些量（否则这些数字就没有意义）。这意味着坐标空间形成了一个层次结构——每个坐标空间都是其他坐标空间的子空间，而另一个坐标空间则充当其父空间。世界空间没有父空间；它位于坐标空间树的根部，所有其他坐标系最终都是相对于它的，要么是直接子空间，要么是更远的亲属。

5.3.10.2 构建基础变更矩阵

将任意子坐标系 C 中的点和方向变换到其父坐标系 P 的矩阵可以写成 MC → P（读作“C to P”）。下标表示该矩阵将点和方向从子空间变换到父空间。任何子空间位置向量 PC 都可以

Figure 5.20. Left- and right-handed examples of view space, also known as camera space.

Two possible definitions of view space are illustrated in Figure 5.20.

5.3.10 Change of Basis

In games and computer graphics, it is often quite useful to convert an object's position, orientation and scale from one coordinate system into another. We call this operation a *change of basis*.

5.3.10.1 Coordinate Space Hierarchies

Coordinate frames are relative. That is, if you want to quantify the position, orientation and scale of a set of axes in three-dimensional space, you must specify these quantities *relative to* some other set of axes (otherwise the numbers would have no meaning). This implies that coordinate spaces form a *hierarchy*—every coordinate space is a *child* of some other coordinate space, and the other space acts as its *parent*. World space has no parent; it is at the root of the coordinate-space tree, and all other coordinate systems are ultimately specified relative to it, either as direct children or more-distant relatives.

5.3.10.2 Building a Change of Basis Matrix

The matrix that transforms points and directions from any child coordinate system C to its parent coordinate system P can be written $\mathbf{M}_{C\to P}$ (pronounced "C to P"). The subscript indicates that this matrix transforms points and directions from child space to parent space. Any child-space position vector $\mathbf{P}_C$ can

转换为父空间位置向量 PP 如下：

$$\mathbf{P}_P = \mathbf{P}_C \mathbf{M}_{C\to P};$$

$$\mathbf{M}_{C\to P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 1 \end{bmatrix}$$

$$= \begin{bmatrix} i_{Cx} & i_{Cy} & i_{Cz} & 0 \\ j_{Cx} & j_{Cy} & j_{Cz} & 0 \\ k_{Cx} & k_{Cy} & k_{Cz} & 0 \\ t_{Cx} & t_{Cy} & t_{Cz} & 1 \end{bmatrix}.$$

在这个等式中，

- iC 是沿子空间 x 轴的单位基向量，以父空间坐标表示；
- jC 是父空间中沿子空间 y 轴的单位基向量；
- kC 是父空间中沿子空间z轴的单位基向量；
- t C 是子坐标系相对于父空间的平移。

这个结果应该不会太令人惊讶。tC 向量只是子空间轴相对于父空间的平移，所以如果矩阵的其余部分是单位向量，那么子空间中的点 (0, 0, 0) 就会变成父空间中的 tC，正如我们所期望的那样。iC 、jC 和 kC 单位向量构成了矩阵的上半部分 3 × 3，这是一个纯旋转矩阵，因为这些向量都是单位长度。我们可以通过一个简单的例子更清楚地看到这一点，例如子空间绕 z 轴旋转角度 γ ，而没有平移的情况。回想一下公式 (5.6)，这种旋转的矩阵由下式给出

$$\text{旋转 z (r , γ)} = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

但在图 5.21 中，我们可以看到，向量 iC 和 jC 在父空间中的坐标分别为 iC = [cos γ sin γ 0] 和 jC = [− sin γ cos γ 0]。当我们将这些向量代入 MC → P 公式中，其中 kC = [0 0 1] 时，它与公式 (5.6) 中的矩阵 rotate z (r , γ) 完全匹配。

be transformed into a parent-space position vector $\mathbf{P}_P$ as follows:

$$\mathbf{P}_P = \mathbf{P}_C \mathbf{M}_{C\to P};$$

$$\mathbf{M}_{C\to P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 1 \end{bmatrix}$$

$$= \begin{bmatrix} i_{Cx} & i_{Cy} & i_{Cz} & 0 \\ j_{Cx} & j_{Cy} & j_{Cz} & 0 \\ k_{Cx} & k_{Cy} & k_{Cz} & 0 \\ t_{Cx} & t_{Cy} & t_{Cz} & 1 \end{bmatrix}.$$

In this equation,

- $\mathbf{i}_C$ is the unit basis vector along the child space x-axis, expressed in parent-space coordinates;
- $\mathbf{j}_C$ is the unit basis vector along the child space y-axis, in parent space;
- $\mathbf{k}_C$ is the unit basis vector along the child space z-axis, in parent space; and
- $\mathbf{t}_C$ is the translation of the child coordinate system relative to parent space.

This result should not be too surprising. The $\mathbf{t}_C$ vector is just the translation of the child-space axes relative to parent space, so if the rest of the matrix were identity, the point (0, 0, 0) in child space would become $\mathbf{t}_C$ in parent space, just as we'd expect. The $\mathbf{i}_C$, $\mathbf{j}_C$ and $\mathbf{k}_C$ unit vectors form the upper 3×3 of the matrix, which is a pure rotation matrix because these vectors are of unit length. We can see this more clearly by considering a simple example, such as a situation in which child space is rotated by an angle γ about the z-axis, with no translation. Recall from Equation (5.6) that the matrix for such a rotation is given by

$$\text{rotate}_z(\mathbf{r}, \gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

But in Figure 5.21, we can see that the coordinates of the $\mathbf{i}_C$ and $\mathbf{j}_C$ vectors, expressed in parent space, are $\mathbf{i}_C = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \end{bmatrix}$ and $\mathbf{j}_C = \begin{bmatrix} -\sin\gamma & \cos\gamma & 0 \end{bmatrix}$. When we plug these vectors into our formula for $\mathbf{M}_{C\to P}$, with $\mathbf{k}_C = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$, it exactly matches the matrix $\text{rotate}_z(\mathbf{r}, \gamma)$ from Equation (5.6).

缩放子轴

子坐标系的缩放只需适当缩放单位基向量即可。例如，如果子空间按 2 倍缩放，则基向量 iC 、 jC 和 kC 的长度将为 2，而不是单位长度。

5.3.10.3 从矩阵中提取单位基向量

事实上，我们可以用平移和三个笛卡尔基向量构建一个基变换矩阵，这给了我们另一个强大的工具：给定任何仿射 4×4 变换矩阵，我们可以朝另一个方向前进，通过简单地隔离矩阵的适当行（或列，如果你的数学库使用列向量）从中提取子空间基向量 iC、jC 和 kC。

这非常有用。假设我们给定一个车辆的模型到世界变换，它是一个 4 × 4 的仿射矩阵（一种非常常见的表示）。这实际上只是基矩阵的变换，将模型空间中的点转换为世界空间中的对应点。进一步假设在我们的游戏中，z 轴的正方向始终指向物体朝向。因此，为了找到一个表示车辆朝向的单位向量，我们可以直接从模型到世界矩阵中提取 kC（通过获取其第三行）。

该向量已经标准化并准备就绪。

5.3.10.4 坐标系与向量的转换

我们已经说过，矩阵 MC → P 将点和方向从子空间变换到父空间。回想一下，MC → P 的第四行包含 tC，即子坐标轴相对于世界空间坐标轴的平移。因此，另一种可视化矩阵 MC → P 的方法是想象它取父坐标轴并将其变换到子坐标轴上。这就是

图 5.21. 当子轴相对于父轴旋转角度 γ 时，基的变化。

Scaling the Child Axes

Scaling of the child coordinate system is accomplished by simply scaling the unit basis vectors appropriately. For example, if child space is scaled up by a factor of two, then the basis vectors $\mathbf{i}_C$, $\mathbf{j}_C$ and $\mathbf{k}_C$ will be of length 2 instead of unit length.

5.3.10.3 Extracting Unit Basis Vectors from a Matrix

The fact that we can build a change of basis matrix out of a translation and three Cartesian basis vectors gives us another powerful tool: Given *any* affine 4×4 transformation matrix, we can go in the other direction and extract the child-space basis vectors $\mathbf{i}_C$, $\mathbf{j}_C$ and $\mathbf{k}_C$ from it by simply isolating the appropriate rows of the matrix (or columns if your math library uses column vectors).

This can be incredibly useful. Let's say we are given a vehicle's model-to-world transform as an affine 4×4 matrix (a very common representation). This is really just a change of basis matrix, transforming points in model space into their equivalents in world space. Let's further assume that in our game, the positive z-axis always points in the direction that an object is facing. So, to find a unit vector representing the vehicle's facing direction, we can simply extract $\mathbf{k}_C$ directly from the model-to-world matrix (by grabbing its third row). This vector will already be normalized and ready to go.

5.3.10.4 Transforming Coordinate Systems versus Vectors

We've said that the matrix $\mathbf{M}_{C\to P}$ transforms points and directions from child space into parent space. Recall that the fourth row of $\mathbf{M}_{C\to P}$ contains $\mathbf{t}_C$, the translation of the child coordinate axes relative to the world-space axes. Therefore, another way to visualize the matrix $\mathbf{M}_{C\to P}$ is to imagine it taking the parent *coordinate axes* and transforming them *into* the child axes. This is the

Figure 5.21. Change of basis when child axes are rotated by an angle γ relative to parent.

图 5.22 两种解释变换矩阵的方法。左侧表示点沿一组固定的轴移动。右侧表示轴沿相反方向移动，而点保持不变。

与点和方向向量的情况相反。换句话说，如果矩阵将向量从子空间变换到父空间，那么它也会将坐标轴从父空间变换到子空间。仔细想想，这很有道理——在坐标轴固定的情况下，将点向右移动 20 个单位，与在坐标轴固定的情况下，将坐标轴向左移动 20 个单位是一样的。图 5.22 说明了这个概念。

当然，这只是另一个容易混淆的地方。如果你用坐标轴来思考，那么变换的方向就是一个方向；但如果你用点和向量来思考，变换的方向就是另一个方向！就像生活中许多令人困惑的事情一样，你最好的选择可能是选择一种“规范”的思维方式并坚持下去。例如，在本书中，我们选择了以下约定：

- 变换适用于矢量（不是坐标轴）。
- 向量写为行（而不是列）。

综上所述，这两个约定使我们能够从左到右读取矩阵乘法序列，并使其有意义（例如，在表达式 r D = r AMA → BMB → CMC → D 中，B 和 C 实际上“抵消”，只剩下 r D = r AMA → D ）。显然，如果您开始考虑移动坐标轴而不是点和向量，那么您要么必须从右到左读取变换，要么将这两个约定中的一个反过来。选择哪种约定并不重要，只要您觉得它们容易记住和使用即可。

话虽如此，需要注意的是，有些问题更容易用向量变换来思考，而有些问题则更容易想象坐标轴的移动。一旦你熟练掌握了三维向量和矩阵的数学思维，你就会发现，根据手头的问题，在不同的传统方法之间来回切换非常容易。

Figure 5.22. Two ways to interpret a transformation matrix. On the left, the point moves against a fixed set of axes. On the right, the axes move in the opposite direction while the point remains fixed.

reverse of what happens to points and direction vectors. In other words, if a matrix transforms *vectors* from child space to parent space, then it also transforms *coordinate axes* from parent space to child space. This makes sense when you think about it—moving a point 20 units to the right with the coordinate axes fixed is the same as moving the coordinate axes 20 units to the left with the point fixed. This concept is illustrated in Figure 5.22.

Of course, this is just another point of potential confusion. If you're thinking in terms of coordinate axes, then transformations go in one direction, but if you're thinking in terms of points and vectors, they go in the other direction! As with many confusing things in life, your best bet is probably to choose a single "canonical" way of thinking about things and stick with it. For example, in this book we've chosen the following conventions:

- Transformations apply to vectors (not coordinate axes).
- Vectors are written as rows (not columns).

Taken together, these two conventions allow us to read sequences of matrix multiplications from left to right and have them make sense (e.g., in the expression $\mathbf{r}_D = \mathbf{r}_A\mathbf{M}_{A\to B}\mathbf{M}_{B\to C}\mathbf{M}_{C\to D}$, the B's and C's in effect "cancel out," leaving only $\mathbf{r}_D = \mathbf{r}_A\mathbf{M}_{A\to D}$). Obviously if you start thinking about the coordinate axes moving around rather than the points and vectors, you either have to read the transforms from right to left, or flip one of these two conventions around. It doesn't really matter what conventions you choose as long as *you* find them easy to remember and work with.

That said, it's important to note that certain problems are easier to think about in terms of vectors being transformed, while others are easier to work with when you imagine the coordinate axes moving around. Once you get good at thinking about 3D vector and matrix math, you'll find it pretty easy to flip back and forth between conventions as needed to suit the problem at hand.

5.3.11 变换法向量

法向量是一种特殊的向量，因为除了（通常！）具有单位长度之外，它还具有额外的要求，即它必须始终垂直于与其关联的表面或平面。在变换法向量时必须特别小心，以确保其长度和垂直度属性均得到保持。

一般来说，如果一个点或（非法向量）可以通过 3 × 3 矩阵 MA → B 从空间 A 旋转到空间 B，那么法向量 n 也可以通过该矩阵的逆转置（M − 1 A → B）T 从空间 A 变换到空间 B。我们不会在这里证明或推导这个结果（参见 [32，第 3.5 节] 的详细推导）。但是，我们会观察到，如果矩阵 MA → B 只包含均匀尺度而没有剪切，那么空间 B 中所有表面和向量之间的角度将与空间 A 中的相同。在这种情况下，矩阵 MA → B 实际上对任何向量（无论是法向量还是非法向量）都适用。但是，如果 MA → B 包含非均匀尺度或剪切（即非正交），则从空间 A 移动到空间 B 时，表面和矢量之间的角度不会保持不变。在空间 A 中垂直于表面的矢量不一定在空间 B 中垂直于该表面。逆转置运算可以解释这种扭曲，即使变换涉及非均匀尺度或剪切，也能使法向量恢复与其表面的垂直。另一种看待这个问题的方式是，需要逆转置，因为表面法线实际上是一个伪矢量而不是常规矢量（参见第 5.2.4.9 节）。

5.3.12 在内存中存储矩阵

在 C 和 C++ 语言中，二维数组通常用于存储矩阵。回想一下，在 C/C++ 二维数组语法中，第一个下标代表行，第二个下标代表列，并且列索引在内存中顺序移动时变化最快。

```
float m[4][4]; // [row][col], col varies fastest

// "flatten" the array to demonstrate ordering
float* pm = &m[0][0];
ASSERT( &pm[0] == &m[0][0] );
ASSERT( &pm[1] == &m[0][1] );
ASSERT( &pm[2] == &m[0][2] );
// etc.
```

将矩阵存储在二维 C/C++ 数组中时，我们有两种选择。我们可以

5.3.11 Transforming Normal Vectors

A *normal vector* is a special kind of vector, because in addition to (usually!) being of unit length, it carries with it the additional requirement that it should always remain *perpendicular* to whatever surface or plane it is associated with. Special care must be taken when transforming a normal vector to ensure that both its length and perpendicularity properties are maintained.

In general, if a point or (non-normal) vector can be rotated from space A to space B via the 3×3 matrix $\mathbf{M}_{A\to B}$, then a normal vector $\mathbf{n}$ will be transformed from space A to space B via the *inverse transpose* of that matrix, $(\mathbf{M}_{A\to B}^{-1})^{\mathrm{T}}$. We will not prove or derive this result here (see [32, Section 3.5] for an excellent derivation). However, we will observe that if the matrix $\mathbf{M}_{A\to B}$ contains only uniform scale and no shear, then the angles between all surfaces and vectors in space B will be the same as they were in space A. In this case, the matrix $\mathbf{M}_{A\to B}$ will actually work just fine for any vector, normal or non-normal. However, if $\mathbf{M}_{A\to B}$ contains nonuniform scale or shear (i.e., is *non-orthogonal*), then the angles between surfaces and vectors are *not* preserved when moving from space A to space B. A vector that was normal to a surface in space A will not necessarily be perpendicular to that surface in space B. The inverse transpose operation accounts for this distortion, bringing normal vectors back into perpendicularity with their surfaces even when the transformation involves nonuniform scale or shear. Another way of looking at this is that the inverse transpose is required because a surface normal is really a *pseudovector* rather than a regular vector (see Section 5.2.4.9).

5.3.12 Storing Matrices in Memory

In the C and C++ languages, a two-dimensional array is often used to store a matrix. Recall that in C/C++ two-dimensional array syntax, the first subscript is the row and the second is the column, and the column index varies fastest as you move through memory sequentially.

```
float m[4][4]; // [row][col], col varies fastest

// "flatten" the array to demonstrate ordering
float* pm = &m[0][0];
ASSERT( &pm[0] == &m[0][0] );
ASSERT( &pm[1] == &m[0][1] );
ASSERT( &pm[2] == &m[0][2] );
// etc.
```

We have two choices when storing a matrix in a two-dimensional C/C++ array. We can either

1. 将向量（iC、jC、kC、tC）连续存储在内存中（即每行包含一个向量），或者

2. 将跨步向量存储在内存中（即每列包含一个向量）。

方法 (1) 的好处在于，我们只需对矩阵进行索引，并将找到的四个连续值解释为一个四元素向量，即可处理四个向量中的任意一个。这种布局的另一个好处是它与行向量矩阵方程完全匹配（这也是我在本书中选择行向量表示法的另一个原因）。方法 (2) 在使用支持矢量 (SIMD) 的微处理器进行快速矩阵向量乘法时有时是必要的，我们将在本章后面看到。在我个人遇到的大多数游戏引擎中，矩阵都使用方法 (1) 存储，向量存储在二维 C/C++ 数组的行中。如下所示：

```
float M[4][4];

M[0][0]=ix;  M[0][1]=iy;  M[0][2]=iz;  M[0][3]=0.0f;
M[1][0]=jx;  M[1][1]=jy;  M[1][2]=jz;  M[1][3]=0.0f;
M[2][0]=kx;  M[2][1]=ky;  M[2][2]=kz;  M[2][3]=0.0f;
M[3][0]=tx;  M[3][1]=ty;  M[3][2]=tz;  M[3][3]=1.0f;
```

在调试器中查看时，矩阵如下所示：

```
M[][]
    [0]
        [0] ix
        [1] iy
        [2] iz
        [3] 0.0000
    [1]
        [0] jx
        [1] jy
        [2] jz
        [3] 0.0000
    [2]
        [0] kx
        [1] ky
        [2] kz
        [3] 0.0000
    [3]
        [0] tx
        [1] ty
        [2] tz
        [3] 1.0000
```


1. store the vectors ($\mathbf{i}_C$, $\mathbf{j}_C$, $\mathbf{k}_C$, $\mathbf{t}_C$) *contiguously* in memory (i.e., each row contains a single vector), or

2. store the vectors *strided* in memory (i.e., each column contains one vector).

The benefit of approach (1) is that we can address any one of the four vectors by simply indexing into the matrix and interpreting the four contiguous values we find there as a 4-element vector. This layout also has the benefit of matching up exactly with *row vector* matrix equations (which is another reason why I've selected row vector notation for this book). Approach (2) is sometimes necessary when doing fast matrix-vector multiplies using a vector-enabled (SIMD) microprocessor, as we'll see later in this chapter. In most game engines I've personally encountered, matrices are stored using approach (1), with the *vectors* in the *rows* of the two-dimensional C/C++ array. This is shown below:

```
float M[4][4];

M[0][0]=ix;  M[0][1]=iy;  M[0][2]=iz;  M[0][3]=0.0f;
M[1][0]=jx;  M[1][1]=jy;  M[1][2]=jz;  M[1][3]=0.0f;
M[2][0]=kx;  M[2][1]=ky;  M[2][2]=kz;  M[2][3]=0.0f;
M[3][0]=tx;  M[3][1]=ty;  M[3][2]=tz;  M[3][3]=1.0f;
```

The matrix looks like this when viewed in a debugger:

```
M[][]
    [0]
        [0] ix
        [1] iy
        [2] iz
        [3] 0.0000
    [1]
        [0] jx
        [1] jy
        [2] jz
        [3] 0.0000
    [2]
        [0] kx
        [1] ky
        [2] kz
        [3] 0.0000
    [3]
        [0] tx
        [1] ty
        [2] tz
        [3] 1.0000
```

确定引擎使用哪种布局的一个简单方法是找到一个构建 4 × 4 平移矩阵的函数。（每个优秀的 3D 数学库都提供这样的函数。）然后，您可以检查源代码以查看 t 向量元素的存储位置。如果您无法访问数学库的源代码（这在游戏行业中非常罕见），您可以随时使用易于识别的平移（例如（4, 3, 2））来调用该函数，然后检查生成的矩阵。如果第 3 行包含值 4.0f 、 3.0f 、 2.0f 、 1.0f ，则向量位于行中，否则向量位于列中。

5.4 四元数

我们已经知道，3×3 矩阵可以用来表示三维空间中的任意旋转。然而，矩阵并不总是旋转的理想表示，原因如下：

1. 我们需要九个浮点值来表示一次旋转，考虑到我们只有三个自由度——俯仰、偏航和滚转，这似乎有些过多。

2. 旋转向量需要进行向量矩阵乘法，这涉及三次点积，总共需要九次乘法和六次加法。如果可能的话，我们希望找到一种计算成本更低的旋转表示方法。

3. 在游戏和计算机图形学中，找到两个已知旋转角度之间一定百分比的旋转角度通常非常重要。例如，如果我们要在几秒钟内让摄像机从起始方向 A 平滑地移动到最终方向 B，就需要能够在动画过程中找到 A 和 B 之间的大量中间旋转角度。当 A 和 B 方向以矩阵形式表示时，这一点就变得非常困难。

值得庆幸的是，有一种旋转表示方法可以克服这三个问题。它就是一个叫做四元数的数学对象。四元数看起来很像四维向量，但其行为却截然不同。我们通常使用非斜体、非粗体字体来表示四元数，如下所示：

$\mathsf{q} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}$.

四元数由威廉 · 罗文 · 汉密尔顿爵士于 1843 年提出，是对复数的扩展。（具体来说，四元数可以解释为一个四维复数，只有一个实轴

One easy way to determine which layout your engine uses is to find a function that builds a 4 × 4 translation matrix. (Every good 3D math library provides such a function.) You can then inspect the source code to see where the elements of the **t** vector are being stored. If you don't have access to the source code of your math library (which is pretty rare in the game industry), you can always call the function with an easy-to-recognize translation like (4, 3, 2), and then inspect the resulting matrix. If row 3 contains the values `4.0f`, `3.0f`, `2.0f`, `1.0f`, then the vectors are in the rows, otherwise the vectors are in the columns.

5.4 Quaternions

We've seen that a 3 × 3 matrix can be used to represent an arbitrary rotation in three dimensions. However, a matrix is not always an ideal representation of a rotation, for a number of reasons:

1. We need nine floating-point values to represent a rotation, which seems excessive considering that we only have three degrees of freedom—pitch, yaw and roll.

2. Rotating a vector requires a vector-matrix multiplication, which involves three dot products, or a total of nine multiplications and six additions. We would like to find a rotational representation that is less expensive to calculate, if possible.

3. In games and computer graphics, it's often important to be able to find rotations that are some percentage of the way between two known rotations. For example, if we are to smoothly animate a camera from some starting orientation A to some final orientation B over the course of a few seconds, we need to be able to find lots of intermediate rotations between A and B over the course of the animation. It turns out to be difficult to do this when the A and B orientations are expressed as matrices.

Thankfully, there is a rotational representation that overcomes these three problems. It is a mathematical object known as a *quaternion*. A quaternion looks a lot like a four-dimensional vector, but it *behaves* quite differently. We usually write quaternions using non-italic, non-boldface type, like this: $\mathsf{q} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}$.

Quaternions were developed by Sir William Rowan Hamilton in 1843 as an extension to the complex numbers. (Specifically, a quaternion may be interpreted as a four-dimensional complex number, with a single real axis

以及用虚数 i 、 j 和 k 表示的三个虚轴。因此，四元数可以写成“复数形式”如下： q = iq x + jq y + kq z + qw 。）四元数最初是用来解决力学领域的问题的。从技术上讲，四元数遵循一组称为实数上的四维范数除法代数的规则。值得庆幸的是，我们不需要了解这些相当深奥的代数规则的细节。对于我们的目的而言，知道单位长度四元数（即所有遵循约束 q 2 x + q 2 y + q 2 z + q 2 w = 1 的四元数）表示三维旋转就足够了。

网上有很多关于四元数的优秀论文、网页和演示文稿可供进一步阅读。以下是我最喜欢的一个：http://graphics.ucsd.edu/courses/cse169_w05/CSE169_04.ppt 。

5.4.1 单位四元数作为三维旋转

单位四元数可以看作是一个三维向量加上第四个标量坐标。向量部分 q V 是单位旋转轴，用旋转半角的正弦值进行缩放。标量部分 q S 是半角的余弦值。因此，单位四元数 q 可以写成如下形式：

$$\begin{aligned} \mathsf{q} &= \begin{bmatrix} \mathbf{q}_V & q_S \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{a} \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \end{aligned}$$

其中 a 是沿旋转轴的单位向量，θ 是旋转角度。旋转方向遵循右手定则，因此，如果拇指指向 a 方向，则正向旋转方向为弯曲手指的方向。

当然，我们也可以将 q 写成一个简单的四元素向量：

q = [qxqyqzqw] ，其中 qx = q V x = αx sin θ 2 ， qy = q V y = αy sin θ 2 ，

$$\begin{aligned} q_z &= q_{V_z} = a_z \sin\frac{\theta}{2}, \\ q_w &= q_S = \cos\frac{\theta}{2}. \end{aligned}$$

单位四元数非常类似于旋转的轴+角表示（即形式为 [α θ] 的四元素向量）。然而，四元数在数学上比轴+角表示更方便，正如我们将在下文中看到的那样。

and three imaginary axes represented by the imaginary numbers i, j and k. As such, a quaternion can be written in “complex form” as follows: $\mathsf{q} = iq_x + jq_y + kq_z + q_w$.) Quaternions were first used to solve problems in the area of mechanics. Technically speaking, a quaternion obeys a set of rules known as a four-dimensional *normed division algebra* over the real numbers. Thankfully, we won’t need to understand the details of these rather esoteric algebraic rules. For our purposes, it will suffice to know that the *unit-length* quaternions (i.e., all quaternions obeying the constraint $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$) represent three-dimensional rotations.

There are a lot of great papers, web pages and presentations on quaternions available on the web for further reading. Here’s one of my favorites: http://graphics.ucsd.edu/courses/cse169_w05/CSE169_04.ppt.

5.4.1 Unit Quaternions as 3D Rotations

A unit quaternion can be visualized as a three-dimensional vector plus a fourth scalar coordinate. The vector part $\mathbf{q}_V$ is the unit axis of rotation, scaled by the sine of the half-angle of the rotation. The scalar part q_S is the cosine of the half-angle. So the unit quaternion q can be written as follows:

$$\begin{aligned} \mathsf{q} &= \begin{bmatrix} \mathbf{q}_V & q_S \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{a} \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \end{aligned}$$

where $\mathbf{a}$ is a unit vector along the axis of rotation, and θ is the angle of rotation. The direction of the rotation follows the *right-hand rule*, so if your thumb points in the direction of $\mathbf{a}$, positive rotations will be in the direction of your curved fingers.

Of course, we can also write q as a simple four-element vector:

$$\mathsf{q} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}, \text{where}$$

$$\begin{aligned} q_x &= q_{V_x} = a_x \sin\frac{\theta}{2}, \\ q_y &= q_{V_y} = a_y \sin\frac{\theta}{2}, \\ q_z &= q_{V_z} = a_z \sin\frac{\theta}{2}, \\ q_w &= q_S = \cos\frac{\theta}{2}. \end{aligned}$$

A unit quaternion is very much like an axis+angle representation of a rotation (i.e., a four-element vector of the form $\begin{bmatrix} \mathbf{a} & \theta \end{bmatrix}$). However, quaternions are more convenient mathematically than their axis+angle counterparts, as we shall see below.

5.4.2 四元数运算

四元数支持一些向量代数中常见的运算，例如幅值和向量加法。然而，我们必须记住，两个单位四元数的和并不代表三维旋转，因为这样的四元数长度不是单位的。因此，在游戏引擎中你不会看到任何四元数和，除非它们以某种方式缩放以保持单位长度的要求。

5.4.2.1 四元数乘法

我们将对四元数执行的最重要的运算之一是乘法。给定两个四元数 p 和 q ，分别表示两个旋转 P 和 Q ，乘积 pq 表示复合旋转（即旋转 Q 后跟旋转 P ）。实际上，四元数乘法有很多种，但我们将仅限于讨论与三维旋转结合使用的那一种，即 Grassman 乘积。

使用此定义，乘积 pq 定义如下：

$$\mathsf{pq} = \left[(p_S \mathbf{q}_V + q_S \mathbf{p}_V + \mathbf{p}_V \times \mathbf{q}_V) \quad (p_S q_S - \mathbf{p}_V \cdot \mathbf{q}_V) \right].$$

请注意 Grassman 积是如何根据矢量部分（最终为结果四元数的 x 、 y 和 z 分量）和标量部分（最终为 w 分量）来定义的。

5.4.2.2 共轭与逆

四元数 q 的逆表示为 q − 1，其定义为：当与原四元数相乘时，其标量为 1（即 qq − 1 = 0 i + 0 j + 0 k + 1）。四元数 [0 0 0 1] 表示零旋转（这是合理的，因为前三个分量的 sin (0) = 0，最后一个分量的 cos (0) = 1）。

为了计算四元数的逆，我们必须首先定义一个称为共轭的量。它通常表示为 q □ ，定义如下：

$$\mathsf{q}^* = \left[-\mathbf{q}_V \quad q_S \right].$$

换句话说，我们否定矢量部分但保持标量部分不变。
根据四元数共轭的定义，逆四元数 q − 1 定义如下：

$$\mathsf{q}^{-1} = \frac{\mathsf{q}^*}{|\mathsf{q}|^2}.$$

5.4.2 Quaternion Operations

Quaternions support some of the familiar operations from vector algebra, such as magnitude and vector addition. However, we must remember that the sum of two unit quaternions does not represent a 3D rotation, because such a quaternion would not be of unit length. As a result, you won't see any quaternion sums in a game engine, unless they are scaled in some way to preserve the unit length requirement.

5.4.2.1 Quaternion Multiplication

One of the most important operations we will perform on quaternions is that of multiplication. Given two quaternions p and q representing two rotations **P** and **Q**, respectively, the product pq represents the composite rotation (i.e., rotation **Q** followed by rotation **P**). There are actually quite a few different kinds of quaternion multiplication, but we'll restrict this discussion to the variety used in conjunction with 3D rotations, namely the Grassman product. Using this definition, the product pq is defined as follows:

$$\mathsf{pq} = \left[(p_S \mathbf{q}_V + q_S \mathbf{p}_V + \mathbf{p}_V \times \mathbf{q}_V) \quad (p_S q_S - \mathbf{p}_V \cdot \mathbf{q}_V) \right].$$

Notice how the Grassman product is defined in terms of a vector part, which ends up in the x, y and z components of the resultant quaternion, and a scalar part, which ends up in the w component.

5.4.2.2 Conjugate and Inverse

The *inverse* of a quaternion q is denoted q^{-1} and is defined as a quaternion that, when multiplied by the original, yields the scalar 1 (i.e., $\mathsf{q}\mathsf{q}^{-1} = 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k} + 1$). The quaternion $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ represents a zero rotation (which makes sense since $\sin(0) = 0$ for the first three components, and $\cos(0) = 1$ for the last component).

In order to calculate the inverse of a quaternion, we must first define a quantity known as the *conjugate*. This is usually denoted q^* and it is defined as follows:

$$\mathsf{q}^* = \left[-\mathbf{q}_V \quad q_S \right].$$

In other words, we negate the vector part but leave the scalar part unchanged.

Given this definition of the quaternion conjugate, the inverse quaternion q^{-1} is defined as follows:

$$\mathsf{q}^{-1} = \frac{\mathsf{q}^*}{|\mathsf{q}|^2}.$$

我们的四元数总是单位长度（即 | q | = 1 ），因为它们表示三维旋转。因此，就我们的目的而言，逆元数和共轭元数是相同的：

$$\mathsf{q}^{-1} = \mathsf{q}^* = \begin{bmatrix} -\mathbf{q}_V & q_S \end{bmatrix} \quad \text{什么时候} \quad |\mathsf{q}| = 1.$$

这个事实非常有用，因为它意味着只要我们事先知道四元数是标准化的，我们就可以避免在求四元数的逆时进行（相对昂贵的）平方除法。这也意味着求四元数的逆通常比求 3 × 3 矩阵的逆快得多——在某些情况下，你可以在优化引擎时利用这一点。

乘积的共轭和逆

四元数乘积的共轭（ pq ）等于各个四元数的共轭的逆乘积：

$$(\mathsf{pq})^* = \mathsf{q}^*\mathsf{p}^*.$$

同样，四元数乘积的逆等于各个四元数的逆的逆乘积：

$$(\mathsf{pq})^{-1} = \mathsf{q}^{-1}\mathsf{p}^{-1}. \tag{5.8}$$

这类似于转置或反转矩阵乘积时发生的逆转。

5.4.3 用四元数旋转向量

如何将四元数旋转应用于向量？第一步是将向量重写为四元数形式。向量是单位基向量 i 、 j 和 k 的和。四元数是 i 、 j 和 k 的和，但也包含第四个标量项。因此，向量可以写成四元数，其标量项 qS 等于零，这是有道理的。给定向量 v ，我们可以写出相应的四元数 v = [v 0] = [vxvyvz 0] 。

为了将向量 v 旋转四元数 q，我们将向量（写成四元数形式 v ）预乘以 q ，然后将其后乘以逆四元数 q − 1 。因此，旋转后的向量 v ′ 可以通过以下公式求得：

$$\mathsf{v}' = \text{rotate}(\mathsf{q}, \mathbf{v}) = \mathsf{qvq}^{-1}.$$

这相当于使用四元数共轭，因为我们的四元数始终是单位长度：

$$\mathsf{v}' = \text{旋转}(\mathsf{q}, \mathsf{v}) = \mathsf{qvq}^{\square}。 \tag{5.9}$$

Our quaternions are always of unit length (i.e., $|\mathsf{q}| = 1$), because they represent 3D rotations. So, for our purposes, the inverse and the conjugate are identical:

$$\mathsf{q}^{-1} = \mathsf{q}^* = \begin{bmatrix} -\mathbf{q}_V & q_S \end{bmatrix} \quad \text{when} \quad |\mathsf{q}| = 1.$$

This fact is incredibly useful, because it means we can always avoid doing the (relatively expensive) division by the squared magnitude when inverting a quaternion, as long as we know a priori that the quaternion is normalized. This also means that inverting a quaternion is generally much faster than inverting a 3×3 matrix—a fact that you may be able to leverage in some situations when optimizing your engine.

Conjugate and Inverse of a Product

The conjugate of a quaternion product (pq) is equal to the reverse product of the conjugates of the individual quaternions:

$$(\mathsf{pq})^* = \mathsf{q}^*\mathsf{p}^*.$$

Likewise, the inverse of a quaternion product is equal to the reverse product of the inverses of the individual quaternions:

$$(\mathsf{pq})^{-1} = \mathsf{q}^{-1}\mathsf{p}^{-1}. \tag{5.8}$$

This is analogous to the reversal that occurs when transposing or inverting matrix products.

5.4.3 Rotating Vectors with Quaternions

How can we apply a quaternion rotation to a vector? The first step is to rewrite the vector in *quaternion form*. A vector is a sum involving the unit basis vectors **i**, **j** and **k**. A quaternion is a sum involving **i**, **j** and **k**, but with a fourth scalar term as well. So it makes sense that a vector can be written as a quaternion with its scalar term q_S equal to zero. Given the vector **v**, we can write a corresponding quaternion $\mathsf{v} = \begin{bmatrix} \mathbf{v} & 0 \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z & 0 \end{bmatrix}$.

In order to rotate a vector **v** by a quaternion q, we premultiply the vector (written in its quaternion form v) by q and then post-multiply it by the *inverse* quaternion q^{-1}. Therefore, the rotated vector $\mathbf{v}'$ can be found as follows:

$$\mathsf{v}' = \text{rotate}(\mathsf{q}, \mathbf{v}) = \mathsf{qvq}^{-1}.$$

This is equivalent to using the quaternion conjugate, because our quaternions are always unit length:

$$\mathsf{v}' = \text{rotate}(\mathsf{q}, \mathbf{v}) = \mathsf{qvq}^*. \tag{5.9}$$

旋转向量 v′ 可以通过简单地从其四元数形式 v′ 中提取出来获得。

四元数乘法在实际游戏中的各种情况下都很有用。例如，假设我们想要找到一个描述飞机飞行方向的单位向量。我们进一步假设在我们的游戏中，按照惯例，正 z 轴始终指向物体的前方。因此，根据定义，模型空间中任何物体的前向单位向量始终为 FM ≡ [0 0 1]。要将此向量转换到世界空间，我们可以简单地取飞机的方向四元数 q，并将其与公式 (5.9) 结合使用，将模型空间向量 FM 旋转为其世界空间等价向量 FW（当然，在将这些向量转换为四元数形式之后）：

$$\mathsf{F}_W = \mathsf{q}\mathsf{F}_M\mathsf{q}^{-1} = \mathsf{q}\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}\mathsf{q}^{-1}.$$

5.4.3.1 四元数串联

旋转可以像基于矩阵的变换一样，通过将四元数相乘来连接。例如，考虑三个不同的旋转，分别用四元数 q 1 、 q 2 和 q 3 表示，其对应的矩阵分别为 R 1 、 R 2 和 R 3 。我们希望首先应用旋转 1，然后是旋转 2，最后是旋转 3。可以找到复合旋转矩阵 R net 并将其应用于向量 v，如下所示：

$$\begin{aligned}\mathbf{R}_{\text{net}} &= \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3;\\ \mathbf{v}' &= \mathbf{v}\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3\\ &= \mathbf{v}\mathbf{R}_{\text{net}}.\end{aligned}$$

同样，可以找到复合旋转四元数 q 网络并将其应用于向量 v（四元数形式为 v），如下所示：

$$\begin{aligned}\mathsf{q}_{\text{net}} &= \mathsf{q}_3\mathsf{q}_2\mathsf{q}_1;\\ \mathsf{v}' &= \mathsf{q}_3\mathsf{q}_2\mathsf{q}_1\,\mathsf{v}\,\mathsf{q}_1^{-1}\mathsf{q}_2^{-1}\mathsf{q}_3^{-1}\\ &= \mathsf{q}_{\text{net}}\,\mathsf{v}\,\mathsf{q}_{\text{net}}^{-1}.\end{aligned}$$

注意，四元数乘积的执行顺序必须与旋转的顺序相反（q 3 q 2 q 1）。这是因为四元数旋转总是在向量的两边进行乘法运算，即正四元数在左边，逆四元数在右边。正如我们在公式 (5.8) 中看到的，四元数乘积的逆是各个逆的逆乘积，因此正四元数的读取顺序是从右到左，而逆四元数的读取顺序是从左到右。

The rotated vector $\mathbf{v}'$ is obtained by simply extracting it from its quaternion form v'.

Quaternion multiplication can be useful in all sorts of situations in real games. For example, let's say that we want to find a unit vector describing the direction in which an aircraft is flying. We'll further assume that in our game, the positive z-axis always points toward the front of an object by convention. So the forward unit vector of any object *in model space* is always $\mathbf{F}_M \equiv \begin{bmatrix}0 & 0 & 1\end{bmatrix}$ by definition. To transform this vector into world space, we can simply take our aircraft's orientation quaternion q and use it with Equation (5.9) to rotate our model-space vector $\mathbf{F}_M$ into its world-space equivalent $\mathbf{F}_W$ (after converting these vectors into quaternion form, of course):

$$\mathsf{F}_W = \mathsf{q}\mathsf{F}_M\mathsf{q}^{-1} = \mathsf{q}\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}\mathsf{q}^{-1}.$$

5.4.3.1 Quaternion Concatenation

Rotations can be *concatenated* in exactly the same way that matrix-based transformations can, by multiplying the quaternions together. For example, consider three distinct rotations, represented by the quaternions q_1, q_2 and q_3, with matrix equivalents $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$. We want to apply rotation 1 first, followed by rotation 2 and finally rotation 3. The composite rotation matrix $\mathbf{R}_{\text{net}}$ can be found and applied to a vector $\mathbf{v}$ as follows:

$$\begin{aligned}\mathbf{R}_{\text{net}} &= \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3;\\ \mathbf{v}' &= \mathbf{v}\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3\\ &= \mathbf{v}\mathbf{R}_{\text{net}}.\end{aligned}$$

Likewise, the composite rotation quaternion q_{net} can be found and applied to vector $\mathbf{v}$ (in its quaternion form, v) as follows:

$$\begin{aligned}\mathsf{q}_{\text{net}} &= \mathsf{q}_3\mathsf{q}_2\mathsf{q}_1;\\ \mathsf{v}' &= \mathsf{q}_3\mathsf{q}_2\mathsf{q}_1\,\mathsf{v}\,\mathsf{q}_1^{-1}\mathsf{q}_2^{-1}\mathsf{q}_3^{-1}\\ &= \mathsf{q}_{\text{net}}\,\mathsf{v}\,\mathsf{q}_{\text{net}}^{-1}.\end{aligned}$$

Notice how the quaternion product must be performed in an order *opposite* to that in which the rotations are applied ($\mathsf{q}_3\mathsf{q}_2\mathsf{q}_1$). This is because quaternion rotations always multiply on *both* sides of the vector, with the uninverted quaternions on the left and the inverted quaternions on the right. As we saw in Equation (5.8), the inverse of a quaternion product is the reverse product of the individual inverses, so the uninverted quaternions read right-to-left while the inverted quaternions read left-to-right.

5.4.4 四元数-矩阵等价

我们可以将任意三维旋转在 3 × 3 矩阵表示 R 和四元数表示 q 之间自由转换。设 q = [q V q S] = [q Vx q Vy q Vz q S] = [xyzw]，则 R 的表达式如下：

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2zw & 2xz-2yw \\ 2xy-2zw & 1-2x^2-2z^2 & 2yz+2xw \\ 2xz+2yw & 2yz-2xw & 1-2x^2-2y^2 \end{bmatrix}.$$

同样，给定 R，我们可以按如下方式求出 q （其中 q[0] = q Vx，q[1] = q Vy，q[2] = q Vz，q[3] = q S ）。此代码假设我们在 C/C++ 中使用行向量（即，矩阵的行与上面所示的矩阵 R 的行相对应）。该代码改编自 Nick Bobic 于 1998 年 7 月 5 日发表于 Gamasutra 的一篇文章，可在此处获取：

http://www.gamasutra.com/view/feature/3278/rotating_objects_using_quaternions.php。有关利用有关矩阵性质的各种假设将矩阵转换为四元数的一些更快方法的讨论，请参阅http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm。

```
void matrixToQuaternion(
      const float  R[3][3],
      float        q[/*4*/])
{
   float trace = R[0][0] + R[1][1] + R[2][2];

   // check the diagonal
   if (trace > 0.0f)
   {
      float s = sqrt(trace + 1.0f);
      q[3] = s * 0.5f;

      float t = 0.5f / s;
      q[0] = (R[2][1] - R[1][2]) * t;
      q[1] = (R[0][2] - R[2][0]) * t;
      q[2] = (R[1][0] - R[0][1]) * t;
   }
   else
   {
      // diagonal is negative
      int i = 0;
      if (R[1][1] > R[0][0]) i = 1;
      if (R[2][2] > R[i][i]) i = 2;

      static const int NEXT[3] = {1, 2, 0};
```


5.4.4 Quaternion-Matrix Equivalence

We can convert any 3D rotation freely between a 3×3 matrix representation $\mathbf{R}$ and a quaternion representation q. If we let $\mathtt{q} = [\mathbf{q}_V \;\; q_S] = [q_{Vx} \;\; q_{Vy} \;\; q_{Vz} \;\; q_S] = [x \;\; y \;\; z \;\; w]$, then we can find $\mathbf{R}$ as follows:

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2zw & 2xz-2yw \\ 2xy-2zw & 1-2x^2-2z^2 & 2yz+2xw \\ 2xz+2yw & 2yz-2xw & 1-2x^2-2y^2 \end{bmatrix}.$$

Likewise, given $\mathbf{R}$, we can find q as follows (where `q[0]` = q_{Vx}, `q[1]` = q_{Vy}, `q[2]` = q_{Vz} and `q[3]` = q_S). This code assumes that we are using row vectors in C/C++ (i.e., that the rows of the matrix correspond to the rows of the matrix $\mathbf{R}$ shown above). The code was adapted from a *Gamasutra* article by Nick Bobic, published on July 5, 1998, which is available here: http://www.gamasutra.com/view/feature/3278/rotating_objects_using_quaternions.php. For a discussion of some even faster methods for converting a matrix to a quaternion, leveraging various assumptions about the nature of the matrix, see http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm.

```
void matrixToQuaternion(
      const float  R[3][3],
      float        q[/*4*/])
{
   float trace = R[0][0] + R[1][1] + R[2][2];

   // check the diagonal
   if (trace > 0.0f)
   {
      float s = sqrt(trace + 1.0f);
      q[3] = s * 0.5f;

      float t = 0.5f / s;
      q[0] = (R[2][1] - R[1][2]) * t;
      q[1] = (R[0][2] - R[2][0]) * t;
      q[2] = (R[1][0] - R[0][1]) * t;
   }
   else
   {
      // diagonal is negative
      int i = 0;
      if (R[1][1] > R[0][0]) i = 1;
      if (R[2][2] > R[i][i]) i = 2;

      static const int NEXT[3] = {1, 2, 0};
```

```
        int j = NEXT[i];
        int k = NEXT[j];

        float s = sqrt((R[i][j]
                       - (R[j][j] + R[k][k]))
                         + 1.0f);

        q[i] = s * 0.5f;

        float t;
        if (s != 0.0)  t = 0.5f / s;
        else           t = s;

        q[3] = (R[k][j] - R[j][k]) * t;
        q[j] = (R[j][i] + R[i][j]) * t;
        q[k] = (R[k][i] + R[i][k]) * t;
    }
}
```

让我们暂停一下，考虑一下符号约定。在本书中，我们这样写四元数：[xyzw] 。这与许多关于四元数作为复数扩展的学术论文中的 [wxyz] 约定不同。我们的约定源于努力与将齐次向量写为 [xyz 1] （末尾为 w = 1）的常见做法保持一致。学术约定源于四元数和复数之间的相似之处。常规二维复数通常写为 c = a + jb 的形式，相应的四元数符号为 q = w + ix + jy + kz 。所以要小心 - 在深入研究论文之前，请确保您知道正在使用哪种约定！

5.4.5 旋转线性插值

旋转插值在游戏引擎的动画、动力学和摄像机系统中有着广泛的应用。借助四元数，旋转可以像向量和点一样轻松地进行插值。

最简单且计算量最小的方法是，对想要插值的四元数执行四维向量 LERP。给定两个四元数 q A 和 q B ，分别代表旋转 A 和 B ，我们可以找到一个中间旋转 q LERP ，其值为从 A 到 B 路径的 β 百分比，如下所示：


```
        int j = NEXT[i];
        int k = NEXT[j];

        float s = sqrt((R[i][j]
                       - (R[j][j] + R[k][k]))
                         + 1.0f);

        q[i] = s * 0.5f;

        float t;
        if (s != 0.0)  t = 0.5f / s;
        else           t = s;

        q[3] = (R[k][j] - R[j][k]) * t;
        q[j] = (R[j][i] + R[i][j]) * t;
        q[k] = (R[k][i] + R[i][k]) * t;
    }
}
```

Let's pause for a moment to consider notational conventions. In this book, we write our quaternions like this: $[\,x\ \ y\ \ z\ \ w\,]$. This differs from the $[\,w\ \ x\ \ y\ \ z\,]$ convention found in many academic papers on quaternions as an extension of the complex numbers. Our convention arises from an effort to be consistent with the common practice of writing homogeneous vectors as $[\,x\ \ y\ \ z\ \ 1\,]$ (with the $w = 1$ at the end). The academic convention arises from the parallels between quaternions and complex numbers. Regular two-dimensional complex numbers are typically written in the form $c = a + jb$, and the corresponding quaternion notation is $q = w + ix + jy + kz$. So be careful out there—make sure you know which convention is being used before you dive into a paper head first!

5.4.5 Rotational Linear Interpolation

Rotational interpolation has many applications in the animation, dynamics and camera systems of a game engine. With the help of quaternions, rotations can be easily interpolated just as vectors and points can.

The easiest and least computationally intensive approach is to perform a four-dimensional vector LERP on the quaternions you wish to interpolate. Given two quaternions q_A and q_B representing rotations A and B, we can find an intermediate rotation $\mathrm{q}_{\mathrm{LERP}}$ that is β percent of the way from A to B as fol-

图 5.23 两个四元数 q A 和 q B 之间的线性插值（LERP）。

低点：

$$\mathsf{q}_{\text{LERP}} = \text{LERP}(\mathsf{q}_A, \mathsf{q}_B, \beta) = \frac{(1-\beta)\mathsf{q}_A + \beta\mathsf{q}_B}{|(1-\beta)\mathsf{q}_A + \beta\mathsf{q}_B|}$$
$$= \text{normalize}\left(\begin{bmatrix}(1-\beta)q_{Ax} + \beta q_{Bx}\\ (1-\beta)q_{Ay} + \beta q_{By}\\ (1-\beta)q_{Az} + \beta q_{Bz}\\ (1-\beta)q_{Aw} + \beta q_{Bw}\end{bmatrix}^{\mathrm{T}}\right).$$

请注意，结果插值四元数必须重新规范化。
这是必要的，因为 LERP 操作通常不保留向量的长度。

从几何学上讲，q LERP = LERP（q A , q B , β）表示方向位于方向 A 到方向 B 之间 β % 的四元数，如图 5.23 所示（为清晰起见，以二维表示）。从数学上讲，LERP 运算的结果是两个四元数的加权平均值，权重分别为（1 − β）和 β（注意，这两个权重之和为 1）。

5.4.5.1 球面线性插值

LERP 操作的问题在于，它没有考虑到四元数实际上是四维超球面上的点。LERP 实际上是沿着超球面的弦进行插值，而不是沿着超球面本身的表面进行插值。这导致当参数 β 以恒定速率变化时，旋转动画的角速度不恒定。旋转在端点处会显得更慢，而在动画中间会显得更快。

为了解决这个问题，我们可以使用 LERP 运算的一种变体，即球面线性插值，简称 SLERP。SLERP 运算使用正弦和余弦沿着四维超球面的大圆进行插值，

Figure 5.23. Linear interpolation (LERP) between two quaternions q_A and q_B.

lows:

$$\mathsf{q}_{\text{LERP}} = \text{LERP}(\mathsf{q}_A, \mathsf{q}_B, \beta) = \frac{(1-\beta)\mathsf{q}_A + \beta\mathsf{q}_B}{|(1-\beta)\mathsf{q}_A + \beta\mathsf{q}_B|}$$
$$= \text{normalize}\left(\begin{bmatrix}(1-\beta)q_{Ax} + \beta q_{Bx}\\ (1-\beta)q_{Ay} + \beta q_{By}\\ (1-\beta)q_{Az} + \beta q_{Bz}\\ (1-\beta)q_{Aw} + \beta q_{Bw}\end{bmatrix}^{\mathrm{T}}\right).$$

Notice that the resultant interpolated quaternion had to be renormalized. This is necessary because the LERP operation does not preserve a vector's length in general.

Geometrically, $\mathsf{q}_{\text{LERP}} = \text{LERP}(\mathsf{q}_A, \mathsf{q}_B, \beta)$ is the quaternion whose orientation lies β percent of the way from orientation A to orientation B, as shown (in two dimensions for clarity) in Figure 5.23. Mathematically, the LERP operation results in a *weighted average* of the two quaternions, with weights $(1-\beta)$ and β (notice that these two weights sum to 1).

5.4.5.1 Spherical Linear Interpolation

The problem with the LERP operation is that it does not take account of the fact that quaternions are really points on a four-dimensional *hypersphere*. A LERP effectively interpolates along a *chord* of the hypersphere, rather than along the surface of the hypersphere itself. This leads to rotation animations that do not have a constant angular speed when the parameter β is changing at a constant rate. The rotation will appear slower at the end points and faster in the middle of the animation.

To solve this problem, we can use a variant of the LERP operation known as *spherical linear interpolation*, or SLERP for short. The SLERP operation uses sines and cosines to interpolate along a *great circle* of the 4D hypersphere,

图 5.24 沿 4D 超球面的大圆弧进行球面线性插值。

而不是沿着弦线，如图 5.24 所示。当 β 以恒定速率变化时，角速度也保持恒定。

SLERP 的公式与 LERP 公式类似，但权重（1 − β）和 β 被涉及两个四元数之间角度的正弦的权重 wp 和 wq 取代。

$$\mathrm{SLERP}(\mathsf{p}, \mathsf{q}, \beta) = w_p \mathsf{p} + w_q \mathsf{q},$$

在哪里

$$w_p = \frac{\sin(1-\beta)\theta}{\sin\theta},$$
$$w_q = \frac{\sin\beta\theta}{\sin\theta}.$$

任意两个单位长度四元数之间的夹角余弦可以通过它们的四维点积来求得。一旦我们知道了 cosθ，我们就可以很容易地计算出夹角 θ 以及我们需要的各种正弦值：

$$\cos\theta = \mathsf{p}\cdot\mathsf{q} = p_x q_x + p_y q_y + p_z q_z + p_w q_w;$$
$$\theta = \cos^{-1}(\mathsf{p}\cdot\mathsf{q}).$$

5.4.5.2 是否使用 SLERP（这仍然是个问题）

关于是否应该在游戏引擎中使用 SLERP，目前尚无定论。Jonathan Blow 写了一篇很棒的文章，他认为 SLERP 成本过高，而 LERP 的质量其实并不算太差——因此，他建议我们应该了解 SLERP，但避免在游戏引擎中使用它（参见 http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/index.html）。

Figure 5.24. Spherical linear interpolation along a great circle arc of a 4D hypersphere.

rather than along a chord, as shown in Figure 5.24. This results in a constant angular speed when β varies at a constant rate.

The formula for SLERP is similar to the LERP formula, but the weights $(1-\beta)$ and β are replaced with weights w_p and w_q involving sines of the angle between the two quaternions.

$$\mathrm{SLERP}(\mathsf{p}, \mathsf{q}, \beta) = w_p \mathsf{p} + w_q \mathsf{q},$$

where

$$w_p = \frac{\sin(1-\beta)\theta}{\sin\theta},$$
$$w_q = \frac{\sin\beta\theta}{\sin\theta}.$$

The cosine of the angle between any two unit-length quaternions can be found by taking their four-dimensional dot product. Once we know $\cos\theta$, we can calculate the angle θ and the various sines we need quite easily:

$$\cos\theta = \mathsf{p}\cdot\mathsf{q} = p_x q_x + p_y q_y + p_z q_z + p_w q_w;$$
$$\theta = \cos^{-1}(\mathsf{p}\cdot\mathsf{q}).$$

5.4.5.2 To SLERP or Not to SLERP (That's Still the Question)

The jury is still out on whether or not to use SLERP in a game engine. Jonathan Blow wrote a great article positing that SLERP is too expensive, and LERP's quality is not really that bad—therefore, he suggests, we should understand SLERP but avoid it in our game engines (see http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/index.html).

另一方面，我在顽皮狗的一些同事发现，好的 SLERP 实现的性能几乎与 LERP 一样好。（例如，在 PS3 的 SPU 上，顽皮狗 Ice 团队的 SLERP 实现每个关节需要 20 个周期，而其 LERP 实现每个关节需要 16.25 个周期。）因此，我个人建议您在做出任何决定之前，先对 SLERP 和 LERP 实现进行性能分析。如果 SLERP 的性能损失不至于让人无法接受，我建议您尝试一下，因为它可能会让动画效果略有改善。但如果您的 SLERP 速度较慢（并且您无法加快速度，或者您没有时间这样做），那么 LERP 通常足以满足大多数需求。

5.5 旋转表示的比较

我们已经了解，旋转可以用多种不同的方式表示。本节总结了最常见的旋转表示方法，并概述了它们的优缺点。没有一种表示方法适用于所有情况。使用本节中的信息，您应该能够为特定应用选择最佳表示方法。

5.5.1 欧拉角

我们在 5.3.9.1 节中简要探讨了欧拉角。用欧拉角表示的旋转由三个标量值组成：偏航角、俯仰角和滚转角。这些量有时用三维向量 [θ Y θ P θ R] 表示。

这种表示法的优点在于简单、体积小（三个浮点数）且直观——偏航角、俯仰角和滚转角很容易直观地表示出来。您还可以轻松地对绕单轴的简单旋转进行插值。例如，通过对标量 θ Y 进行线性插值，很容易找到两个不同偏航角之间的中间旋转。然而，当旋转绕任意方向的轴时，欧拉角很难进行插值。

此外，欧拉角容易出现一种称为万向节锁的情况。
当旋转 90 度导致三个主轴中的一个“坍缩”到另一个主轴上时，就会发生这种情况。例如，如果绕 x 轴旋转 90 度，y 轴就会坍缩到 z 轴上。这会阻止任何绕原始 y 轴的进一步旋转，因为绕 y 轴和 z 轴的旋转实际上已经变得等价了。

欧拉角的另一个问题是，绕每个轴旋转的顺序很重要。顺序可能是PYR、YPR、RYP等等，并且每种顺序都可能产生不同的复合旋转。不

On the other hand, some of my colleagues at Naughty Dog have found that a good SLERP implementation performs nearly as well as LERP. (For example, on the PS3's SPUs, Naughty Dog's Ice team's implementation of SLERP takes 20 cycles per joint, while its LERP implementation takes 16.25 cycles per joint.) Therefore, I'd personally recommend that you profile your SLERP and LERP implementations before making any decisions. If the performance hit for SLERP isn't unacceptable, I say go for it, because it may result in slightly better-looking animations. But if your SLERP is slow (and you cannot speed it up, or you just don't have the time to do so), then LERP is usually good enough for most purposes.

5.5 Comparison of Rotational Representations

We've seen that rotations can be represented in quite a few different ways. This section summarizes the most common rotational representations and outlines their pros and cons. No one representation is ideal in all situations. Using the information in this section, you should be able to select the best representation for a particular application.

5.5.1 Euler Angles

We briefly explored *Euler angles* in Section 5.3.9.1. A rotation represented via Euler angles consists of three scalar values: yaw, pitch and roll. These quantities are sometimes represented by a 3D vector $\begin{bmatrix}\theta_Y & \theta_P & \theta_R\end{bmatrix}$.

The benefits of this representation are its simplicity, its small size (three floating-point numbers) and its intuitive nature—yaw, pitch and roll are easy to visualize. You can also easily interpolate simple rotations about a single axis. For example, it's trivial to find intermediate rotations between two distinct yaw angles by linearly interpolating the scalar θ_Y. However, Euler angles cannot be interpolated easily when the rotation is about an arbitrarily oriented axis.

In addition, Euler angles are prone to a condition known as *gimbal lock*. This occurs when a 90-degree rotation causes one of the three principal axes to "collapse" onto another principal axis. For example, if you rotate by 90 degrees about the x-axis, the y-axis collapses onto the z-axis. This prevents any further rotations about the original y-axis, because rotations about y and z have effectively become equivalent.

Another problem with Euler angles is that the order in which the rotations are performed around each axis matters. The order could be PYR, YPR, RYP and so on, and each ordering may produce a different composite rotation. No

所有学科的欧拉角都遵循一个标准的旋转顺序（尽管某些学科确实遵循特定的约定）。因此，旋转角度 [θ Y θ P θ R] 并不能唯一地定义特定的旋转——你需要知道旋转顺序才能正确解释这些数字。

欧拉角的最后一个问题是，它们依赖于 x 轴、y 轴和 z 轴与旋转物体自然的正面、左右和向上方向的映射。例如，偏航总是被定义为绕向上轴的旋转，但如果没有其他信息，我们无法判断这究竟是绕 x 轴、y 轴还是 z 轴的旋转。

3×3矩阵 5.5.2

3 × 3 矩阵是一种便捷有效的旋转表示，原因有很多。它不会出现万向节锁，并且可以唯一地表示任意旋转。旋转可以通过矩阵乘法（即一系列点积）直接应用于点和向量。现在大多数 CPU 和所有 GPU 都内置了对硬件加速点积和矩阵乘法的支持。旋转也可以通过求逆矩阵来反转，对于纯旋转矩阵来说，这相当于求转置矩阵——一个简单的运算。4 × 4 矩阵提供了一种以完全一致的方式表示任意仿射变换（旋转、平移和缩放）的方法。

然而，旋转矩阵并不是特别直观。查看一大堆数字表格并不能帮助我们在三维空间中描绘出相应的变换。此外，旋转矩阵也不容易进行插值。最后，相对于欧拉角（三个浮点数），旋转矩阵占用的存储空间更大（九个浮点数）。

5.5.3 轴+角度

我们可以将旋转表示为一个单位向量，定义旋转轴加上一个表示旋转角度的标量。这被称为“轴+角度”表示法，有时用四维向量 [α θ] = [αxαyαz θ] 表示，其中 α 表示旋转轴，θ 表示以弧度为单位的角度。在右手坐标系中，正旋转的方向由右手定则定义；而在左手坐标系中，我们使用左手定则。

轴+角度表示的优点在于它相当直观且紧凑。（它只需要四个浮点数，而 3×3 矩阵则需要九个浮点数。）

one standard rotation order exists for Euler angles across all disciplines (although certain disciplines do follow specific conventions). So the rotation angles $\begin{bmatrix}\theta_Y & \theta_P & \theta_R\end{bmatrix}$ do not uniquely define a particular rotation—you need to know the rotation order to interpret these numbers properly.

A final problem with Euler angles is that they depend upon the mapping from the x-, y- and z-axes onto the natural *front*, *left/right* and *up* directions for the object being rotated. For example, yaw is always defined as rotation about the *up* axis, but without additional information we cannot tell whether this corresponds to a rotation about x, y or z.

5.5.2 3 × 3 Matrices

A 3 × 3 matrix is a convenient and effective rotational representation for a number of reasons. It does not suffer from gimbal lock, and it can represent arbitrary rotations uniquely. Rotations can be applied to points and vectors in a straightforward manner via matrix multiplication (i.e., a series of dot products). Most CPUs and all GPUs now have built-in support for hardware-accelerated dot products and matrix multiplication. Rotations can also be reversed by finding an inverse matrix, which for a pure rotation matrix is the same thing as finding the transpose—a trivial operation. And 4 × 4 matrices offer a way to represent arbitrary affine transformations—rotations, translations and scaling—in a totally consistent way.

However, rotation matrices are not particularly intuitive. Looking at a big table of numbers doesn't help one picture the corresponding transformation in three-dimensional space. Also, rotation matrices are not easily interpolated. Finally, a rotation matrix takes up a lot of storage (nine floating-point numbers) relative to Euler angles (three floats).

5.5.3 Axis + Angle

We can represent rotations as a unit vector, defining the axis of rotation plus a scalar for the angle of rotation. This is known as an *axis+angle* representation, and it is sometimes denoted by the four-dimensional vector $\begin{bmatrix}\mathbf{a} & \theta\end{bmatrix} = \begin{bmatrix}a_x & a_y & a_z & \theta\end{bmatrix}$, where $\mathbf{a}$ is the axis of rotation and θ the angle in radians. In a right-handed coordinate system, the direction of a positive rotation is defined by the right-hand rule, while in a left-handed system, we use the left-hand rule instead.

The benefits of the axis+angle representation are that it is reasonably intuitive and also compact. (It only requires four floating-point numbers, as opposed to the nine required for a 3 × 3 matrix.)

轴+角表示的一个重要限制是旋转无法轻易进行插值。此外，这种格式的旋转无法直接应用于点和向量——需要先将轴+角表示转换为矩阵或四元数。

5.5.4 四元数

正如我们所见，单位长度四元数可以以类似于轴+角度表示的方式表示三维旋转。这两种表示方式的主要区别在于，四元数的旋转轴是通过旋转半角的正弦值进行缩放的，并且我们不是将角度存储在向量的第四个分量中，而是存储半角的余弦值。

与轴+角度表示相比，四元数公式有两个巨大的优势。首先，它允许将旋转连接起来，并通过四元数乘法直接应用于点和向量。其次，它允许通过简单的 LERP 或 SLERP 操作轻松进行旋转插值。它的小尺寸（四个浮点数）也是优于矩阵公式的一个优势。

5.5.5 SRT 变换

四元数本身只能表示旋转，而 4×4 矩阵可以表示任意仿射变换（旋转、平移和缩放）。当四元数与平移向量和比例因子（用于均匀缩放的标量或用于非均匀缩放的向量）组合时，我们就有了一种可行的替代 4×4 矩阵表示仿射变换的方法。我们有时称之为 SRT 变换，因为它包含一个比例因子、一个旋转四元数和一个平移向量。（有时也被称为 SQT 变换，因为旋转是一个四元数。）

$$\mathrm{SRT} = \begin{bmatrix} s & \mathsf{q} & \mathbf{t} \end{bmatrix} \qquad \text{（均匀尺度 s ），}$$

or

$$\mathrm{SRT} = \begin{bmatrix} \mathbf{s} & \mathsf{q} & \mathbf{t} \end{bmatrix} \qquad \text{（非均匀尺度向量 s ）。}$$

SRT 变换在计算机动画中应用广泛，因为它们的尺寸较小（均匀缩放时仅需 8 个浮点数，非均匀缩放时则需 10 个浮点数，而 4×3 矩阵则需要 12 个浮点数），并且易于插值。平移向量和比例因子通过 LERP 进行插值，四元数则可以通过 LERP 或 SLERP 进行插值。

One important limitation of the axis+angle representation is that rotations cannot be easily interpolated. Also, rotations in this format cannot be applied to points and vectors in a straightforward way—one needs to convert the axis+angle representation into a matrix or quaternion first.

5.5.4 Quaternions

As we've seen, a unit-length quaternion can represent 3D rotations in a manner analogous to the axis+angle representation. The primary difference between the two representations is that a quaternion's axis of rotation is scaled by the sine of the half-angle of rotation, and instead of storing the angle in the fourth component of the vector, we store the cosine of the half-angle.

The quaternion formulation provides two immense benefits over the axis +angle representation. First, it permits rotations to be concatenated and applied directly to points and vectors via quaternion multiplication. Second, it permits rotations to be easily interpolated via simple LERP or SLERP operations. Its small size (four floating-point numbers) is also a benefit over the matrix formulation.

5.5.5 SRT Transformations

By itself, a quaternion can only represent a rotation, whereas a 4×4 matrix can represent an arbitrary affine transformation (rotation, translation and scale). When a quaternion is combined with a *translation vector* and a *scale factor* (either a scalar for uniform scaling or a vector for nonuniform scaling), then we have a viable alternative to the 4×4 matrix representation of affine transformations. We sometimes call this an *SRT transform*, because it contains a *s*cale factor, a *r*otation quaternion and a *t*ranslation vector. (It's also sometimes called an *SQT*, because the rotation is a quaternion.)

$$\mathrm{SRT} = \begin{bmatrix} s & \mathsf{q} & \mathbf{t} \end{bmatrix} \qquad \text{(uniform scale } s\text{)},$$

or

$$\mathrm{SRT} = \begin{bmatrix} \mathbf{s} & \mathsf{q} & \mathbf{t} \end{bmatrix} \qquad \text{(nonuniform scale vector } \mathbf{s}\text{)}.$$

SRT transforms are widely used in computer animation because of their smaller size (eight floats for uniform scale, or ten floats for nonuniform scale, as opposed to the 12 floating-point numbers needed for a 4×3 matrix) and their ability to be easily interpolated. The translation vector and scale factor are interpolated via LERP, and the quaternion can be interpolated with either LERP or SLERP.

5.5.6 对偶四元数

刚性变换是一种涉及旋转和平移的变换——一种“螺旋”运动。这种变换在动画和机器人技术中很常见。刚性变换可以用称为对偶四元数的数学对象来表示。与典型的矢量四元数表示相比，对偶四元数表示具有许多优势。关键优势在于，线性插值混合可以以恒速、最短路径、坐标不变的方式执行，类似于使用 LERP 表示平移矢量，使用 SLERP 表示旋转四元数（参见第 5.4.5.1 节），但这种方式很容易推广到涉及三个或更多变换的混合。

对偶四元数与普通四元数类似，只是它的四个分量是对偶数，而不是常规的实数。对偶数可以写成非对偶部分与对偶部分的和，如下所示：ˆ a = a + ε b 。其中 ε 是一个称为对偶单位的神奇数字，其定义为 ε 2 = 0（但 ε 本身不为零）。这类似于在将复数写成实部与虚部之和时使用的虚数 j = √ − 1： c = a + jb 。

由于每个对偶数都可以用两个实数（非对偶部分和对偶部分，a 和 b ）表示，因此对偶四元数可以用一个八元素向量表示。它也可以表示为两个普通四元数的和，其中第二个四元数乘以对偶单位，如下所示：

$\hat{q} = q_a + \varepsilon q_b$.

关于对偶数和对偶四元数的完整讨论超出了本文的范围。不过，Kavan 等人撰写的优秀论文《用于刚性变换混合的对偶四元数》概述了使用对偶四元数表示刚性变换的理论和实践——该论文可在线访问：https://bit.ly/2vjD5sz。需要注意的是，本文中对偶数的形式为 ˆ a = a 0 + ε a ε ，而上文中我使用了 a + ε b 来强调对偶数与复数之间的相似性。1

5.5.7 旋转和自由度

“自由度”（简称 DOF）是指物体物理状态（位置和方向）可以相互独立地改变的方式的数量。你可能听说过“六自由度”这个说法。

1 我个人更喜欢用符号 a 1 而不是 a 0 ，这样对偶数就可以写成 ˆ a = (1) a 1 + (ε) a ε 。就像我们在复平面上绘制复数一样，我们可以将实数单位视为沿实轴的“基向量”，将对偶单位 ε 视为沿对偶轴的“基向量”。

5.5.6 Dual Quaternions

A *rigid transformation* is a transformation involving a rotation and a translation—a “corkscrew” motion. Such transformations are prevalent in animation and robotics. A rigid transformation can be represented using a mathematical object known as a *dual quaternion*. The dual quaternion representation offers a number of benefits over the typical vector-quaternion representation. The key benefit is that linear interpolation blending can be performed in a constant-speed, shortest-path, coordinate-invariant manner, similar to using LERP for translation vectors and SLERP for rotational quaternions (see Section 5.4.5.1), but in a way that is easily generalizable to blends involving three or more transforms.

A dual quaternion is like an ordinary quaternion, except that its four components are *dual numbers* instead of regular real-valued numbers. A dual number can be written as the sum of a non-dual part and a dual part as follows: $\hat{a} = a + \varepsilon b$. Here ε is a magical number called the *dual unit*, defined in such a way that $\varepsilon^2 = 0$ (yet without ε itself being zero). This is analogous to the imaginary number $j = \sqrt{-1}$ used when writing a complex number as the sum of a real and an imaginary part: $c = a + jb$.

Because each dual number can be represented by two real numbers (the non-dual and dual parts, a and b), a dual quaternion can be represented by an eight-element vector. It can also be represented as the sum of two ordinary quaternions, where the second one is multiplied by the dual unit, as follows: $\hat{q} = q_a + \varepsilon q_b$.

A full discussion of dual numbers and dual quaternions is beyond our scope here. However, the excellent paper entitled, “Dual Quaternions for Rigid Transformation Blending” by Kavan et al. outlines the theory and practice of using dual quaternions to represent rigid transformations—it is available online at https://bit.ly/2vjD5sz. Note that in this paper, a dual number is written in the form $\hat{a} = a_0 + \varepsilon a_\varepsilon$, whereas I have used $a + \varepsilon b$ above to underscore the similarity between dual numbers and complex numbers.[1]

5.5.7 Rotations and Degrees of Freedom

The term *“degrees of freedom”* (or *DOF* for short) refers to the number of mutually independent ways in which an object’s physical state (position and orientation) can change. You may have encountered the phrase “six degrees of

[1] Personally I would have preferred the symbol a_1 over a_0, so that a dual number would be written $\hat{a} = (1)a_1 + (\varepsilon)a_\varepsilon$. Just as when we plot a complex number in the complex plane, we can think of the real unit as a “basis vector” along the real axis, and the dual unit ε as a “basis vector” along the dual axis.

自由度”在力学、机器人、航空等领域被广泛应用。它指的是一个三维物体（其运动不受人为约束）在平移（沿x、y、z轴）时拥有三个自由度，在旋转（绕x、y、z轴）时拥有三个自由度，总共六个自由度。

DOF 概念将帮助我们理解，不同的旋转表示法如何能够使用不同数量的浮点参数，但所有表示法都只指定三个自由度的旋转。例如，欧拉角需要三个浮点数，而轴+角和四元数表示法需要四个浮点数，而 3×3 矩阵则需要九个浮点数。这些表示法如何能够描述三自由度旋转？

答案在于约束。所有 3D 旋转表示都采用三个或更多浮点参数，但有些表示还会对这些参数设置一个或多个约束。这些约束表明参数之间并非相互独立——一个参数的更改会导致其他参数的更改，以保持约束的有效性。如果我们从浮点参数的数量中减去约束的数量，就得到了自由度的数量——对于 3D 旋转来说，这个数字应该始终为 3：

$$N_{\text{DOF}} = N_{\text{parameters}} - N_{\text{constraints}}. \tag{5.10}$$

以下列表显示了本书中遇到的每个旋转表示中公式 (5.10) 的实际应用。

- 欧拉角。3 个参数 − 0 个约束 = 3 个自由度。
- 轴+角度。4 个参数 − 1 个约束 = 3 个自由度。
  约束：轴被约束为单位长度。
- 四元数。4 个参数 − 1 个约束 = 3 个自由度。
  约束：四元数被约束为单位长度。
- 3×3 矩阵。9 个参数 − 6 个约束 = 3 个自由度。
  约束：所有三行和所有三列必须是单位长度（当被视为三元素向量时）。

5.6 其他有用的数学对象

作为游戏工程师，除了点、向量、矩阵和四元数之外，我们还会遇到许多其他数学对象。本节简要概述其中最常见的几种。

freedom” in fields such as mechanics, robotics and aeronautics. This refers to the fact that a three-dimensional object (whose motion is not artificially constrained) has three degrees of freedom in its translation (along the x-, y- and z-axes) and three degrees of freedom in its rotation (about the x-, y- and z-axes), for a total of six degrees of freedom.

The DOF concept will help us to understand how different rotational representations can employ different numbers of floating-point parameters, yet all specify rotations with only three degrees of freedom. For example, Euler angles require three floats, but axis+angle and quaternion representations use four floats, and a 3×3 matrix takes up nine floats. How can these representations all describe 3-DOF rotations?

The answer lies in *constraints*. All 3D rotational representations employ three or more floating-point parameters, but some representations also have one or more constraints on those parameters. The constraints indicate that the parameters are not *independent*—a change to one parameter induces changes to the other parameters in order to maintain the validity of the constraint(s). If we subtract the number of constraints from the number of floating-point parameters, we arrive at the number of degrees of freedom—and this number should always be three for a 3D rotation:

$$N_{\text{DOF}} = N_{\text{parameters}} - N_{\text{constraints}}. \tag{5.10}$$

The following list shows Equation (5.10) in action for each of the rotational representations we've encountered in this book.

- *Euler Angles*. 3 parameters − 0 constraints = 3 DOF.
- *Axis+Angle*. 4 parameters − 1 constraint = 3 DOF.
  Constraint: Axis is constrained to be unit length.
- *Quaternion*. 4 parameters − 1 constraint = 3 DOF.
  Constraint: Quaternion is constrained to be unit length.
- 3×3 *Matrix*. 9 parameters − 6 constraints = 3 DOF.
  Constraints: All three rows and all three columns must be of unit length (when treated as three-element vectors).

5.6 Other Useful Mathematical Objects

As game engineers, we will encounter a host of other mathematical objects in addition to points, vectors, matrices and quaternions. This section briefly outlines the most common of these.

图 5.25. 直线的参数方程。

图 5.26 射线的参数方程。

5.6.1 线、射线和线段

一条无限长的直线可以用一个点 P 0 加上一个沿直线方向的单位向量 u 来表示。直线的参数方程从初始点 P 0 出发，沿单位向量 v 的方向移动任意距离 t，从而描绘出直线上所有可能的点 P 。无限大的点集 P 变成了标量参数 t 的矢量函数：

$$\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}, \quad \text{where } -\infty < t < \infty. \tag{5.11}$$

如图 5.25 所示。

射线是一条只沿一个方向延伸到无穷远的线。它很容易表示为 P(t)，其中 t ⩾ 0，如图 5.26 所示。

线段的两端分别以 P 0 和 P 1 为界。它也可以用 P（t）表示，具体如下（其中 L = P 1 − P 0 ，L = | L | 为线段长度，u =（1/ L）L 为 L 方向上的单位向量）：

1. P(t)=P0+tu，其中0⩽t⩽L，或
2. P(t)=P0+tL，其中0⩽t⩽1。

后一种格式（如图 5.27 所示）尤其方便，因为参数 t 是归一化的；换句话说，无论我们处理的是哪条线段，t 总是从 0 到 1。这意味着我们不必将约束 L 存储在单独的浮点参数中；它已经编码在向量 L = L u 中（无论如何我们都必须存储它）。

5.6.2 球体

球体在游戏引擎编程中随处可见。球体通常定义为中心点 C 加上半径 r，如图 5.28 所示。这可以很好地封装成一个四元素向量 [C x C y C zr] 。正如我们在讨论 SIMD 向量处理时所看到的，将数据封装成一个包含四个 32 位浮点数的向量（即 128 位包）具有明显的好处。

Figure 5.25. Parametric equation of a line.

Figure 5.26. Parametric equation of a ray.

5.6.1 Lines, Rays and Line Segments

An infinite line can be represented by a point $\mathbf{P}_0$ plus a unit vector $\mathbf{u}$ in the direction of the line. A *parametric equation* of a line traces out every possible point $\mathbf{P}$ along the line by starting at the initial point $\mathbf{P}_0$ and moving an arbitrary distance t along the direction of the unit vector $\mathbf{v}$. The infinitely large set of points $\mathbf{P}$ becomes a *vector function* of the scalar parameter t:

$$\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}, \quad \text{where } -\infty < t < \infty. \tag{5.11}$$

This is depicted in Figure 5.25.

A *ray* is a line that extends to infinity in only one direction. This is easily expressed as $\mathbf{P}(t)$ with the constraint $t \geq 0$, as shown in Figure 5.26.

A *line segment* is bounded at both ends by $\mathbf{P}_0$ and $\mathbf{P}_1$. It too can be represented by $\mathbf{P}(t)$, in either one of the following two ways (where $\mathbf{L} = \mathbf{P}_1 - \mathbf{P}_0$, $L = |\mathbf{L}|$ is the length of the line segment, and $\mathbf{u} = (1/L)\mathbf{L}$ is a unit vector in the direction of $\mathbf{L}$):

1. $\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}$, where $0 \leq t \leq L$, or
2. $\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{L}$, where $0 \leq t \leq 1$.

The latter format, depicted in Figure 5.27, is particularly convenient because the parameter t is normalized; in other words, t always goes from zero to one, no matter which particular line segment we are dealing with. This means we do not have to store the constraint L in a separate floating-point parameter; it is already encoded in the vector $\mathbf{L} = L\,\mathbf{u}$ (which we have to store anyway).

5.6.2 Spheres

Spheres are ubiquitous in game engine programming. A sphere is typically defined as a center point $\mathbf{C}$ plus a radius r, as shown in Figure 5.28. This packs nicely into a four-element vector, $\begin{bmatrix} C_x & C_y & C_z & r \end{bmatrix}$. As we saw when we discussed SIMD vector processing, there are distinct benefits to being able to pack data into a vector containing four 32-bit floats (i.e., a 128-bit package).

图 5.27. 线段的参数方程，其归一化参数为 t。

图 5.28.球体的点半径表示。

5.6.3 平面

平面是三维空间中的二维表面。你可能还记得高中代数课本上的知识，平面方程通常写成如下形式：

$$Ax + By + Cz + D = 0.$$

该方程仅对点 P = 的轨迹成立 $\begin{bmatrix} x & y & z \end{bmatrix}$ 位于飞机。

平面可以用点 P 0 和一个垂直于该平面的单位向量 n 来表示。这有时被称为点法线形式，如图 5.29 所示。

图 5.29 点法线形式的平面。

有趣的是，当将传统平面方程中的参数 A 、 B 和 C 解释为三维向量时，该向量位于平面法线的方向。如果将向量 [ABC] 归一化为单位长度，则归一化向量 [abc] = n ，归一化参数 d = D / √ A 2 + B 2 + C 2 就是平面到原点的距离。如果平面的法向量 n 指向原点（即原点位于平面的“正面”），则 d 的符号为正；如果法向量 n 指向原点的反面（即原点位于平面的“背面”），则 d 的符号为负。

Figure 5.27. Parametric equation of a line segment, with normalized parameter t.

Figure 5.28. Point-radius representation of a sphere.

5.6.3 Planes

A *plane* is a 2D surface in 3D space. As you may recall from high-school algebra, the equation of a plane is often written as follows:

$$Ax + By + Cz + D = 0.$$

This equation is satisfied only for the locus of points $\mathbf{P} = \begin{bmatrix} x & y & z \end{bmatrix}$ that lie on the plane.

Planes can be represented by a point $\mathbf{P}_0$ and a unit vector $\mathbf{n}$ that is normal to the plane. This is sometimes called *point-normal form*, as depicted in Figure 5.29.

Figure 5.29. A plane in point-normal form.

It's interesting to note that when the parameters A, B and C from the traditional plane equation are interpreted as a 3D vector, that vector lies in the direction of the plane normal. If the vector $\begin{bmatrix} A & B & C \end{bmatrix}$ is normalized to unit length, then the normalized vector $\begin{bmatrix} a & b & c \end{bmatrix} = \mathbf{n}$, and the normalized parameter $d = D/\sqrt{A^2 + B^2 + C^2}$ is just the distance from the plane to the origin. The sign of d is positive if the plane's normal vector $\mathbf{n}$ is pointing toward the origin (i.e., the origin is on the "front" side of the plane) and negative if the normal is pointing away from the origin (i.e., the origin is "behind" the plane).

另一种看待这个问题的方式是，平面方程和点法线形式实际上只是同一方程的两种写法。想象一下，测试任意点 P = [xyz] 是否位于平面上。为此，我们求出从点 P 沿法线 n = [abc] 到原点的有符号距离，如果这个有符号距离等于平面到原点的有符号距离 d = − n · P 0 ，则 P 必定位于平面上。因此，让我们使它们相等，并展开一些项：

$$
\begin{aligned}
&\text{（P 到原点的有符号距离）} = \text{（平面到原点的有符号距离）} \\
\mathbf{n}\cdot\mathbf{P} &= \mathbf{n}\cdot\mathbf{P}_0 \\
\mathbf{n}\cdot\mathbf{P} - \mathbf{n}\cdot\mathbf{P}_0 &= 0 \\
ax + by + cz - \mathbf{n}\cdot\mathbf{P}_0 &= 0 \\
ax + by + cz + d &= 0. \qquad (5.12)
\end{aligned}
$$

仅当点 P 位于平面上时，方程 (5.12) 才成立。但是，当点 P 不在平面上时会发生什么情况？在这种情况下，平面方程 (ax + by + cz ，等于 n · P) 的左边表示该点距离平面有多远。该表达式计算了 P 到原点的距离与平面到原点的距离之差。换句话说，方程 (5.12) 的左边给出了点与平面之间的垂直距离 h ！这只是 5.2.4.7 节中方程 (5.2) 的另一种写法。

$$
\begin{aligned}
h &= (\mathbf{P} - \mathbf{P}_0)\cdot\mathbf{n}; \\
h &= ax + by + cz + d. \qquad (5.13)
\end{aligned}
$$

平面实际上可以像球体一样被压缩成一个四元素向量。为此，我们观察到，要唯一地描述一个平面，我们只需要法向量 n = [abc] 和到原点的距离 d 。四元素向量 L = [nd] = [abcd] 是一种紧凑且方便的表示和存储平面的方式。注意，当 P 写成齐次坐标系，w = 1 时，方程 (L · P) = 0 是 (n · P) = − d 的另一种写法。对于位于平面 L 上的所有点 P ，这些方程都成立。

以四元素向量形式定义的平面可以轻松地从一个坐标空间变换到另一个坐标空间。给定一个矩阵 MA → B，它将点和（非法向量）从空间 A 变换到空间 B。我们已经知道，要变换法向量（例如平面的 n 向量），需要使用该矩阵的逆转置 (M − 1 A → B) T 。因此，如果知道将矩阵的逆转置应用于四元素平面向量 L，实际上可以正确地将该平面从空间 A 变换到空间 B，这并不奇怪。我们不会在这里进一步推导或证明这个结果，但在 [32] 的 4.2.3 节中，对这个小“技巧”□□的工作原理进行了详尽的解释。

Another way of looking at this is that the plane equation and the point-normal form are really just two ways of writing the same equation. Imagine testing whether or not an arbitrary point $\mathbf{P} = \begin{bmatrix} x & y & z \end{bmatrix}$ lies on the plane. To do this, we find the signed distance from point $\mathbf{P}$ to the origin along the normal $\mathbf{n} = \begin{bmatrix} a & b & c \end{bmatrix}$, and if this signed distance is equal to the signed distance $d = -\mathbf{n}\cdot\mathbf{P}_0$ from the plane from the origin, then $\mathbf{P}$ must lie on the plane. So let's set them equal and expand some terms:

$$
\begin{aligned}
&\text{(signed distance } \mathbf{P} \text{ to origin)} = \text{(signed distance plane to origin)} \\
\mathbf{n}\cdot\mathbf{P} &= \mathbf{n}\cdot\mathbf{P}_0 \\
\mathbf{n}\cdot\mathbf{P} - \mathbf{n}\cdot\mathbf{P}_0 &= 0 \\
ax + by + cz - \mathbf{n}\cdot\mathbf{P}_0 &= 0 \\
ax + by + cz + d &= 0. \qquad (5.12)
\end{aligned}
$$

Equation (5.12) only holds when the point $\mathbf{P}$ lies on the plane. But what happens when the point $\mathbf{P}$ does *not* lie on the plane? In this case, the left-hand side of the plane equation ($ax + by + cz$, which is equal to $\mathbf{n}\cdot\mathbf{P}$) tells how far "off" the point is from being on the plane. This expression calculates the difference between the distance from $\mathbf{P}$ to the origin and the distance from the plane to the origin. In other words, the left-hand side of Equation (5.12) gives us the perpendicular distance h between the point and the plane! This is just another way to write Equation (5.2) from Section 5.2.4.7.

$$
\begin{aligned}
h &= (\mathbf{P} - \mathbf{P}_0)\cdot\mathbf{n}; \\
h &= ax + by + cz + d. \qquad (5.13)
\end{aligned}
$$

A plane can actually be packed into a four-element vector, much like a sphere can. To do so, we observe that to describe a plane uniquely, we need only the normal vector $\mathbf{n} = \begin{bmatrix} a & b & c \end{bmatrix}$ and the distance from the origin d. The four-element vector $\mathbf{L} = \begin{bmatrix} \mathbf{n} & d \end{bmatrix} = \begin{bmatrix} a & b & c & d \end{bmatrix}$ is a compact and convenient way to represent and store a plane in memory. Note that when $\mathbf{P}$ is written in homogeneous coordinates with $w = 1$, the equation $(\mathbf{L}\cdot\mathbf{P}) = 0$ is yet another way of writing $(\mathbf{n}\cdot\mathbf{P}) = -d$. These equations are satisfied for all points $\mathbf{P}$ that lie on the plane $\mathbf{L}$.

Planes defined in four-element vector form can be easily transformed from one coordinate space to another. Given a matrix $\mathbf{M}_{A\to B}$ that transforms points and (non-normal) vectors from space A to space B, we already know that to transform a *normal vector* such as the plane's $\mathbf{n}$ vector, we need to use the inverse transpose of that matrix, $(\mathbf{M}_{A\to B}^{-1})^{\mathrm{T}}$. So it shouldn't be a big surprise to learn that applying the inverse transpose of a matrix to a four-element plane vector $\mathbf{L}$ will, in fact, correctly transform that plane from space A to space B. We won't derive or prove this result any further here, but a thorough explanation of why this little "trick" works is provided in Section 4.2.3 of [32].

5.6.4 轴对齐边界框（AABB）

轴对齐边界框 (AABB) 是一个三维长方体，其六个矩形面与特定坐标系的相互正交的轴对齐。因此，AABB 可以用一个六元素向量表示，该向量包含三个主轴 [x min , y min , z min , x max , y max , z max] 上的最小和最大坐标，或者两个点 P min 和 P max 。

这种简单的表示方法提供了一种非常方便且廉价的方法来测试点 P 是否位于给定的 AABB 内部或外部。我们只需测试以下所有条件是否成立：

$$
\begin{aligned}
P_x \geq x_{\text{min}} \text{ and } P_x \leq x_{\text{max}} \text{ and} \\
P_y \geq y_{\text{min}} \text{ and } P_y \leq y_{\text{max}} \text{ and} \\
P_z \geq z_{\text{min}} \text{ and } P_z \leq z_{\text{max}}.
\end{aligned}
$$

由于相交测试非常快，AABB 通常用作“早期”碰撞检查；如果两个物体的 AABB 不相交，则无需进行更详细（且更昂贵）的碰撞测试。

5.6.5 定向边界框（OBB）

定向边界框 (OBB) 是一个经过定向的长方体，其方向与它所包围的对象以某种逻辑方式对齐。通常，OBB 与对象的局部空间轴对齐。因此，它在局部空间中的作用类似于 AABB，尽管它不一定与世界空间轴对齐。

存在各种用于测试某个点是否位于 OBB 内的技术，但一种常见的方法是将点转换到 OBB 的“对齐”坐标系中，然后使用如上所述的 AABB 相交测试。

5.6.6 鞭子

如图 5.30 所示，视锥体是由六个平面组成的，它们构成了一个截锥形。视锥体在 3D 渲染中很常见，因为当从虚拟摄像机的视角通过透视投影进行渲染时，它们可以方便地定义 3D 世界的可视区域。其中四个平面界定了屏幕空间的边缘，而另外两个平面则代表近裁剪平面和远裁剪平面（即，它们定义了任何可见点的最小和最大 z 坐标）。

截头体的一个方便的表示是作为六个平面的阵列，每个平面以点法线形式表示（即每个平面一个点和一个法线向量）。

图 5.30 一个片段。

5.6.4 Axis-Aligned Bounding Boxes (AABB)

An axis-aligned bounding box (AABB) is a 3D *cuboid* whose six rectangular faces are aligned with a particular coordinate frame's mutually orthogonal axes. As such, an AABB can be represented by a six-element vector containing the minimum and maximum coordinates along each of the 3 principal axes, $[x_{\text{min}}, y_{\text{min}}, z_{\text{min}}, x_{\text{max}}, y_{\text{max}}, z_{\text{max}}]$, or two points $\mathbf{P}_{\text{min}}$ and $\mathbf{P}_{\text{max}}$.

This simple representation allows for a particularly convenient and inexpensive method of testing whether a point $\mathbf{P}$ is inside or outside any given AABB. We simply test if all of the following conditions are true:

$$
\begin{aligned}
P_x \geq x_{\text{min}} \text{ and } P_x \leq x_{\text{max}} \text{ and} \\
P_y \geq y_{\text{min}} \text{ and } P_y \leq y_{\text{max}} \text{ and} \\
P_z \geq z_{\text{min}} \text{ and } P_z \leq z_{\text{max}}.
\end{aligned}
$$

Because intersection tests are so speedy, AABBs are often used as an "early out" collision check; if the AABBs of two objects do not intersect, then there is no need to do a more detailed (and more expensive) collision test.

5.6.5 Oriented Bounding Boxes (OBB)

An oriented bounding box (OBB) is a *cuboid* that has been oriented so as to align in some logical way with the object it bounds. Usually an OBB aligns with the local-space axes of the object. Hence, it acts like an AABB in local space, although it may not necessarily align with the world-space axes.

Various techniques exist for testing whether or not a point lies within an OBB, but one common approach is to transform the point into the OBB's "aligned" coordinate system and then use an AABB intersection test as presented above.

5.6.6 Frusta

As shown in Figure 5.30, a *frustum* is a group of six planes that define a *truncated pyramid* shape. Frusta are commonplace in 3D rendering because they conveniently define the viewable region of the 3D world when rendered via a perspective projection from the point of view of a virtual camera. Four of the planes bound the edges of the screen space, while the other two planes represent the the near and far clipping planes (i.e., they define the minimum and maximum z coordinates possible for any visible point).

One convenient representation of a frustum is as an array of six planes, each of which is represented in point-normal form (i.e., one point and one normal vector per plane).

Figure 5.30. A frustum.

判断一个点是否位于截头体内部有点复杂，但基本思路是使用点积来确定该点位于每个平面的正面还是背面。如果它位于所有六个平面内，则它位于截头体内部。

一个有用的技巧是将相机的透视投影应用于被测试的世界空间点，从而对其进行变换。这会将点从世界空间转换到一个称为 齐次裁剪空间 的空间。在这个空间中，视锥体只是一个轴对齐长方体 (AABB)。这允许执行更简单的输入/输出测试。

5.6.7 凸多面体区域

凸多面体区域由任意一组平面定义，所有平面的法线均指向内（或外）。判断一个点位于这些平面定义的体积之内还是之外的方法相对简单；它类似于视锥体测试，但可能包含更多平面。凸区域对于在游戏中实现任意形状的触发区域非常有用。许多引擎都采用了这种技术；例如，Quake 引擎中随处可见的画笔就是用这种方式由平面包围的体积。

5.7 随机数生成

随机数在游戏引擎中无处不在，因此我们有必要简要了解一下两种最常见的随机数生成器 (RNG)：线性同余生成器和梅森旋转算法。重要的是要认识到，随机数生成器实际上并不生成随机数——它们只是产生一个复杂但完全确定的预定义值序列。因此，我们将它们产生的序列称为伪随机数，从技术上讲，我们实际上应该称它们为“伪随机数生成器”(PRNG)。好生成器与坏生成器的区别在于数字序列在重复之前的长度（其周期），以及序列在各种众所周知的随机性测试下的保持情况。

5.7.1 线性同余生成器

线性同余生成器是一种生成伪随机数序列的非常快速简便的方法。根据平台的不同，C 标准库的 rand() 函数有时会使用此算法。但是，您的情况可能会有所不同，因此不要指望 rand() 会基于任何特定算法。如果您想确保万无一失，最好实现自己的随机数生成器。

Testing whether a point lies inside a frustum is a bit involved, but the basic idea is to use dot products to determine whether the point lies on the front or back side of each plane. If it lies inside all six planes, it is inside the frustum.

A helpful trick is to transform the world-space point being tested by applying the camera's perspective projection to it. This takes the point from world space into a space known as *homogeneous clip space*. In this space, the frustum is just an axis-aligned cuboid (AABB). This permits much simpler in/out tests to be performed.

5.6.7 Convex Polyhedral Regions

A *convex polyhedral region* is defined by an arbitrary set of planes, all with normals pointing inward (or outward). The test for whether a point lies inside or outside the volume defined by the planes is relatively straightforward; it is similar to a frustum test, but with possibly more planes. Convex regions are very useful for implementing arbitrarily shaped trigger regions in games. Many engines employ this technique; for example, the Quake engine's ubiquitous *brushes* are just volumes bounded by planes in exactly this way.

5.7 Random Number Generation

Random numbers are ubiquitous in game engines, so it behooves us to have a brief look at the two most common random number generators (RNG), the linear congruential generator and the Mersenne Twister. It's important to realize that random number generators don't actually generate random numbers—they merely produce a complex, but totally deterministic, predefined sequence of values. For this reason, we call the sequences they produce *pseudorandom*, and technically speaking we should really call them "pseudorandom number generators" (PRNG). What differentiates a good generator from a bad one is how long the sequence of numbers is before it repeats (its *period*), and how well the sequences hold up under various well-known randomness tests.

5.7.1 Linear Congruential Generators

Linear congruential generators are a very fast and simple way to generate a sequence of pseudorandom numbers. Depending on the platform, this algorithm is sometimes used in the C standard library's `rand()` function. However, your mileage may vary, so don't count on `rand()` being based on any particular algorithm. If you want to be sure, you'll be better off implementing your own random number generator.

线性同余算法在《C 语言数值方法》一书中有详细解释，因此我在这里就不再赘述了。

我要说的是，这个随机数生成器并不能生成特别高质量的伪随机序列。给定相同的初始种子值，生成的序列总是完全相同。生成的数字不符合许多被广泛接受的理想标准，例如周期长、低位和高位周期长度相近，以及生成的值之间缺乏序列或空间相关性。

5.7.2 梅森旋转算法

梅森旋转伪随机数生成器算法是专门为改进线性同余算法的各种问题而设计的。维基百科对该算法的优点进行了如下描述：

1. 它被设计为具有 2 19937 − 1 的巨大周期（算法的创建者证明了这一特性）。在实践中，几乎没有理由使用更大的周期，因为大多数应用程序不需要 2 19937 个唯一组合（2 19937 ≈ 4.3 × 10 6001）。

2. 它具有非常高的维度均匀分布阶数。注意，这意味着默认情况下，输出序列中连续值之间的序列相关性可以忽略不计。

3. 它通过了多项统计随机性测试，包括严格的 Diehard 测试。

4.速度快。

网络上有各种Twister的实现，其中一种特别酷炫，它使用SIMD矢量指令来进一步提升速度，称为SFMT（面向SIMD的快速梅森旋转算法）。SFMT可以从http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html下载。

5.7.3 Mother-of-All、Xorshift 和 KISS99

1994年，计算机科学家兼数学家乔治·马萨利亚（George Marsaglia）发表了一种伪随机数生成算法，该算法比梅森旋转算法（Mersenne Twister algorithm）实现起来简单得多，运行速度也更快。马萨利亚以开发Diehard随机性测试系统（http://www.stat.fsu.edu/pub/diehard）而闻名。他声称，该算法可以生成一个序列

The linear congruential algorithm is explained in detail in the book *Numerical Recipes in C*, so I won't go into the details of it here.

What I will say is that this random number generator does not produce particularly high-quality pseudorandom sequences. Given the same initial seed value, the sequence is always exactly the same. The numbers produced do not meet many of the criteria widely accepted as desirable, such as a long period, low- and high-order bits that have similarly long periods, and absence of sequential or spatial correlation between the generated values.

5.7.2 Mersenne Twister

The Mersenne Twister pseudorandom number generator algorithm was designed specifically to improve upon the various problems of the linear congruential algorithm. Wikipedia provides the following description of the benefits of the algorithm:

1. It was designed to have a colossal period of $2^{19937} - 1$ (the creators of the algorithm proved this property). In practice, there is little reason to use larger ones, as most applications do not require 2^{19937} unique combinations ($2^{19937} \approx 4.3 \times 10^{6001}$).
2. It has a very high order of dimensional equidistribution. Note that this means, by default, that there is negligible serial correlation between successive values in the output sequence.
3. It passes numerous tests for statistical randomness, including the stringent Diehard tests.
4. It is fast.

Various implementations of the Twister are available on the web, including a particularly cool one that uses SIMD vector instructions for an extra speed boost, called *SFMT* (SIMD-oriented fast Mersenne Twister). SFMT can be downloaded from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html.

5.7.3 Mother-of-All, Xorshift and KISS99

In 1994, George Marsaglia, a computer scientist and mathematician best known for developing the Diehard battery of tests of randomness (http://www.stat.fsu.edu/pub/diehard), published a pseudorandom number generation algorithm that is much simpler to implement and runs faster than the Mersenne Twister algorithm. He claimed that it could produce a sequence

一个32位伪随机数，非重复周期为2250。它通过了所有Diehard测试，至今仍是高速应用中最佳的伪随机数生成器之一。他将自己的算法称为“伪随机数生成器之母”，因为在他看来，这是人们唯一需要的随机数生成器。

后来，Marsaglia 又发表了另一个生成器，名为 Xorshift，其随机性介于 Mersenne 和 Mother-of-All 之间，但运行速度比 Mother 稍快。

Marsaglia 还开发了一系列随机数生成器，统称为 KISS（Keep It Simple Stupid）。KISS99 算法是一个受欢迎的选择，因为它的周期较大（2123），并且通过了 TestU01 测试套件（https://bit.ly/2r5FmSP）中的所有测试。

您可以在 http://en.wikipedia.org/wiki/George_Marsaglia 上阅读有关 George Marsaglia 的文章，也可以在 ftp://ftp.forth.org/pub/C/mother.c 和 http://www.agner.org/random 上阅读有关 Mother-of-All 生成器的文章。您还可以在 http://www.jstatsoft.org/v08/i14/paper 下载 George 关于 Xorshift 的论文 PDF 版本。

5.7.4 PCG

另一个非常流行且高质量的伪随机数生成器系列是 PCG。它的工作原理是将用于状态转换的同余生成器（PCG 中的“CG”）与用于生成输出的置换函数（PCG 中的“P”）相结合。您可以在 http://www.pcg-random.org/ 上了解有关该系列 PRNG 的更多信息。

of 32-bit pseudorandom numbers with a period of non-repetition of 2^{250}. It passed all of the Diehard tests and still stands today as one of the best pseudorandom number generators for high-speed applications. He called his algorithm the *Mother of All Pseudorandom Number Generators*, because it seemed to him to be the only random number generator one would ever need.

Later, Marsaglia published another generator called Xorshift, which is between Mersenne and Mother-of-All in terms of randomness, but runs slightly faster than Mother.

Marsaglia also developed a series of random number generators that are collectively called KISS (Keep It Simple Stupid). The KISS99 algorithm is a popular choice, because it has a large period (2^{123}) and passes all tests in the TestU01 test suite (https://bit.ly/2r5FmSP).

You can read about George Marsaglia at http://en.wikipedia.org/wiki/George_Marsaglia, and about the Mother-of-All generator at ftp://ftp.forth.org/pub/C/mother.c and at http://www.agner.org/random. You can download a PDF of George's paper on Xorshift at http://www.jstatsoft.org/v08/i14/paper.

5.7.4 PCG

Another very popular and high-quality family of pseudorandom number generators is called PCG. It works by combining a congruential generator for its state transitions (the "CG" in PCG) with permutation functions to generate its output (the "P" in PCG). You can read more about this family of PRNGs at http://www.pcg-random.org/.

第二部分

低级
发动机系统

Part II

Low-Level
Engine Systems

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

6 发动机支持系统

每个游戏引擎都需要一些底层支持系统来管理一些日常但至关重要的任务，例如启动和关闭引擎、配置引擎和游戏功能、管理引擎的内存使用、处理文件系统访问、提供对游戏中使用的各种异构资源类型（网格、纹理、动画、音频等）的访问，以及提供供游戏开发团队使用的调试工具。本章将重点介绍大多数游戏引擎中存在的底层支持系统。在接下来的章节中，我们将探讨一些较大的核心系统，包括资源管理、人机界面设备和游戏内调试工具。

6.1 子系统启动和关闭

游戏引擎是一个复杂的软件，由许多相互作用的子系统组成。引擎首次启动时，每个子系统都必须按照特定顺序进行配置和初始化。子系统之间的相互依赖关系隐式地定义了它们的启动顺序——例如，如果子系统 B 依赖于子系统 A，则需要先启动 A，然后才能初始化 B。关闭引擎通常以相反的顺序进行，因此 B 会先关闭，然后再关闭 A。

6 Engine Support Systems

Every game engine requires some low-level support systems that manage mundane but crucial tasks, such as starting up and shutting down the engine, configuring engine and game features, managing the engine's memory usage, handling access to file system(s), providing access to the wide range of heterogeneous asset types used by the game (meshes, textures, animations, audio, etc.), and providing debugging tools for use by the game development team. This chapter will focus on the lowest-level support systems found in most game engines. In the chapters that follow, we will explore some of the larger core systems, including resource management, human interface devices and in-game debugging tools.

6.1 Subsystem Start-Up and Shut-Down

A game engine is a complex piece of software consisting of many interacting subsystems. When the engine first starts up, each subsystem must be configured and initialized in a specific order. Interdependencies between subsystems implicitly define the order in which they must be started—i.e., if subsystem B depends on subsystem A, then A will need to be started up before B can be initialized. Shut-down typically occurs in the reverse order, so B would shut down first, followed by A.

6.1.1 C++ 静态初始化顺序（或缺乏顺序）

由于大多数现代游戏引擎使用的编程语言是 C++，我们应该简要考虑一下是否可以利用 C++ 的原生启动和关闭语义来启动和关闭引擎的子系统。在 C++ 中，全局对象和静态对象是在调用程序入口点（main()，或 Windows 下的 WinMain()）之前构造的。但是，这些构造函数的调用顺序完全不可预测。全局和静态类实例的析构函数在 main()（或 WinMain()）返回后调用，并且它们的调用顺序也是不可预测的。显然，这种行为对于初始化和关闭游戏引擎的子系统是不可取的，实际上对于任何全局对象之间存在相互依赖的软件系统都是不可取的。

这多少有点可惜，因为实现主要子系统（例如构成游戏引擎的子系统）的常见设计模式是为每个子系统定义一个单例类（通常称为管理器）。如果 C++ 允许我们更好地控制全局和静态类实例的构造和销毁顺序，我们就可以把单例实例定义为全局变量，而无需动态内存分配。例如，我们可以这样写：

```
class RenderManager
{
public:
    RenderManager()
    {
        // start up the manager...
    }

    ~RenderManager()
    {
        // shut down the manager...
    }

    // ...
};

// singleton instance
static RenderManager gRenderManager;
```

唉，由于无法直接控制构造和销毁顺序，这种方法行不通。

6.1.1 C++ Static Initialization Order (or Lack Thereof)

Since the programming language used in most modern game engines is C++, we should briefly consider whether C++'s native start-up and shut-down semantics can be leveraged in order to start up and shut down our engine's subsystems. In C++, global and static objects are constructed before the program's entry point (`main()`, or `WinMain()` under Windows) is called. However, these constructors are called in a totally unpredictable order. The destructors of global and static class instances are called after `main()` (or `WinMain()`) returns, and once again they are called in an unpredictable order. Clearly this behavior is not desirable for initializing and shutting down the subsystems of a game engine, or indeed any software system that has interdependencies between its global objects.

This is somewhat unfortunate, because a common design pattern for implementing major subsystems such as the ones that make up a game engine is to define a *singleton class* (often called a *manager*) for each subsystem. If C++ gave us more control over the order in which global and static class instances were constructed and destroyed, we could define our singleton instances as globals, without the need for dynamic memory allocation. For example, we could write:

```
class RenderManager
{
public:
    RenderManager()
    {
        // start up the manager...
    }

    ~RenderManager()
    {
        // shut down the manager...
    }

    // ...
};

// singleton instance
static RenderManager gRenderManager;
```

Alas, with no way to directly control construction and destruction order, this approach won't work.

6.1.1.1 按需构建

这里我们可以利用一个 C++“技巧”。在函数内声明的静态变量不会在 main() 调用之前构造，而是在第一次调用该函数时构造。因此，如果我们的全局单例是函数静态的，我们就可以控制全局单例的构造顺序。

```
class RenderManager
{
public:
    // Get the one and only instance.
    static RenderManager& get()
    {
        // This function-static will be constructed on the
        // first call to this function.
        static RenderManager sSingleton;
        return sSingleton;
    }

    RenderManager()
    {
        // Start up other managers we depend on, by
        // calling their get() functions first...
        VideoManager::get();
        TextureManager::get();

        // Now start up the render manager.
        // ...
    }

    ~RenderManager()
    {
        // Shut down the manager.
        // ...
    }
};
```

您会发现许多软件工程教科书都建议采用这种设计或涉及单例动态分配的变体，如下所示。

```
static RenderManager& get()
{
    static RenderManager* gpSingleton = nullptr;
    if (gpSingleton == nullptr)
    {
        gpSingleton = new RenderManager;
```


6.1.1.1 Construct On Demand

There is one C++ “trick” we can leverage here. A static variable that is declared within a function will not be constructed before `main()` is called, but rather on the first invocation of that function. So if our global singleton is function-static, we can control the order of construction for our global singletons.

```
class RenderManager
{
public:
    // Get the one and only instance.
    static RenderManager& get()
    {
        // This function-static will be constructed on the
        // first call to this function.
        static RenderManager sSingleton;
        return sSingleton;
    }

    RenderManager()
    {
        // Start up other managers we depend on, by
        // calling their get() functions first...
        VideoManager::get();
        TextureManager::get();

        // Now start up the render manager.
        // ...
    }

    ~RenderManager()
    {
        // Shut down the manager.
        // ...
    }
};
```

You’ll find that many software engineering textbooks suggest this design or a variant that involves dynamic allocation of the singleton as shown below.

```
static RenderManager& get()
{
    static RenderManager* gpSingleton = nullptr;
    if (gpSingleton == nullptr)
    {
        gpSingleton = new RenderManager;
```

```
        }
        ASSERT(gpSingleton);
        return *gpSingleton;
    }
```

不幸的是，这仍然让我们无法控制销毁顺序。C++ 可能会在调用 RenderManager 的析构函数之前销毁 RenderManager 关闭过程中所依赖的某个管理器。此外，很难准确预测 RenderManager 单例的构造时间，因为构造将在第一次调用 RenderManager::get() 时发生——谁知道那是什么时候呢？而且，使用该类的程序员可能并不指望看似无害的 get() 函数会做一些昂贵的事情，比如分配和初始化一个重量级的单例。这是一种不可预测且危险的设计。因此，我们被提示采取一种更直接的方法来获得更大的控制权。

6.1.2 一种简单有效的方法

假设我们想坚持使用单例管理器来管理子系统。在这种情况下，最简单的“强力”方法是为每个单例管理器类定义显式的启动和关闭函数。这些函数取代了构造函数和析构函数，实际上，我们应该让构造函数和析构函数完全不执行任何操作。这样，就可以在 main() 函数中（或从某个管理整个引擎的总体单例对象中）按照所需的顺序显式调用启动和关闭函数。例如：

```
class RenderManager
{
public:
    RenderManager()
    {
        // do nothing
    }

    ~RenderManager()
    {
        // do nothing
    }

    void startUp()
    {
        // start up the manager...
```



```
        }
        ASSERT(gpSingleton);
        return *gpSingleton;
    }
```

Unfortunately, this still gives us no way to control destruction order. It is possible that C++ will destroy one of the managers upon which the `RenderManager` depends for its shut-down procedure, prior to the `RenderManager`'s destructor being called. In addition, it's difficult to predict exactly when the `RenderManager` singleton will be constructed, because the construction will happen on the first call to `RenderManager::get()`—and who knows when that might be? Moreover, the programmers using the class may not be expecting an innocuous-looking `get()` function to do something expensive, like allocating and initializing a heavyweight singleton. This is an unpredictable and dangerous design. Therefore, we are prompted to resort to a more direct approach that gives us greater control.

6.1.2 A Simple Approach That Works

Let's presume that we want to stick with the idea of singleton managers for our subsystems. In this case, the simplest "brute-force" approach is to define explicit start-up and shut-down functions for each singleton manager class. These functions take the place of the constructor and destructor, and in fact we should arrange for the constructor and destructor to do *absolutely nothing*. That way, the start-up and shut-down functions can be explicitly called *in the required order* from within `main()` (or from some overarching singleton object that manages the engine as a whole). For example:

```
class RenderManager
{
public:
    RenderManager()
    {
        // do nothing
    }

    ~RenderManager()
    {
        // do nothing
    }

    void startUp()
    {
        // start up the manager...
```

```
    }

    void shutDown()
    {
        // shut down the manager...
    }

    // ...
};

class PhysicsManager    { /* similar... */ };

class AnimationManager  { /* similar... */ };

class MemoryManager     { /* similar... */ };

class FileSystemManager { /* similar... */ };

// ...

RenderManager           gRenderManager;
PhysicsManager          gPhysicsManager;
AnimationManager        gAnimationManager;
TextureManager          gTextureManager;
VideoManager            gVideoManager;
MemoryManager           gMemoryManager;
FileSystemManager       gFileSystemManager;
// ...

int main(int argc, const char* argv)
{
    // Start up engine systems in the correct order.
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ...

    // Run the game.
    gSimulationManager.run();

    // Shut everything down, in reverse order.
    // ...
    gPhysicsManager.shutDown();
```



```
    }

    void shutDown()
    {
        // shut down the manager...
    }

    // ...
};

class PhysicsManager    { /* similar... */ };

class AnimationManager  { /* similar... */ };

class MemoryManager     { /* similar... */ };

class FileSystemManager { /* similar... */ };

// ...

RenderManager           gRenderManager;
PhysicsManager          gPhysicsManager;
AnimationManager        gAnimationManager;
TextureManager          gTextureManager;
VideoManager            gVideoManager;
MemoryManager           gMemoryManager;
FileSystemManager       gFileSystemManager;
// ...

int main(int argc, const char* argv)
{
    // Start up engine systems in the correct order.
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ...

    // Run the game.
    gSimulationManager.run();

    // Shut everything down, in reverse order.
    // ...
    gPhysicsManager.shutDown();
```

```
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

有“更优雅”的方法可以实现这一点。例如，你可以让每个管理器将自己注册到一个全局优先级队列中，然后遍历该队列，按正确的顺序启动所有管理器。你可以定义管理器到管理器的依赖关系图，方法是让每个管理器明确列出它所依赖的其他管理器，然后编写一些代码来计算考虑到它们相互依赖关系的最佳启动顺序。你可以使用上面概述的按需构建方法。根据我的经验，蛮力法总是胜出，原因如下：

- 简单且易于实施。
- 非常清晰。只需查看代码，即可立即看到并理解启动顺序。
- 易于调试和维护。如果某些操作启动得不够早，或者启动得太早，只需移动一行代码即可。

手动强制启动和关闭的一个小缺点是，你可能会意外地以与启动顺序不完全相反的顺序关闭系统。但我不会为此失眠。只要你能成功启动和关闭引擎的子系统，就万事大吉了。

6.1.3 一些来自真实引擎的例子

让我们简单看一下从真实游戏引擎中截取的一些引擎启动和关闭的示例。

6.1.3.1 食人魔

OGRE 的作者承认它本身是一个渲染引擎，而非游戏引擎。但它必然提供了许多成熟游戏引擎所具备的底层功能，包括简洁优雅的启动和关闭机制。OGRE 中的所有内容都由单例对象 Ogre::Root 控制。它包含指向 OGRE 中所有其他子系统的指针，并管理它们的创建和销毁。这使得程序员可以非常轻松地启动 OGRE——只需创建一个 Ogre::Root 实例即可。


```
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

There are “more elegant” ways to accomplish this. For example, you could have each manager register itself into a global priority queue and then walk this queue to start up all the managers in the proper order. You could define the manager-to-manager dependency graph by having each manager explicitly list the other managers upon which it depends and then write some code to calculate the optimal start-up order given their interdependencies. You could use the construct-on-demand approach outlined above. In my experience, the brute-force approach always wins out, because of the following:

- It’s simple and easy to implement.
- It’s explicit. You can see and understand the start-up order immediately by just looking at the code.
- It’s easy to debug and maintain. If something isn’t starting early enough, or is starting too early, you can just move one line of code.

One minor disadvantage to the brute-force manual start-up and shut-down method is that you might accidentally shut things down in an order that isn’t strictly the reverse of the start-up order. But I wouldn’t lose any sleep over it. As long as you can start up and shut down your engine’s subsystems successfully, you’re golden.

6.1.3 Some Examples from Real Engines

Let’s take a brief look at some examples of engine start-up and shut-down taken from real game engines.

6.1.3.1 OGRE

OGRE is by its authors’ admission a rendering engine, not a game engine per se. But by necessity it provides many of the low-level features found in full-fledged game engines, including a simple and elegant start-up and shut-down mechanism. Everything in OGRE is controlled by the singleton object `Ogre::Root`. It contains pointers to every other subsystem in OGRE and manages their creation and destruction. This makes it very easy for a programmer to start up OGRE—just `new` an instance of `Ogre::Root` and you’re done.

以下是 OGRE 源代码的几个摘录，以便我们了解它的作用：

OgreRoot.h

```
class _OgreExport Root : public Singleton<Root>
{
    // <some code omitted...>

    // Singletons
    LogManager* mLogManager;
    ControllerManager* mControllerManager;
    SceneManagerEnumerator* mSceneManagerEnum;
    SceneManager* mCurrentSceneManager;
    DynLibManager* mDynLibManager;
    ArchiveManager* mArchiveManager;
    MaterialManager* mMaterialManager;
    MeshManager* mMeshManager;
    ParticleSystemManager* mParticleManager;
    SkeletonManager* mSkeletonManager;
    OverlayElementFactory* mPanelFactory;
    OverlayElementFactory* mBorderPanelFactory;
    OverlayElementFactory* mTextAreaFactory;
    OverlayManager* mOverlayManager;
    FontManager* mFontManager;
    ArchiveFactory *mZipArchiveFactory;
    ArchiveFactory *mFileSystemArchiveFactory;
    ResourceGroupManager* mResourceGroupManager;
    ResourceBackgroundQueue* mResourceBackgroundQueue;
    ShadowTextureManager* mShadowTextureManager;

    // etc.
};
```

OgreRoot.cpp

```
Root::Root(const String& pluginFileName,
           const String& configFileName,
           const String& logFileName) :
    mLogManager(0),
    mCurrentFrame(0),
    mFrameSmoothingTime(0.0f),
    mNextMovableObjectTypeFlag(1),
    mIsInitialised(false)
{
    // superclass will do singleton checking
    String msg;

    // Init
    mActiveRenderer = 0;
```


Here are a few excerpts from the OGRE source code so we can see what it's doing:

OgreRoot.h

```
class _OgreExport Root : public Singleton<Root>
{
    // <some code omitted...>

    // Singletons
    LogManager* mLogManager;
    ControllerManager* mControllerManager;
    SceneManagerEnumerator* mSceneManagerEnum;
    SceneManager* mCurrentSceneManager;
    DynLibManager* mDynLibManager;
    ArchiveManager* mArchiveManager;
    MaterialManager* mMaterialManager;
    MeshManager* mMeshManager;
    ParticleSystemManager* mParticleManager;
    SkeletonManager* mSkeletonManager;
    OverlayElementFactory* mPanelFactory;
    OverlayElementFactory* mBorderPanelFactory;
    OverlayElementFactory* mTextAreaFactory;
    OverlayManager* mOverlayManager;
    FontManager* mFontManager;
    ArchiveFactory *mZipArchiveFactory;
    ArchiveFactory *mFileSystemArchiveFactory;
    ResourceGroupManager* mResourceGroupManager;
    ResourceBackgroundQueue* mResourceBackgroundQueue;
    ShadowTextureManager* mShadowTextureManager;

    // etc.
};
```

OgreRoot.cpp

```
Root::Root(const String& pluginFileName,
           const String& configFileName,
           const String& logFileName) :
    mLogManager(0),
    mCurrentFrame(0),
    mFrameSmoothingTime(0.0f),
    mNextMovableObjectTypeFlag(1),
    mIsInitialised(false)
{
    // superclass will do singleton checking
    String msg;

    // Init
    mActiveRenderer = 0;
```

```
        mVersion
            = StringConverter::toString(OGRE_VERSION_MAJOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_MINOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_PATCH)
            + OGRE_VERSION_SUFFIX + " "
            + "(" + OGRE_VERSION_NAME + ")";
        mConfigFileName = configFileName;

        // create log manager and default log file if there
        // is no log manager yet
        if(LogManager::getSingletonPtr() == 0)
        {
            mLogManager = new LogManager();
            mLogManager->createLog(logFileName, true, true);
        }

        // dynamic library manager
        mDynLibManager = new DynLibManager();
        mArchiveManager = new ArchiveManager();

        // ResourceGroupManager
        mResourceGroupManager = new ResourceGroupManager();

        // ResourceBackgroundQueue
        mResourceBackgroundQueue
            = new ResourceBackgroundQueue();

        // and so on...
    }
```

OGRE 提供了一个模板化的 Ogre::Singleton 基类，其所有单例（管理器）类都继承自该基类。如果查看它的实现，你会发现 Ogre::Singleton 并没有使用延迟构造，而是依赖于 Ogre::Root 来显式地创建每个单例。正如我们上面所讨论的，这样做是为了确保单例的创建和销毁遵循明确定义的顺序。

6.1.3.2 顽皮狗的《神秘海域》和《最后生还者》系列

顽皮狗公司为其《神秘海域》和《最后生还者》系列游戏创建的引擎也使用了类似的显式技术来启动其子系统。通过查看以下代码，您会注意到，引擎启动并不总是简单的分配单例实例序列。各种各样的操作系统服务、第三方库等等都必须启动。


```
        mVersion
            = StringConverter::toString(OGRE_VERSION_MAJOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_MINOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_PATCH)
            + OGRE_VERSION_SUFFIX + " "
            + "(" + OGRE_VERSION_NAME + ")";
        mConfigFileName = configFileName;

        // create log manager and default log file if there
        // is no log manager yet
        if(LogManager::getSingletonPtr() == 0)
        {
            mLogManager = new LogManager();
            mLogManager->createLog(logFileName, true, true);
        }

        // dynamic library manager
        mDynLibManager = new DynLibManager();
        mArchiveManager = new ArchiveManager();

        // ResourceGroupManager
        mResourceGroupManager = new ResourceGroupManager();

        // ResourceBackgroundQueue
        mResourceBackgroundQueue
            = new ResourceBackgroundQueue();

        // and so on...
    }
```

OGRE provides a templated `Ogre::Singleton` base class from which all of its singleton (manager) classes derive. If you look at its implementation, you'll see that `Ogre::Singleton` does not use deferred construction but instead relies on `Ogre::Root` to explicitly `new` each singleton. As we discussed above, this is done to ensure that the singletons are created and destroyed in a well-defined order.

6.1.3.2 Naughty Dog's *Uncharted* and *The Last of Us* Series

The engine created by Naughty Dog, Inc. for its *Uncharted* and *The Last of Us* series of games uses a similar explicit technique for starting up its subsystems. You'll notice by looking at the following code that engine start-up is not always a simple sequence of allocating singleton instances. A wide range of operating system services, third-party libraries and so on must all be started

在引擎初始化期间启动。此外，尽可能避免动态内存分配，因此许多单例都是静态分配的对象（例如 g_fileSystem 、 g_languageMgr 等）。虽然这并不总是很美观，但总能完成工作。

```
Err BigInit()
{
    init_exception_handler();

    U8* pPhysicsHeap = new(kAllocGlobal, kAlign16)
        U8[ALLOCATION_GLOBAL_PHYS_HEAP];
    PhysicsAllocatorInit(pPhysicsHeap,
        ALLOCATION_GLOBAL_PHYS_HEAP);

    g_textDb.Init();
    g_textSubDb.Init();
    g_spuMgr.Init();

    g_drawScript.InitPlatform();

    PlatformUpdate();

    thread_t init_thr;
    thread_create(&init_thr, threadInit, 0, 30,
        64*1024, 0, "Init");

    char masterConfigFileName[256];
    snprintf(masterConfigFileName,
        sizeof(masterConfigFileName),
        MASTER_CFG_PATH);
    {
        Err err = ReadConfigFromFile(
            masterConfigFileName);
        if (err.Failed())
        {
            MsgErr("Config file not found (%s).\n",
                masterConfigFileName);
        }
    }

    memset(&g_discInfo, 0, sizeof(BootDiscInfo));
    int err1 = GetBootDiscInfo(&g_discInfo);
    Msg("GetBootDiscInfo() : 0x%x\n", err1);
    if(err1 == BOOTDISCINFO_RET_OK)
    {
        printf("titleId       : [%s]\n",
            g_discInfo.titleId);
```


up during engine initialization. Also, dynamic memory allocation is avoided wherever possible, so many of the singletons are statically allocated objects (e.g., `g_fileSystem`, `g_languageMgr`, etc.) It's not always pretty, but it gets the job done.

```
Err BigInit()
{
    init_exception_handler();

    U8* pPhysicsHeap = new(kAllocGlobal, kAlign16)
        U8[ALLOCATION_GLOBAL_PHYS_HEAP];
    PhysicsAllocatorInit(pPhysicsHeap,
        ALLOCATION_GLOBAL_PHYS_HEAP);

    g_textDb.Init();
    g_textSubDb.Init();
    g_spuMgr.Init();

    g_drawScript.InitPlatform();

    PlatformUpdate();

    thread_t init_thr;
    thread_create(&init_thr, threadInit, 0, 30,
        64*1024, 0, "Init");

    char masterConfigFileName[256];
    snprintf(masterConfigFileName,
        sizeof(masterConfigFileName),
        MASTER_CFG_PATH);
    {
        Err err = ReadConfigFromFile(
            masterConfigFileName);
        if (err.Failed())
        {
            MsgErr("Config file not found (%s).\n",
                masterConfigFileName);
        }
    }

    memset(&g_discInfo, 0, sizeof(BootDiscInfo));
    int err1 = GetBootDiscInfo(&g_discInfo);
    Msg("GetBootDiscInfo() : 0x%x\n", err1);
    if(err1 == BOOTDISCINFO_RET_OK)
    {
        printf("titleId       : [%s]\n",
            g_discInfo.titleId);
```

```
        printf("parentalLevel : [%d]\n",
            g_discInfo.parentalLevel);
    }

    g_fileSystem.Init(g_gameInfo.m_onDisc);

    g_languageMgr.Init();
    if (g_shouldQuit) return Err::kOK;

    // and so on...
```

6.2 内存管理

作为游戏开发者，我们始终致力于提高代码的运行速度。任何软件的性能不仅取决于其采用的算法或这些算法的编码效率，还取决于程序如何利用内存 (RAM)。内存通过两种方式影响性能：

1. 通过 malloc() 或 C++ 的全局操作符 new 进行动态内存分配是非常缓慢的操作。我们可以通过完全避免动态分配，或者使用自定义内存分配器来大幅降低分配成本，从而提升代码性能。

2. 在现代 CPU 上，软件的性能通常取决于其内存访问模式。正如我们将看到的，CPU 操作位于小块连续内存中的数据比将相同数据分散在大范围内存地址中更高效。即使是最高效的算法，即使经过精心编写，如果其操作的数据在内存中的布局不合理，也可能陷入瘫痪。

在本节中，我们将学习如何沿着这两个轴优化代码的内存利用率。

6.2.1 优化动态内存分配

通过 malloc() 和 free() 或 C++ 的全局 new 和 delete 运算符（也称为堆分配）进行的动态内存分配通常非常慢。高成本可以归因于两个主要因素。首先，堆分配器是一个通用工具，因此必须编写它来处理任何大小的分配，从 1 字节到 1 GB。这需要大量的管理开销，使得 malloc() 和 free() 函数本身就很慢。其次，


```
        printf("parentalLevel : [%d]\n",
            g_discInfo.parentalLevel);
    }

    g_fileSystem.Init(g_gameInfo.m_onDisc);

    g_languageMgr.Init();
    if (g_shouldQuit) return Err::kOK;

    // and so on...
```

6.2 Memory Management

As game developers, we are always trying to make our code run more quickly. The performance of any piece of software is dictated not only by the algorithms it employs, or the efficiency with which those algorithms are coded, but also by how the program *utilizes memory* (RAM). Memory affects performance in two ways:

1. *Dynamic memory allocation* via `malloc()` or C++'s global `operator new` is a very slow operation. We can improve the performance of our code by either avoiding dynamic allocation altogether or by making use of custom memory allocators that greatly reduce allocation costs.

2. On modern CPUs, the performance of a piece of software is often dominated by its *memory access patterns*. As we'll see, data that is located in small, *contiguous* blocks of memory can be operated on much more efficiently by the CPU than if that same data were to be spread out across a wide range of memory addresses. Even the most efficient algorithm, coded with the utmost care, can be brought to its knees if the data upon which it operates is not laid out efficiently in memory.

In this section, we'll learn how to optimize our code's memory utilization along these two axes.

6.2.1 Optimizing Dynamic Memory Allocation

Dynamic memory allocation via `malloc()` and `free()` or C++'s global `new` and `delete` operators—also known as *heap allocation*—is typically very slow. The high cost can be attributed to two main factors. First, a heap allocator is a general-purpose facility, so it must be written to handle any allocation size, from one byte to one gigabyte. This requires a lot of management overhead, making the `malloc()` and `free()` functions inherently slow. Second,

在大多数操作系统上，调用 malloc() 或 free() 必须先从用户模式切换到内核模式，处理请求，然后再切换回程序。这些上下文切换的开销可能非常大。游戏开发中经常遵循的一条经验法则是：

> 将堆分配保持在最低限度，并且永远不要在紧密循环内从堆进行分配。

当然，没有游戏引擎可以完全避免动态内存分配，因此大多数游戏引擎都会实现一个或多个自定义分配器。自定义分配器的性能特性优于操作系统的堆分配器，原因有二。首先，自定义分配器可以满足预分配内存块（本身使用 malloc() 或 new 分配，或声明为全局变量）的请求。这使得它可以在用户模式下运行，并完全避免上下文切换到操作系统的成本。其次，通过对其使用模式做出各种假设，自定义分配器可以比通用堆分配器高效得多。

在接下来的章节中，我们将介绍一些常见的自定义分配器。有关此主题的更多信息，请参阅 Christian Gyrling 的精彩博客文章：http://www.swedishcoding.com/2008/08/31/are-we-out-of-memory。

6.2.1.1 基于堆栈的分配器

许多游戏以类似堆栈的方式分配内存。每当加载新的游戏关卡时，都会为其分配内存。一旦关卡加载完毕，几乎不会发生动态内存分配。关卡结束时，其数据将被卸载，所有内存都将被释放。对于此类内存分配，使用类似堆栈的数据结构非常合理。

堆栈分配器实现起来非常简单。我们只需使用 malloc() 或 global new 分配一大块连续的内存，或者声明一个全局字节数组（在这种情况下，内存实际上是从可执行文件的 BSS 段中分配的）。系统会维护一个指向堆栈顶部的指针。所有低于此指针的内存地址都被视为正在使用中，而所有高于此指针的内存地址都被视为空闲。栈顶指针初始化为堆栈中的最低内存地址。每个分配请求只需将指针向上移动请求的字节数即可。只需将栈顶指针向下移动相应内存块的大小即可释放最近分配的内存块。

on most operating systems a call to `malloc()` or `free()` must first context-switch from user mode into kernel mode, process the request and then context-switch back to the program. These context switches can be extraordinarily expensive. One rule of thumb often followed in game development is:

> Keep heap allocations to a minimum, and never allocate from the heap within a tight loop.

Of course, no game engine can entirely avoid dynamic memory allocation, so most game engines implement one or more custom allocators. A custom allocator can have better performance characteristics than the operating system's heap allocator for two reasons. First, a custom allocator can satisfy requests from a preallocated memory block (itself allocated using `malloc()` or `new`, or declared as a global variable). This allows it to run in user mode and entirely avoid the cost of context-switching into the operating system. Second, by making various assumptions about its usage patterns, a custom allocator can be much more efficient than a general-purpose heap allocator.

In the following sections, we'll take a look at some common kinds of custom allocators. For additional information on this topic, see Christian Gyrling's excellent blog post, http://www.swedishcoding.com/2008/08/31/are-we-out-of-memory.

6.2.1.1 Stack-Based Allocators

Many games allocate memory in a stack-like fashion. Whenever a new game level is loaded, memory is allocated for it. Once the level has been loaded, little or no dynamic memory allocation takes place. At the conclusion of the level, its data is unloaded and all of its memory can be freed. It makes a lot of sense to use a stack-like data structure for these kinds of memory allocations.

A *stack allocator* is very easy to implement. We simply allocate a large contiguous block of memory using `malloc()` or global `new`, or by declaring a global array of bytes (in which case the memory is effectively allocated out of the executable's BSS segment). A pointer to the top of the stack is maintained. All memory addresses below this pointer are considered to be in use, and all addresses above it are considered to be free. The top pointer is initialized to the lowest memory address in the stack. Each allocation request simply moves the pointer up by the requested number of bytes. The most recently allocated block can be freed by simply moving the top pointer back down by the size of the block.

图 6.1。堆栈分配并释放回标记。

需要注意的是，使用堆栈分配器时，内存不能以任意顺序释放。所有释放操作都必须按照与分配顺序相反的顺序执行。强制执行这些限制的一个简单方法是完全禁止释放单个内存块。我们可以提供一个函数，将堆栈顶部回滚到之前标记的位置，从而释放当前顶部和回滚点之间的所有内存块。

务必将栈顶指针回滚到两个已分配块之间的边界点，否则新的分配操作会覆盖最顶层块的尾部。为了确保正确执行此操作，堆栈分配器通常会提供一个函数，该函数返回一个表示当前栈顶的标记。然后，回滚函数会将其中一个标记作为其参数。如图 6.1 所示。堆栈分配器的接口通常如下所示。

```
class StackAllocator
{
public:
    // Stack marker: Represents the current top of the
    // stack. You can only roll back to a marker, not to
    // arbitrary locations within the stack.
    typedef U32 Marker;
```

Figure 6.1. Stack allocation and freeing back to a marker.

It is important to realize that with a stack allocator, memory cannot be freed in an arbitrary order. All frees must be performed in an order opposite to that in which they were allocated. One simple way to enforce these restrictions is to disallow individual blocks from being freed at all. Instead, we can provide a function that rolls the stack top back to a previously marked location, thereby freeing all blocks between the current top and the roll-back point.

It's important to always roll the top pointer back to a point that lies at the boundary between two allocated blocks, because otherwise new allocations would overwrite the tail end of the top-most block. To ensure that this is done properly, a stack allocator often provides a function that returns a *marker* representing the current top of the stack. The roll-back function then takes one of these markers as its argument. This is depicted in Figure 6.1. The interface of a stack allocator often looks something like this.

```
class StackAllocator
{
public:
    // Stack marker: Represents the current top of the
    // stack. You can only roll back to a marker, not to
    // arbitrary locations within the stack.
    typedef U32 Marker;
```

```
    // Constructs a stack allocator with the given total
    // size.
    explicit StackAllocator(U32 stackSize_bytes);

    // Allocates a new block of the given size from stack
    // top.
    void* alloc(U32 size_bytes);

    // Returns a marker to the current stack top.
    Marker getMarker();

    // Rolls the stack back to a previous marker.
    void freeToMarker(Marker marker);

    // Clears the entire stack (rolls the stack back to
    // zero).
    void clear();

private:
    // ...
};
```

双端堆栈分配器

单个内存块实际上可以包含两个堆栈分配器——一个从内存块底部向上分配，另一个从内存块顶部向下分配。双端堆栈分配器非常有用，因为它允许在底部堆栈的内存使用量和顶部堆栈的内存使用量之间进行权衡，从而更高效地利用内存。在某些情况下，两个堆栈可能使用大致相同的内存量，并在内存块中间相遇。在其他情况下，两个堆栈中的一个可能比另一个堆栈占用更多的内存，但只要请求的内存总量不超过两个堆栈共享的内存块，所有分配请求仍然可以得到满足。如图 6.2 所示。

在 Midway 的街机游戏《Hydro Thunder》中，所有内存分配都来自一个由双端堆栈分配器管理的大型内存块。底层堆栈用于加载和卸载关卡（赛道），而顶层堆栈用于每帧分配和释放的临时内存块。这种分配方案非常有效，确保了《Hydro Thunder》从未出现过内存碎片问题（参见第 6.2.1.4 节）。《Hydro Thunder》的首席工程师 Steve Ranck 在 [8, 第 1.9 节] 中深入描述了这种分配技术。


```
    // Constructs a stack allocator with the given total
    // size.
    explicit StackAllocator(U32 stackSize_bytes);

    // Allocates a new block of the given size from stack
    // top.
    void* alloc(U32 size_bytes);

    // Returns a marker to the current stack top.
    Marker getMarker();

    // Rolls the stack back to a previous marker.
    void freeToMarker(Marker marker);

    // Clears the entire stack (rolls the stack back to
    // zero).
    void clear();

private:
    // ...
};
```


Double-Ended Stack Allocators

A single memory block can actually contain two stack allocators—one that allocates up from the bottom of the block and one that allocates down from the top of the block. A double-ended stack allocator is useful because it uses memory more efficiently by allowing a trade-off to occur between the memory usage of the bottom stack and the memory usage of the top stack. In some situations, both stacks may use roughly the same amount of memory and meet in the middle of the block. In other situations, one of the two stacks may eat up a lot more memory than the other stack, but all allocation requests can still be satisfied as long as the total amount of memory requested is not larger than the block shared by the two stacks. This is depicted in Figure 6.2.

In Midway's *Hydro Thunder* arcade game, all memory allocations are made from a single large block of memory managed by a double-ended stack allocator. The bottom stack is used for loading and unloading levels (race tracks), while the top stack is used for temporary memory blocks that are allocated and freed every frame. This allocation scheme worked extremely well and ensured that *Hydro Thunder* never suffered from memory fragmentation problems (see Section 6.2.1.4). Steve Ranck, *Hydro Thunder*'s lead engineer, describes this allocation technique in depth in [8, Section 1.9].

图 6.2. 双端堆栈分配器。

6.2.1.2 池分配器

在游戏引擎编程（以及一般的软件工程）中，分配大量大小相同的小内存块是很常见的。例如，我们可能想要分配和释放矩阵、迭代器、链表中的链接或可渲染的网格实例。对于这种类型的内存分配模式，池分配器通常是完美的选择。

池分配器通过预分配一大块内存来工作，该内存块的大小是将要分配的元素大小的整数倍。例如，一个 4 × 4 矩阵的池将是 64 字节的整数倍 —— 即每个矩阵 16 个元素乘以每个元素 4 个字节（假设每个元素都是 32 位浮点数）。池中的每个元素都会添加到空闲元素的链接列表中；首次初始化池时，空闲列表包含所有元素。每当发出分配请求时，我们只需从空闲列表中获取下一个空闲元素并返回它即可。当一个元素被释放时，我们只需将其重新添加到空闲列表中即可。分配和释放都是 O(1) 操作，因为每个操作都只涉及几个指针操作，无论当前有多少个元素是空闲的。（符号 O（1）是“大 O”符号的一个例子。在这种情况下，它意味着分配和释放的执行时间大致恒定，并且不依赖于池中当前元素的数量等因素。有关“大 O”符号的解释，请参见第 6.3.3 节。）

空闲元素的链表可以是单向链表，这意味着每个空闲元素需要一个指针（32 位机器上为 4 个字节，64 位机器上为 8 个字节）。我们应该从哪里获取所有这些指针的内存？当然，它们可以存储在单独的预分配内存块中，占用 (sizeof (void*) * numElementsInPool) 字节。然而，这过于浪费。关键是要认识到，根据定义，空闲列表中的内存块就是空闲内存块。那么为什么不将每个空闲列表的“下一个”指针存储在空闲块本身内呢？只要 elementSize >= sizeof(void*)，这个小“技巧”□□就有效。我们不会浪费任何内存，因为我们的空闲列表指针都驻留在空闲内存块内 —— 本来就没用在那些内存中！

如果每个元素都小于一个指针，那么我们可以使用池元素-

Figure 6.2. A double-ended stack allocator.

6.2.1.2 Pool Allocators

It's quite common in game engine programming (and software engineering in general) to allocate lots of small blocks of memory, each of which are the same size. For example, we might want to allocate and free matrices, or iterators, or links in a linked list, or renderable mesh instances. For this type of memory allocation pattern, a *pool allocator* is often the perfect choice.

A pool allocator works by preallocating a large block of memory whose size is an exact multiple of the size of the elements that will be allocated. For example, a pool of 4 × 4 matrices would be an exact multiple of 64 bytes—that's 16 elements per matrix times four bytes per element (assuming each element is a 32-bit `float`). Each element within the pool is added to a linked list of free elements; when the pool is first initialized, the free list contains all of the elements. Whenever an allocation request is made, we simply grab the next free element off the free list and return it. When an element is freed, we simply tack it back onto the free list. Both allocations and frees are $O(1)$ operations, since each involves only a couple of pointer manipulations, no matter how many elements are currently free. (The notation $O(1)$ is an example of "big O" notation. In this case it means that the execution times of both allocations and frees are roughly constant and do not depend on things like the number of elements currently in the pool. See Section 6.3.3 for an explanation of "big O" notation.)

The linked list of free elements can be a singly-linked list, meaning that we need a single pointer (four bytes on 32-bit machines or eight bytes on 64-bit machines) for each free element. Where should we obtain the memory for all these pointers? Certainly they could be stored in a separate preallocated memory block, occupying `(sizeof(void*) * numElementsInPool)` bytes. However, this is unduly wasteful. The key is to realize that the memory blocks residing on the free list are, by definition, free memory blocks. So why not store each free list "next" pointer *within the free block itself*? This little "trick" works as long as `elementSize >= sizeof(void*)`. We don't waste any memory, because our free list pointers all reside inside the free memory blocks—in memory that wasn't being used for anything anyway!

If each element is smaller than a pointer, then we can use pool element in-

使用骰子而不是指针来实现我们的链表。例如，如果我们的池包含 16 位整数，那么我们可以使用 16 位索引作为链表中的“下一个指针”。只要池中的元素不超过 2^16 = 65,536 个，这种方法就有效。

6.2.1.3 对齐分配

正如我们在 3.3.7.1 节中看到的，每个变量和数据对象都有对齐要求。8 位整数变量可以对齐到任意地址，但 32 位整数或浮点变量必须 4 字节对齐，这意味着其地址只能以半字节 0x0、0x4、0x8 或 0xC 结尾。128 位 SIMD 矢量值通常有 16 字节对齐要求，这意味着其内存地址只能以半字节 0x0 结尾。在 PS3 上，为了实现最大 DMA 吞吐量，要通过直接内存访问 (DMA) 控制器传输到 SPU 的内存块应 128 字节对齐，这意味着它们只能以字节 0x00 或 0x80 结尾。

所有内存分配器都必须能够返回对齐的内存块。这实现起来相对简单。我们只需分配比实际请求略大的内存，将内存块的地址稍微上移，使其正确对齐，然后返回移位后的地址。因为我们分配的内存比请求的内存略大，所以即使略微上移，返回的内存块仍然足够大。

在大多数实现中，额外分配的字节数等于对齐值减一，这是我们所能做出的最坏情况的对齐移位。例如，如果我们想要一个 16 字节对齐的内存块，最坏的情况是返回一个以 0x1 结尾的未对齐指针，因为这需要我们进行 15 个字节的移位才能使其到达 16 字节边界。

以下是对齐内存分配器的一种可能实现：

```
// Shift the given address upwards if/as necessary to
// ensure it is aligned to the given number of bytes.
inline uintptr_t AlignAddress(uintptr_t addr, size_t align)
{
    const size_t mask = align - 1;
    assert((align & mask) == 0); // pwr of 2
    return (addr + mask) & ~mask;
}

// Shift the given pointer upwards if/as necessary to
// ensure it is aligned to the given number of bytes.
template<typename T>
inline T* AlignPointer(T* ptr, size_t align)
```


dices instead of pointers to implement our linked list. For example, if our pool contains 16-bit integers, then we can use 16-bit indices as the “next pointers” in our linked list. This works as long as the pool doesn’t contain more than $2^{16} = 65{,}536$ elements.

6.2.1.3 Aligned Allocations

As we saw in Section 3.3.7.1, every variable and data object has an alignment requirement. An 8-bit integer variable can be aligned to any address, but a 32-bit integer or floating-point variable must be 4-byte aligned, meaning its address can only end in the nibbles 0x0, 0x4, 0x8 or 0xC. A 128-bit SIMD vector value generally has a 16-byte alignment requirement, meaning that its memory address can end only in the nibble 0x0. On the PS3, memory blocks that are to be transferred to an SPU via the direct memory access (DMA) controller should be 128-*byte* aligned for maximum DMA throughput, meaning they can only end in the bytes 0x00 or 0x80.

All memory allocators must be capable of returning aligned memory blocks. This is relatively straightforward to implement. We simply allocate a little bit more memory than was actually requested, shift the address of the memory block upward slightly so that it is aligned properly, and then return the shifted address. Because we allocated a bit more memory than was requested, the returned block will still be large enough, even with the slight upward shift.

In most implementations, the number of additional bytes allocated is equal to the alignment minus one, which is the worst-case alignment shift we can make. For example, if we want a 16-byte aligned memory block, the worst case would be to get back an unaligned pointer that ends in 0x1, because it would require us to apply a shift of 15 bytes to bring it to a 16-byte boundary.

Here’s one possible implementation of an aligned memory allocator:

```
// Shift the given address upwards if/as necessary to
// ensure it is aligned to the given number of bytes.
inline uintptr_t AlignAddress(uintptr_t addr, size_t align)
{
    const size_t mask = align - 1;
    assert((align & mask) == 0); // pwr of 2
    return (addr + mask) & ~mask;
}

// Shift the given pointer upwards if/as necessary to
// ensure it is aligned to the given number of bytes.
template<typename T>
inline T* AlignPointer(T* ptr, size_t align)
```

```
{
    const uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
    const uintptr_t addrAligned = AlignAddress(addr, align);
    return reinterpret_cast<T*>(addrAligned);
}

// Aligned allocation function. IMPORTANT: 'align'
// must be a power of 2 (typically 4, 8 or 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Determine worst case number of bytes we'll need.
    size_t worstCaseBytes = bytes + align - 1;

    // Allocate unaligned block.
    U8* pRawMem = new U8[worstCaseBytes];

    // Align the block.
    return AlignPointer(pRawMem, align);
}
```

对齐"魔法"由函数 AlignAddress() 实现。其工作原理如下：给定一个地址和所需的对齐方式 L，我们可以将该地址对齐到 L 字节边界，方法是先在该地址上加上 L - 1，然后去掉结果地址的 N 个最低有效位，其中 N = log 2 (L)。例如，要将任何地址对齐到 16 字节边界，我们将其向上移动 15 个字节，然后屏蔽掉 N = log 2 (16) = 4 个最低有效位。

为了去除这些位，我们需要一个掩码，然后使用按位与运算符将其应用于地址。由于 L 始终是 2 的幂，因此 L - 1 是一个掩码，其最低有效位 N 为二进制 1，其余位为二进制 0。因此，我们需要做的就是取反这个掩码，然后将其与地址进行与运算。

(`addr & ~mask`).

释放对齐的块

当对齐的内存块稍后被释放时，我们传递的是移位后的地址，而不是我们分配的原始地址。但是，为了释放内存，我们需要释放 new 实际返回的地址。那么，如何将对齐的地址转换回其原始的未对齐地址呢？

一种简单的方法是将移位（即对齐地址与原始地址之间的差值）存储在我们的 free 函数能够找到的地方。回想一下，我们实际上在 AllocAligned() 中分配了额外的 align 1 个字节，以便为指针对齐提供一些空间。这些


```
{
    const uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
    const uintptr_t addrAligned = AlignAddress(addr, align);
    return reinterpret_cast<T*>(addrAligned);
}

// Aligned allocation function. IMPORTANT: 'align'
// must be a power of 2 (typically 4, 8 or 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Determine worst case number of bytes we'll need.
    size_t worstCaseBytes = bytes + align - 1;

    // Allocate unaligned block.
    U8* pRawMem = new U8[worstCaseBytes];

    // Align the block.
    return AlignPointer(pRawMem, align);
}
```

The alignment "magic" is performed by the function `AlignAddress()`. Here's how it works: Given an address and a desired alignment L, we can align that address to an L-byte boundary by first adding $L - 1$ to it, and then stripping off the N least-significant bits of the resulting address, where $N = \log_2(L)$. For example, to align any address to a 16-byte boundary, we shift it up by 15 bytes and then mask off the $N = \log_2(16) = 4$ least-significant bits.

To strip off these bits, we need a *mask* that we can apply to the address using the bitwise AND operator. Because L is always a power of two, $L - 1$ will be a mask with binary 1s in the N least-significant bits and binary 0s everywhere else. So all we need to do is *invert* this mask and then AND it with the address (`addr & ~mask`).

Freeing Aligned Blocks

When an aligned block is later freed, we will be passed the *shifted* address, not the original address that we allocated. But in order to free the memory, we need to free the address that was actually returned by `new`. How can we convert an aligned address back into its original, unaligned address?

One simple approach is to store the shift (i.e., the difference between the aligned address and the original address) some place where our free function will be able to find it. Recall that we actually allocate `align - 1` extra bytes in `AllocAligned()`, in order to give us some room to align the pointer. Those

额外的字节是存储移位值的理想位置。我们能做的最小移位是一个字节，所以这就是存储偏移量的最小空间。因此，给定一个对齐指针 p ，我们可以简单地将移位值存储为一个字节值，存放在地址 p 1 处。

然而，有一个问题：new 返回的原始地址可能已经对齐。在这种情况下，我们上面给出的代码根本不会移动原始地址，这意味着没有多余的字节来存储偏移量。为了解决这个问题，我们只需分配 L 个额外的字节，而不是 L - 1 个，然后我们总是将原始指针向上移动到下一个 L 字节边界，即使它已经对齐。现在最大移位量为 L 个字节，最小移位量为 1 个字节。因此，总会有至少一个额外的字节可以用来存储移位值。

将移位存储在单个字节中，可以实现最大 128 字节的对齐。我们永远不会将指针移动零字节，因此，通过将不可能的零移位值解释为 256 字节的移位，我们可以使该方案适用于最大 256 字节的对齐。（对于更大的对齐，我们必须分配更多字节，并将指针进一步向上移动，以便为更宽的“头部”腾出空间。）

修改后的 AllocAligned() 函数及其对应的 FreeAligned() 函数的实现如下。分配和释放对齐块的过程如图 6.3 所示。

```
// Aligned allocation function. IMPORTANT: 'align'
// must be a power of 2 (typically 4, 8 or 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Allocate 'align' more bytes than we need.
    size_t actualBytes = bytes + align;

    // Allocate unaligned block.
    U8* pRawMem = new U8[actualBytes];

    // Align the block. If no alignment occurred,
    // shift it up the full 'align' bytes so we
    // always have room to store the shift.
    U8* pAlignedMem = AlignPointer(pRawMem, align);
    if (pAlignedMem == pRawMem)
        pAlignedMem += align;

    // Determine the shift, and store it.
    // (This works for up to 256-byte alignment.)
    ptrdiff_t shift = pAlignedMem - pRawMem;
    assert(shift > 0 && shift <= 256);
```


extra bytes are a perfect place to store the shift value. The smallest shift we'll ever make is one byte, so that's the minimum space we'll have to store the offset. Therefore, given an aligned pointer `p`, we can simply store the shift as a one-byte value at address `p - 1`.

However, there's one problem: It's possible that the raw address returned by `new` will already be aligned. In that case, the code we presented above would not shift the raw address at all, meaning there'd be no extra bytes into which to store the offset. To overcome this, we simply allocate L extra bytes, instead of $L - 1$, and then we *always* shift the raw pointer up to the next L-byte boundary, even if it was already aligned. Now the maximum shift will be L bytes, and the minimum shift will be 1 byte. So there will always be at least one extra byte into which we can store our shift value.

Storing the shift in a single byte works for alignments up to and including 128. We never ever shift a pointer by zero bytes, so we can make this scheme work for up to 256-byte alignment by interpreting the impossible shift value of zero as a 256-byte shift. (For larger alignments, we'd have to allocate even more bytes, and shift the pointer even further up, to make room for a wider "header".)

Here's how the modified `AllocAligned()` function and its corresponding `FreeAligned()` function could be implemented. The process of allocating and freeing aligned blocks is illustrated in Figure 6.3.

```
// Aligned allocation function. IMPORTANT: 'align'
// must be a power of 2 (typically 4, 8 or 16).
void* AllocAligned(size_t bytes, size_t align)
{
    // Allocate 'align' more bytes than we need.
    size_t actualBytes = bytes + align;

    // Allocate unaligned block.
    U8* pRawMem = new U8[actualBytes];

    // Align the block. If no alignment occurred,
    // shift it up the full 'align' bytes so we
    // always have room to store the shift.
    U8* pAlignedMem = AlignPointer(pRawMem, align);
    if (pAlignedMem == pRawMem)
        pAlignedMem += align;

    // Determine the shift, and store it.
    // (This works for up to 256-byte alignment.)
    ptrdiff_t shift = pAlignedMem - pRawMem;
    assert(shift > 0 && shift <= 256);
```

```
    pAlignedMem[-1] = static_cast<U8>(shift & 0xFF);

    return pAlignedMem;
}

void FreeAligned(void* pMem)
{
    if (pMem)
    {
        // Convert to U8 pointer.
        U8* pAlignedMem = reinterpret_cast<U8*>(pMem);

        // Extract the shift.
        ptrdiff_t shift = pAlignedMem[-1];
        if (shift == 0)
            shift = 256;

        // Back up to the actual allocated address,
        // and array-delete it.
        U8* pRawMem = pAlignedMem - shift;
        delete[] pRawMem;
    }
}
```

图 6.3 对齐内存分配，要求 16 字节对齐。分配的内存地址与调整（对齐）后的地址之间的差值存储在调整后地址之前的字节中，以便在释放内存时可以检索。

6.2.1.4 单帧和双缓冲内存分配器

几乎所有游戏引擎都会在游戏循环中分配至少一些临时数据。这些数据要么在循环的每次迭代结束时被丢弃，要么在下一帧使用后被丢弃。这种分配模式非常常见，以至于许多引擎都支持单帧和双缓冲分配器。


```
    pAlignedMem[-1] = static_cast<U8>(shift & 0xFF);

    return pAlignedMem;
}

void FreeAligned(void* pMem)
{
    if (pMem)
    {
        // Convert to U8 pointer.
        U8* pAlignedMem = reinterpret_cast<U8*>(pMem);

        // Extract the shift.
        ptrdiff_t shift = pAlignedMem[-1];
        if (shift == 0)
            shift = 256;

        // Back up to the actual allocated address,
        // and array-delete it.
        U8* pRawMem = pAlignedMem - shift;
        delete[] pRawMem;
    }
}
```

Figure 6.3. Aligned memory allocation with a 16-byte alignment requirement. The difference between the allocated memory address and the adjusted (aligned) address is stored in the byte immediately preceding the adjusted address, so that it may be retrieved during free.

6.2.1.4 Single-Frame and Double-Buffered Memory Allocators

Virtually all game engines allocate at least some temporary data during the game loop. This data is either discarded at the end of each iteration of the loop or used on the next frame and then discarded. This allocation pattern is so common that many engines support *single-frame* and *double-buffered* allocators.

单帧分配器

单帧分配器是通过预留一块内存并使用如上所述的简单堆栈分配器进行管理来实现的。在每个帧的开始处，堆栈的“顶部”指针会被清除，指向内存块的底部。在帧期间进行的分配会向内存块的顶部增长。

冲洗并重复。

```
StackAllocator g_singleFrameAllocator;

// Main Game Loop
while (true)
{
    // Clear the single-frame allocator's buffer every
    // frame.
    g_singleFrameAllocator.clear();

    // ...

    // Allocate from the single-frame buffer. We never
    // need to free this data! Just be sure to use it
    // only this frame.
    void* p = g_singleFrameAllocator.alloc(nBytes);

    // ...
}
```

单帧分配器的主要优点之一是分配的内存无需释放——我们可以放心，分配器会在每一帧开始时被清除。单帧分配器的速度也快得惊人。但它的一个缺点是，使用单帧分配器需要程序员具备一定的自律性。你需要意识到，从单帧缓冲区中分配的内存块仅在当前帧有效。程序员绝不能将指向单帧内存块的指针缓存到帧边界之外！

双缓冲分配器

双缓冲分配器允许在帧 i 上分配的内存块在帧（i + 1）上使用。为了实现这一点，我们创建了两个大小相同的单帧堆栈分配器，然后在每个帧之间进行乒乓切换。

```
class DoubleBufferedAllocator
{
    U32             m_curStack;
```


Single-Frame Allocators

A single-frame allocator is implemented by reserving a block of memory and managing it with a simple stack allocator as described above. At the beginning of each frame, the stack's "top" pointer is cleared to the bottom of the memory block. Allocations made during the frame grow toward the top of the block. Rinse and repeat.

```
StackAllocator g_singleFrameAllocator;

// Main Game Loop
while (true)
{
    // Clear the single-frame allocator's buffer every
    // frame.
    g_singleFrameAllocator.clear();

    // ...

    // Allocate from the single-frame buffer. We never
    // need to free this data! Just be sure to use it
    // only this frame.
    void* p = g_singleFrameAllocator.alloc(nBytes);

    // ...
}
```

One of the primary benefits of a single-frame allocator is that allocated memory needn't ever be freed—we can rely on the fact that the allocator will be cleared at the start of every frame. Single-frame allocators are also blindingly fast. The one big negative is that using a single-frame allocator requires a reasonable level of discipline on the part of the programmer. You need to realize that a memory block allocated out of the single-frame buffer will only be valid during the current frame. Programmers must *never* cache a pointer to a single-frame memory block across the frame boundary!

Double-Buffered Allocators

A double-buffered allocator allows a block of memory allocated on frame i to be used on frame $(i+1)$. To accomplish this, we create two single-frame stack allocators of equal size and then ping-pong between them every frame.

```
class DoubleBufferedAllocator
{
    U32             m_curStack;
```

```
    StackAllocator m_stack[2];

public:

    void swapBuffers()
    {
        m_curStack = (U32)!m_curStack;
    }

    void clearCurrentBuffer()
    {
        m_stack[m_curStack].clear();
    }

    void* alloc(U32 nBytes)
    {
        return m_stack[m_curStack].alloc(nBytes);
    }

    // ...
};

// ...

DoubleBufferedAllocator g_doubleBufAllocator;

// Main Game Loop
while (true)
{
    // Clear the single-frame allocator every frame as
    // before.
    g_singleFrameAllocator.clear();

    // Swap the active and inactive buffers of the double-
    // buffered allocator.
    g_doubleBufAllocator.swapBuffers();

    // Now clear the newly active buffer, leaving last
    // frame's buffer intact.
    g_doubleBufAllocator.clearCurrentBuffer();

    // ...


    // Allocate out of the current buffer, without
    // disturbing last frame's data. Only use this data
    // this frame or next frame. Again, this memory never
```



```
    StackAllocator m_stack[2];

public:

    void swapBuffers()
    {
        m_curStack = (U32)!m_curStack;
    }

    void clearCurrentBuffer()
    {
        m_stack[m_curStack].clear();
    }

    void* alloc(U32 nBytes)
    {
        return m_stack[m_curStack].alloc(nBytes);
    }

    // ...
};

// ...

DoubleBufferedAllocator g_doubleBufAllocator;

// Main Game Loop
while (true)
{
    // Clear the single-frame allocator every frame as
    // before.
    g_singleFrameAllocator.clear();

    // Swap the active and inactive buffers of the double-
    // buffered allocator.
    g_doubleBufAllocator.swapBuffers();

    // Now clear the newly active buffer, leaving last
    // frame's buffer intact.
    g_doubleBufAllocator.clearCurrentBuffer();

    // ...


    // Allocate out of the current buffer, without
    // disturbing last frame's data. Only use this data
    // this frame or next frame. Again, this memory never
```

```
        // needs to be freed.
        void* p = g_doubleBufAllocator.alloc(nBytes);

        // ...
    }
```

这种分配器对于在 Xbox 360、Xbox One、PlayStation 3 或 PlayStation 4 等多核游戏机上缓存异步处理的结果非常有用。在第 i 帧，我们可以在 PS4 的某个核心上启动一个异步作业，例如，将从我们的双缓冲分配器分配的目标缓冲区的地址交给它。该作业在第 i 帧结束前的某个时间运行并产生其结果，将它们存储到我们提供的缓冲区中。在第（i + 1）帧，缓冲区被交换。作业的结果现在位于非活动缓冲区中，因此它们不会被在此帧期间可能进行的任何双缓冲分配覆盖。只要我们在第（i + 2）帧之前使用该作业的结果，我们的数据就不会被覆盖。

6.2.2 内存碎片

动态堆分配的另一个问题是，随着时间的推移，内存会变得碎片化。当程序首次运行时，它的堆内存完全是空闲的。分配一个块时，适当大小的连续堆内存区域被标记为“正在使用”，而堆的其余部分保持空闲。当一个块被释放时，它也会被标记为释放，相邻的空闲块会合并成一个更大的空闲块。随着时间的推移，由于各种大小的分配和释放以随机顺序发生，堆内存开始看起来像是空闲块和已使用块的拼凑物。我们可以将空闲区域视为已使用内存结构中的“洞”。当洞的数量变大和/或所有洞都相对较小时，我们说内存已经变得碎片化。如图 6.4 所示。

内存碎片化的问题在于，即使有足够的可用字节来满足请求，分配也可能失败。问题的关键在于，分配的内存块必须始终是连续的。例如，为了满足 128 KiB 的请求，必须存在一个 128 KiB 或更大的空闲“空洞”。如果有两个空洞，每个空洞的大小为 64 KiB，则虽然有足够的可用字节，但由于它们不是连续的字节，分配仍然会失败。

在支持虚拟内存的操作系统上，内存碎片问题并不严重。虚拟内存系统将不连续的物理内存块（称为页面）映射到虚拟地址空间中，


```
        // needs to be freed.
        void* p = g_doubleBufAllocator.alloc(nBytes);

        // ...
    }
```

This kind of allocator is extremely useful for caching the results of asynchronous processing on a multicore game console like the Xbox 360, Xbox One, PlayStation 3 or PlayStation 4. On frame i, we can kick off an asynchronous job on one of the PS4's cores, for example, handing it the address of a destination buffer that has been allocated from our double-buffered allocator. The job runs and produces its results some time before the end of frame i, storing them into the buffer we provided. On frame $(i+1)$, the buffers are swapped. The results of the job are now in the inactive buffer, so they will not be overwritten by any double-buffered allocations that might be made during this frame. As long as we use the results of the job before frame $(i+2)$, our data won't be overwritten.

6.2.2 Memory Fragmentation

Another problem with dynamic heap allocations is that memory can become *fragmented* over time. When a program first runs, its heap memory is entirely free. When a block is allocated, a contiguous region of heap memory of the appropriate size is marked as "in use," and the remainder of the heap remains free. When a block is freed, it is marked as such, and adjacent free blocks are merged into a single, larger free block. Over time, as allocations and deallocations of various sizes occur in random order, the heap memory begins to look like a patchwork of free and used blocks. We can think of the free regions as "holes" in the fabric of used memory. When the number of holes becomes large, and/or the holes are all relatively small, we say the memory has become *fragmented*. This is illustrated in Figure 6.4.

The problem with memory fragmentation is that allocations may fail even when there are enough free bytes to satisfy the request. The crux of the problem is that allocated memory blocks must always be *contiguous*. For example, in order to satisfy a request of 128 KiB, there must exist a free "hole" that is 128 KiB or larger. If there are two holes, each of which is 64 KiB in size, then enough bytes are available but the allocation fails because they are not *contiguous bytes*.

Memory fragmentation is not as much of a problem on operating systems that support *virtual memory*. A virtual memory system maps discontiguous blocks of physical memory known as *pages* into a *virtual address space*, in

图 6.4. 内存碎片。

页面在应用程序看来是连续的。当物理内存不足时，可以将过期页面交换到硬盘，并在需要时从磁盘重新加载。有关虚拟内存工作原理的详细讨论，请参阅 http://en.wikipedia.org/wiki/Virtual_memory。大多数嵌入式系统无法实现虚拟内存系统。虽然一些现代游戏机在技术上支持虚拟内存，但大多数游戏机引擎仍然不使用虚拟内存，因为它本身会带来性能开销。

6.2.2.1 使用堆栈和池分配器避免碎片

通过使用堆栈和/或池分配器可以避免内存碎片的有害影响。

- 堆栈分配器不受碎片影响，因为分配始终是连续的，并且块必须按照与分配顺序相反的顺序释放。如图 6.5 所示。

Figure 6.4. Memory fragmentation.

which the pages appear to the application to be contiguous. Stale pages can be swapped to the hard disk when physical memory is in short supply and reloaded from disk when they are needed. For a detailed discussion of how virtual memory works, see http://en.wikipedia.org/wiki/Virtual_memory. Most embedded systems cannot afford to implement a virtual memory system. While some modern consoles do technically support it, most console game engines still do not make use of virtual memory due to the inherent performance overhead.

6.2.2.1 Avoiding Fragmentation with Stack and Pool Allocators

The detrimental effects of memory fragmentation can be avoided by using stack and/or pool allocators.

- A stack allocator is impervious to fragmentation because allocations are always contiguous, and blocks must be freed in an order opposite to that in which they were allocated. This is illustrated in Figure 6.5.

图 6.5。堆栈分配器没有碎片问题。

图 6.6。池分配器不会因碎片而降级。

• 池分配器也不存在碎片问题。池确实会产生碎片，但碎片永远不会像通用堆那样导致过早的内存不足。池分配请求永远不会因为缺少足够大的连续空闲块而失败，因为所有块的大小完全相同。如图 6.6 所示。

6.2.2.2 碎片整理和重定位

当大小不同的对象以随机顺序分配和释放时，基于堆栈的分配器和基于池的分配器均无法使用。在这种情况下，可以通过定期对堆进行碎片整理来避免碎片。碎片整理包括将已分配的块从较高的内存地址向下移动到较低的地址（从而将空洞向上移动到较高的地址），以合并堆中所有空闲的“空洞”。一种简单的算法是搜索第一个“空洞”，然后将紧邻该空洞上方的已分配块向下移动到该空洞的起始位置。这会产生将空洞“向上冒泡”到更高内存地址的效果。如果重复此过程，最终所有已分配的块将占据堆地址空间低端的连续内存区域，并且所有空洞都将向上冒泡到堆高端的一个大空洞中。如图 6.7 所示。

上面描述的内存块的移动实现起来并不特别棘手。真正棘手的是，我们要移动已分配的内存块。如果有人持有指向这些已分配内存块的指针，那么移动该内存块将使该指针失效。

Figure 6.5. A stack allocator is free from fragmentation problems.

Figure 6.6. A pool allocator is not degraded by fragmentation.

- A pool allocator is also free from fragmentation problems. Pools *do* become fragmented, but the fragmentation never causes premature out-of-memory conditions as it does in a general-purpose heap. Pool allocation requests can never fail due to a lack of a large enough contiguous free block, because all of the blocks are exactly the same size. This is shown in Figure 6.6.

6.2.2.2 Defragmentation and Relocation

When differently sized objects are being allocated and freed in a random order, neither a stack-based allocator nor a pool-based allocator can be used. In such cases, fragmentation can be avoided by periodically *defragmenting* the heap. Defragmentation involves coalescing all of the free “holes” in the heap by shifting allocated blocks from higher memory addresses down to lower addresses (thereby shifting the holes up to higher addresses). One simple algorithm is to search for the first “hole” and then take the allocated block immediately above the hole and shift it down to the start of the hole. This has the effect of “bubbling up” the hole to a higher memory address. If this process is repeated, eventually all the allocated blocks will occupy a contiguous region of memory at the low end of the heap’s address space, and all the holes will have bubbled up into one big hole at the high end of the heap. This is illustrated in Figure 6.7.

The shifting of memory blocks described above is not particularly tricky to implement. What *is* tricky is accounting for the fact that we’re moving *allocated* blocks of memory around. If anyone has a *pointer* into one of these allocated blocks, then moving the block will invalidate the pointer.

图 6.7. 通过将分配的块移至较低地址来进行碎片整理。

解决这个问题的方法是将所有指针修补到移位后的内存块中，使它们在移位后指向正确的新地址。这个过程称为指针重定位。遗憾的是，没有通用的方法来查找指向特定内存区域的所有指针。因此，如果我们要在游戏引擎中支持内存碎片整理，程序员必须手动仔细跟踪所有指针以便进行重定位，或者必须放弃指针，转而采用本质上更易于重定位的指针，例如智能指针或句柄。

智能指针是一个小型类，它包含一个指针，并且在大多数意图和目的上都像指针一样工作。但由于智能指针是一个类，因此可以对其进行编码以正确处理内存重定位。一种方法是安排所有智能指针将自身添加到全局链表中。每当在堆内移动一个内存块时，都可以扫描所有智能指针的链表，并适当调整指向被移动内存块的每个指针。

句柄通常实现为不可重定位表的索引，该表本身包含指针。当分配的内存块在内存中移动时，可以扫描句柄表并自动找到并更新所有相关指针。由于句柄只是指针表中的索引，因此无论内存块如何移动，它们的值都不会改变，因此使用这些句柄的对象永远不会受到内存重定位的影响。

当某些内存块无法重定位时，重定位就会出现另一个问题。例如，如果您使用的第三方库不使用智能指针或句柄，则指向其数据结构的任何指针都可能无法重定位。解决这个问题的最佳方法通常是安排相关库从可重定位内存区域之外的特殊缓冲区分配内存。另一种选择是简单地

Figure 6.7. Defragmentation by shifting allocated blocks to lower addresses.

The solution to this problem is to patch any and all pointers into a shifted memory block so that they point to the correct new address after the shift. This procedure is known as pointer *relocation*. Unfortunately, there is no general-purpose way to *find* all the pointers that point into a particular region of memory. So if we are going to support memory defragmentation in our game engine, programmers must either carefully keep track of all the pointers manually so they can be relocated, or pointers must be abandoned in favor of something inherently more amenable to relocation, such as *smart pointers* or *handles*.

A smart pointer is a small class that contains a pointer and acts like a pointer for most intents and purposes. But because a smart pointer is a class, it can be coded to handle memory relocation properly. One approach is to arrange for all smart pointers to add themselves to a global linked list. Whenever a block of memory is shifted within the heap, the linked list of all smart pointers can be scanned, and each pointer that points into the shifted block of memory can be adjusted appropriately.

A handle is usually implemented as an index into a non-relocatable table, which itself contains the pointers. When an allocated block is shifted in memory, the handle table can be scanned and all relevant pointers found and updated automatically. Because the handles are just indices into the pointer table, their values never change no matter how the memory blocks are shifted, so the objects that use the handles are never affected by memory relocation.

Another problem with relocation arises when certain memory blocks cannot be relocated. For example, if you are using a third-party library that does not use smart pointers or handles, it's possible that any pointers into its data structures will not be relocatable. The best way around this problem is usually to arrange for the library in question to allocate its memory from a special buffer outside of the relocatable memory area. The other option is to simply

接受某些块不可重定位。如果不可重定位块的数量和大小都很小，重定位系统仍然会表现良好。

值得注意的是，顽皮狗的所有引擎都支持碎片整理。为了避免重定位指针，我们尽可能地使用句柄。然而，在某些情况下，原始指针的使用是不可避免的。每当碎片整理导致内存块发生移动时，这些指针都会被仔细跟踪并手动重定位。顽皮狗的一些游戏对象类由于各种原因不可重定位。然而，如上所述，这不会造成任何实际问题，因为此类对象的数量始终非常少，并且与可重定位内存区域的总大小相比，它们的大小非常小。

摊销碎片整理成本

碎片整理可能是一个缓慢的操作，因为它涉及复制内存块。但是，我们无需一次性对堆进行完全碎片整理。相反，可以将成本分摊到多个帧上。我们可以允许每帧最多移动 N 个已分配的块，N 可以取较小的值，例如 8 或 16。如果我们的游戏以每秒 30 帧的速度运行，那么每帧持续 1/30 秒（33 毫秒）。因此，通常可以在不到一秒的时间内完全整理堆，而不会对游戏的帧速率产生任何明显的影响。只要分配和释放的速度不超过碎片整理移动的速度，堆将始终保持基本碎片整理的状态。

这种方法仅当每个块的大小相对较小时才有效，这样移动单个块所需的时间就不会超过每帧分配给重定位的时间。如果需要重定位非常大的块，我们通常可以将它们拆分成两个或多个子块，每个子块都可以独立重定位。这在顽皮狗的引擎中尚未被证明是一个问题，因为重定位仅用于动态游戏对象，而且它们的大小永远不会超过几千字节——通常要小得多。

6.3 容器

游戏程序员使用各种面向集合的数据结构，也称为容器或集合。容器的作用始终相同——容纳和管理零个或多个数据元素；然而，它们的具体实现方式却千差万别，每种类型的容器都有其优缺点。常见的容器数据类型包括但不限于：

accept that some blocks will not be relocatable. If the number and size of the non-relocatable blocks are both small, a relocation system will still perform quite well.

It is interesting to note that all of Naughty Dog's engines have supported defragmentation. Handles are used wherever possible to avoid the need to relocate pointers. However, in some cases raw pointers cannot be avoided. These pointers are carefully tracked and relocated manually whenever a memory block is shifted due to defragmentation. A few of Naughty Dog's game object classes are not relocatable for various reasons. However, as mentioned above, this doesn't pose any practical problems, because the number of such objects is always very small, and their sizes are tiny when compared to the overall size of the relocatable memory area.

Amortizing Defragmentation Costs

Defragmentation can be a slow operation because it involves copying memory blocks. However, we needn't fully defragment the heap all at once. Instead, the cost can be amortized over many frames. We can allow up to N allocated blocks to be shifted each frame, for some small value of N like 8 or 16. If our game is running at 30 frames per second, then each frame lasts 1/30 of a second (33 ms). So, the heap can usually be completely defragmented in less than one second without having any noticeable effect on the game's frame rate. As long as allocations and deallocations aren't happening at a faster rate than the defragmentation shifts, the heap will remain mostly defragmented at all times.

This approach is only valid when the size of each block is relatively small, so that the time required to move a single block does not exceed the time allotted to relocation each frame. If very large blocks need to be relocated, we can often break them up into two or more subblocks, each of which can be relocated independently. This hasn't proved to be a problem in Naughty Dog's engine, because relocation is only used for dynamic game objects, and they are never larger than a few kibibytes—and usually much smaller.

6.3 Containers

Game programmers employ a wide variety of collection-oriented data structures, also known as *containers* or *collections*. The job of a container is always the same—to house and manage zero or more data elements; however, the details of how they do this vary greatly, and each type of container has its pros and cons. Common container data types include, but are certainly not limited

至，以下。

• 数组。一个有序的、连续的元素集合，通过索引访问。数组的长度通常在编译时静态定义。它可能是多维的。C 和 C++ 原生支持这些（例如，

`int a[5]`).

• 动态数组. 长度可以在运行时动态改变的数组（例如，C++ 标准库的 std::vector）。

• 链表。元素的有序集合，它们不是连续存储在内存中，而是通过指针相互链接（例如，C++ 标准库的 std::list）。

• 栈。支持后进先出 (LIFO) 模型添加和移除元素的容器，也称为推/弹出（例如，

`std::stack`).

• 队列. 支持先进先出 (FIFO) 模型添加和删除元素的容器（例如 std::queue）。

• Deque . 双端队列——支持在数组两端高效插入和移除（例如， std::deque）。

• 树。一种按层次结构对元素进行分组的容器。每个元素（节点）有零个或一个父元素，以及零个或多个子元素。树是有向无环图 (DAG) 的一种特例（见下文）。

• 二叉搜索树 (BST)。一种每个节点最多有两个子节点的树，具有有序属性，使节点按照某些明确定义的标准进行排序。二叉搜索树有很多种，包括红黑树、伸展树、AVL 树等等。

• 二叉堆。一种二叉树，它通过两个规则保持自身的排序顺序，很像二叉搜索树：形状属性，它指定树必须完全填充，并且树的最后一行从左到右填充；堆属性，它规定每个节点都根据某些用户定义的标准“大于”或“等于”其所有子节点。

• 优先级队列。一种允许以任意顺序添加元素，并按照元素自身属性（即优先级）定义的顺序移除元素的容器。优先级队列通常实现为堆（例如 std::priority_queue），但也有其他实现方式。优先级队列有点像始终保持排序的列表，不同之处在于优先级队列仅支持检索最高优先级的元素，并且很少在底层实现为列表。

to, the following.

- *Array*. An ordered, contiguous collection of elements accessed by index. The length of the array is usually statically defined at compile time. It may be multidimensional. C and C++ support these natively (e.g., `int a[5]`).
- *Dynamic array*. An array whose length can change dynamically at runtime (e.g., the C++ standard library's `std::vector`).
- *Linked list*. An ordered collection of elements not stored contiguously in memory but rather linked to one another via pointers (e.g., the C++ standard library's `std::list`).
- *Stack*. A container that supports the last-in-first-out (LIFO) model for adding and removing elements, also known as push/pop (e.g., `std::stack`).
- *Queue*. A container that supports the first-in-first-out (FIFO) model for adding and removing elements (e.g., `std::queue`).
- *Deque*. A double-ended queue—supports efficient insertion and removal at both ends of the array (e.g., `std::deque`).
- *Tree*. A container in which elements are grouped hierarchically. Each element (node) has zero or one parent and zero or more children. A tree is a special case of a DAG (see below).
- *Binary search tree (BST)*. A tree in which each node has at most two children, with an order property to keep the nodes sorted by some well-defined criteria. There are various kinds of binary search trees, including red-black trees, splay trees, AVL trees, etc.
- *Binary heap*. A binary tree that maintains itself in sorted order, much like a binary search tree, via two rules: the *shape property*, which specifies that the tree must be fully filled and that the last row of the tree is filled from left to right; and the *heap property*, which states that every node is, by some user-defined criterion, “greater than” or “equal to” all of its children.
- *Priority queue*. A container that permits elements to be added in any order and then removed in an order defined by some property of the elements themselves (i.e., their *priority*). A priority queue is typically implemented as a *heap* (e.g., `std::priority_queue`), but other implementations are possible. A priority queue is a bit like a list that stays sorted at all times, except that a priority queue only supports retrieval of the highest-priority element, and it is rarely implemented as a list under the hood.

• 字典。一个由键值对组成的表。通过给定对应的键，可以高效地“查找”值。字典也称为映射或哈希表，尽管从技术上讲，哈希表只是字典的一种可能的实现（例如，std::map、std::hash_map）。

• 集合。一种容器，保证所有元素根据某些条件唯一。集合的作用类似于只有键而没有值的字典。

• 图。以任意模式通过单向或双向路径相互连接的节点集合。

• 有向无环图 (DAG)。具有单向（即有向）互连的节点集合，没有循环（即，没有始于和止于同一节点的非空路径）。

6.3.1 容器操作

使用容器类的游戏引擎不可避免地会使用各种常见的算法。例如：

• 插入。向容器中添加一个新元素。新元素可能位于列表的开头、结尾或其他位置；或者容器可能根本没有排序的概念。

• 移除 。从容器中移除一个元素；这可能需要执行查找操作（见下文）。但是，如果存在指向所需元素的迭代器，则使用该迭代器移除元素可能更高效。

• 顺序访问（迭代）。按照某种“自然”的预定义顺序访问容器中的每个元素。

• 随机访问。以任意顺序访问容器中的元素。

• 查找。在容器中搜索符合给定条件的元素。
  查找操作有各种变体，包括反向查找、查找多个元素等。此外，不同类型的数据结构和不同情况需要不同的算法（参见http://en.wikipedia.org/wiki/Search_algorithm）。

• 排序。根据给定的标准对容器的内容进行排序。
  排序算法有很多种，包括冒泡排序、选择排序、插入排序、快速排序等等。（参见http://en.wikipedia。）
  org/wiki/Sorting_algorithm 了解详细信息。）

- *Dictionary*. A table of key-value pairs. A value can be “looked up” efficiently given the corresponding key. A dictionary is also known as a *map* or *hash table*, although technically a hash table is just one possible implementation of a dictionary (e.g., `std::map`, `std::hash_map`).
- *Set*. A container that guarantees that all elements are unique according to some criteria. A set acts like a dictionary with only keys, but no values.
- *Graph*. A collection of nodes connected to one another by unidirectional or bidirectional pathways in an arbitrary pattern.
- *Directed acyclic graph (DAG)*. A collection of nodes with unidirectional (i.e., *directed*) interconnections, with no *cycles* (i.e., there is no nonempty path that starts and ends on the same node).

6.3.1 Container Operations

Game engines that make use of container classes inevitably make use of various commonplace algorithms as well. Some examples include:

- *Insert*. Add a new element to the container. The new element might be placed at the beginning of the list, or the end, or in some other location; or the container might not have a notion of ordering at all.
- *Remove*. Remove an element from the container; this may require a find operation (see below). However, if an iterator is available that refers to the desired element, it may be more efficient to remove the element using the iterator.
- *Sequential access (iteration)*. Accessing each element of the container in some “natural” predefined order.
- *Random access*. Accessing elements in the container in an arbitrary order.
- *Find*. Search a container for an element that meets a given criterion. There are all sorts of variants on the find operation, including finding in reverse, finding multiple elements, etc. In addition, different types of data structures and different situations call for different algorithms (see http://en.wikipedia.org/wiki/Search_algorithm).
- *Sort*. Sort the contents of a container according to some given criteria. There are many different sorting algorithms, including bubble sort, selection sort, insertion sort, quicksort and so on. (See http://en.wikipedia.org/wiki/Sorting_algorithm for details.)

6.3.2 迭代器

迭代器是一个小类，它“知道”如何高效地访问特定容器中的元素。它的作用类似于数组索引或指针——它每次指向容器中的一个元素，可以前进到下一个元素，并且提供了某种机制来测试容器中所有元素是否都已被访问。例如，以下两个代码片段中的第一个使用指针迭代 C 风格数组，而第二个使用几乎相同的语法迭代链表。

```
void processArray(int container[], int numElements)
{
    int* pBegin = &container[0];
    int* pEnd = &container[numElements];

    for (int* p = pBegin; p != pEnd; p++)
    {
        int element = *p;
        // process element...
    }
}

void processList(std::list<int>& container)
{
    std::list<int>::iterator pBegin = container.begin();
    std::list<int>::iterator pEnd = container.end();

    for (auto p = pBegin; p != pEnd; ++p)
    {
        int element = *p;
        // process element...
    }
}
```

与尝试直接访问容器元素相比，使用迭代器的主要好处如下：

- 直接访问会破坏容器类的封装性。另一方面，迭代器通常是容器类的友元，因此它可以高效地迭代，而无需向外界暴露任何实现细节。（事实上，大多数优秀的容器类都会隐藏其内部细节，如果没有迭代器就无法进行迭代。）
- 迭代器可以简化迭代过程。大多数迭代器的行为类似于数组索引或指针，因此可以编写一个简单的循环，其中

6.3.2 Iterators

An iterator is a little class that “knows” how to efficiently visit the elements in a particular kind of container. It acts like an array index or pointer—it refers to one element in the container at a time, it can be advanced to the next element, and it provides some sort of mechanism for testing whether or not all elements in the container have been visited. As an example, the first of the following two code snippets iterates over a C-style array using a pointer, while the second iterates over a linked list using almost identical syntax.

```
void processArray(int container[], int numElements)
{
    int* pBegin = &container[0];
    int* pEnd = &container[numElements];

    for (int* p = pBegin; p != pEnd; p++)
    {
        int element = *p;
        // process element...
    }
}

void processList(std::list<int>& container)
{
    std::list<int>::iterator pBegin = container.begin();
    std::list<int>::iterator pEnd = container.end();

    for (auto p = pBegin; p != pEnd; ++p)
    {
        int element = *p;
        // process element...
    }
}
```

The key benefits to using an iterator over attempting to access the container’s elements directly are as follows:

- Direct access would break the container class’ encapsulation. An iterator, on the other hand, is typically a *friend* of the container class, and as such it can iterate efficiently without exposing any implementation details to the outside world. (In fact, most good container classes hide their internal details and cannot be iterated over *without* an iterator.)
- An iterator can simplify the process of iterating. Most iterators act like array indices or pointers, so a simple loop can be written in which the

迭代器会递增并与终止条件进行比较——即使底层数据结构任意复杂。例如，迭代器可以使按顺序的深度优先树遍历看起来不超过简单的数组迭代复杂度。

6.3.2.1 前增量与后增量

请注意，在 processArray() 示例中，我们使用的是 C++ 的后增运算符 p++ ，而不是预增运算符 ++p 。这是一个微妙但有时很重要的优化。预增运算符在表达式中使用变量（现已修改）的值之前增加变量的内容。后增运算符在使用变量之后增加变量的内容。这意味着编写 ++p 会在代码中引入数据依赖关系 - CPU 必须等待增量操作完成才能在表达式中使用其值。在深度流水线 CPU 上，这会引入停顿。另一方面，使用 p++ 则没有数据依赖关系。变量的值可以立即使用，而增量操作可以稍后发生或与其使用并行发生。无论哪种方式，流水线都不会引入停顿。

当然，在 for 循环的“更新”表达式中，preincrement 和 postincrement 之间应该没有区别。这是因为任何优秀的编译器都能识别出 update_expr 中没有使用该变量的值。但在使用该值的情况下，postincrement 是更可取的，因为它不会在 CPU 流水线中引入停顿。

对于重载了自增运算符的类，不妨打破这条小规则，这在迭代器类中很常见。根据定义，后自增运算符必须返回调用它的对象的未修改副本。根据类数据成员的大小和复杂度，在性能至关重要的循环中使用此类类时，复制迭代器的额外成本可能会使人们倾向于使用前自增。（在像上面显示的 processList() 函数这样简单的示例中，前自增并不一定比后自增更好，但我还是用前自增来实现了它，以突出两者的区别。）

6.3.3 算法复杂度

对于给定的应用程序，选择哪种容器类型取决于所考虑容器的性能和内存特性。对于每种容器类型，我们可以确定插入、删除、查找和排序等常见操作的理论性能。

我们通常表示操作预计要执行的时间 T

iterator is incremented and compared against a terminating condition—even when the underlying data structure is arbitrarily complex. For example, an iterator can make an in-order depth-first tree traversal look no more complex than a simple array iteration.

6.3.2.1 Preincrement versus Postincrement

Notice in the `processArray()` example that we are using C++'s *postincrement* operator, `p++`, rather than the *preincrement* operator, `++p`. This is a subtle but sometimes important optimization. The preincrement operator increments the contents of the variable *before* its (now modified) value is used in the expression. The postincrement operator increments the contents of the variable *after* it has been used. This means that writing `++p` introduces a *data dependency* into your code—the CPU must wait for the increment operation to be completed before its value can be used in the expression. On a deeply pipelined CPU, this introduces a *stall*. On the other hand, with `p++` there is no data dependency. The value of the variable can be used immediately, and the increment operation can happen later or in parallel with its use. Either way, no stall is introduced into the pipeline.

Of course, within the "update" expression of a `for` loop, there should be no difference between pre- and postincrement. This is because any good compiler will recognize that the *value* of the variable isn't used in `update_expr`. But in cases where the value *is* used, postincrement is preferable because it doesn't introduce a stall in the CPU's pipeline.

It can be wise to make an exception to this little rule of thumb for classes with *overloaded* increment operators, as is common practice in *iterator* classes. By definition, the postincrement operator must return an unmodified *copy* of the object on which it is called. Depending on the size and complexity of the data members of the class, the added cost of copying the iterator can tip the scales toward a preference for preincrement when using such classes in performance-critical loops. (Preincrement isn't necessarily better than postincrement in such a simple example as the function `processList()` shown above, but I've implemented it with preincrement to highlight the difference.)

6.3.3 Algorithmic Complexity

The choice of which container type to use for a given application depends upon the performance and memory characteristics of the container being considered. For each container type, we can determine the theoretical performance of common operations such as insertion, removal, find and sort.

We usually indicate the amount of time T that an operation is expected to

作为容器中元素数量 n 的函数：

$$T = f(n).$$

我们并不试图找到函数 f 的精确值，而是只关心函数的整体阶数。例如，假设实际的理论函数是以下任意一个：

$$T = 5n^2 + 17,$$

$$T = 102n^2 + 50n + 12,$$
$$T = \tfrac{1}{2}n^2,$$

在所有情况下，我们都会将表达式简化为最相关的项——在本例中为 n^2。为了表明我们只陈述函数的阶，而不是它的确切方程，我们使用“大 O”符号并写成

$$T = O(n^2).$$

算法的顺序通常可以通过检查伪代码来确定。如果算法的执行时间完全不依赖于容器中元素的数量，我们称其为 O(1)（即，它在常数时间内完成）。如果算法对容器中的元素执行循环并访问每个元素一次，例如在无序列表的线性搜索中，我们称该算法为 O(n)。如果嵌套两个循环，每个循环都可能访问每个节点一次，那么我们称该算法为 O(n2)。如果使用分治方法，如二分搜索（每一步消除一半列表），那么我们预计在最坏的情况下算法实际上只会访问□log2(n)+1□个元素，因此我们将其称为 O(logn) 运算。如果一个算法执行一个子算法 n 次，并且该子算法为 O(log n)，则最终算法将为 O(nlog n)。

要选择合适的容器类别，我们应该考虑预期最常见的操作，然后选择性能特征最符合这些操作的容器。以下列出了最常见的顺序，从快到慢依次为：O(1)，

$O(\log n), O(n), O(n \log n), O(n^2), O(n^k)$ for $k > 2$.

我们还应该考虑容器的内存布局和使用特性。例如，数组（例如 int a[5] 或 std::vector ）将其元素连续存储在内存中，除了元素本身之外，不需要任何其他开销存储空间。（注意

take as a function of the number of elements n in the container:

$$T = f(n).$$

Rather than try to find the exact function f, we concern ourselves only with finding the overall *order* of the function. For example, if the actual theoretical function were any of the following,

$$T = 5n^2 + 17,$$

$$T = 102n^2 + 50n + 12,$$
$$T = \tfrac{1}{2}n^2,$$

we would, in all cases, simplify the expression down to its most relevant term—in this case n^2. To indicate that we are only stating the *order* of the function, not its exact equation, we use “big O” notation and write

$$T = O(n^2).$$

The order of an algorithm can usually be determined via an inspection of the pseudocode. If the algorithm’s execution time is not dependent upon the number of elements in the container at all, we say it is $O(1)$ (i.e., it completes in *constant time*). If the algorithm performs a *loop* over the elements in the container and visits each element once, such as in a linear search of an unsorted list, we say the algorithm is $O(n)$. If two loops are nested, each of which potentially visits each node once, then we say the algorithm is $O(n^2)$. If a divide-and-conquer approach is used, as in a *binary search* (where half of the list is eliminated at each step), then we would expect that only $\lfloor \log_2(n) + 1 \rfloor$ elements will actually be visited by the algorithm in the worst case, and hence we refer to it as an $O(\log n)$ operation. If an algorithm executes a subalgorithm n times, and the subalgorithm is $O(\log n)$, then the resulting algorithm would be $O(n \log n)$.

To select an appropriate container class, we should look at the operations that we expect to be most common, then select the container whose performance characteristics for those operations are most favorable. The most common orders you’ll encounter are listed here from fastest to slowest: $O(1)$, $O(\log n), O(n), O(n \log n), O(n^2), O(n^k)$ for $k > 2$.

We should also take the memory layout and usage characteristics of our containers into account. For example, an array (e.g., `int a[5]` or `std::vector`) stores its elements *contiguously* in memory and requires *no overhead storage* for anything other than the elements themselves. (Note that

动态数组确实需要少量的固定开销。）另一方面，链表（例如 std::list ）将每个元素包装在一个“链接”数据结构中，该结构包含指向下一个元素的指针，也可能包含指向前一个元素的指针，在 64 位机器上每个元素总共最多 16 个字节的开销。此外，链表中的元素在内存中不需要连续，而且通常也不是。连续的内存块通常比一组不同的内存块更适合缓存。因此，对于高速算法，数组在缓存性能方面通常比链表更好（除非链表的节点本身是从一小块连续的内存块中分配的）。但对于插入和删除元素的速度至关重要的情况，链表更适合。

6.3.4 构建自定义容器类

许多游戏引擎提供了自己定制的通用容器数据结构实现。这种做法在主机游戏引擎以及面向手机和 PDA 平台的游戏中尤为普遍。自行构建这些类的原因包括：

- 完全控制。您可以控制数据结构的内存需求、使用的算法、何时以及如何分配内存等。
- 优化机会。您可以优化数据结构和算法，以利用特定于目标控制台的硬件功能；或者针对引擎中的特定应用程序对其进行微调。
- 可定制性。您可以提供 C++ 标准库或 Boost 等第三方库中不流行的自定义算法（例如，在容器中搜索 n 个最相关的元素，而不仅仅是单个最相关的元素）。
- 消除外部依赖。由于您自行构建了软件，因此无需依赖任何其他公司或团队来维护。如果出现问题，可以立即进行调试和修复，而不必等到库的下一个版本（这可能要等到您游戏发布之后！）。
- 控制并发数据结构。编写自己的容器类时，您可以完全控制在多线程或多核系统上保护它们免受并发访问的方式。例如，在 PS4 上，顽皮狗对大多数并发数据结构使用轻量级“自旋锁”互斥锁，因为它们与我们基于光纤的作业调度系统配合良好。第三方容器库可能无法提供这种灵活性。

a *dynamic* array does require a small fixed overhead.) On the other hand, a linked list (e.g., `std::list`) wraps each element in a “link” data structure that contains a pointer to the next element and possibly also a pointer to the previous element, for a total of up to 16 bytes of overhead per element on a 64-bit machine. Also, the elements in a linked list need not be contiguous in memory and often aren't. A contiguous block of memory is usually much more cache-friendly than a set of disparate memory blocks. Hence, for high-speed algorithms, arrays are usually better than linked lists in terms of cache performance (unless the nodes of the linked list are themselves allocated from a small, contiguous block of memory). But a linked list is better for situations in which speed of inserting and removing elements is of prime importance.

6.3.4 Building Custom Container Classes

Many game engines provide their own custom implementations of the common container data structures. This practice is especially prevalent in console game engines and games targeted at mobile phone and PDA platforms. The reasons for building these classes yourself include:

- *Total control*. You control the data structure's memory requirements, the algorithms used, when and how memory is allocated, etc.
- *Opportunities for optimization*. You can optimize your data structures and algorithms to take advantage of hardware features specific to the console(s) you are targeting; or fine-tune them for a particular application within your engine.
- *Customizability*. You can provide custom algorithms not prevalent in the C++ standard library or third-party libraries like Boost (for example, searching for the *n* most relevant elements in a container, instead of just the single most relevant).
- *Elimination of external dependencies*. Since you built the software yourself, you are not beholden to any other company or team to maintain it. If problems arise, they can be debugged and fixed immediately rather than waiting until the next release of the library (which might not be until after you have shipped your game!)
- *Control over concurrent data structures*. When you write your own container classes, you have full control over the means by which they are protected against concurrent access on a multithreaded or multicore system. For example, on the PS4, Naughty Dog uses lightweight “spin lock” mutexes for the majority of our concurrent data structures, because they work well with our fiber-based job scheduling system. A third-party container library might not have given us this kind of flexibility.

我们无法在这里涵盖所有可能的数据结构，但让我们看一下游戏引擎程序员处理容器的几种常见方法。

6.3.4.1 建还是不建

我们不会在这里讨论如何实现所有这些数据类型和算法的细节——关于这方面的书籍和在线资源已经非常丰富。然而，我们将关注如何获取所需类型和算法的实现。作为游戏引擎设计师，我们基本上有三种选择：

1. 手动构建所需的数据结构。
2. 利用C++标准库提供的STL风格的容器。
3. 依赖第三方库，例如 Boost（http://www.boost.org）。

C++ 标准库和 Boost 等第三方库都是不错的选择，因为它们提供了丰富而强大的容器类，几乎涵盖了所有可以想到的数据结构类型。此外，这些库还提供了一套强大的基于模板的泛型算法——一些常见算法的实现，例如在容器中查找元素，几乎可以应用于任何类型的数据对象。然而，这些实现可能并不适用于某些类型的游戏引擎。让我们花点时间探讨一下每种方法的优缺点。

C++ 标准库

C++ 标准库的 STL 风格容器类的优点包括：

- 它们提供了丰富的功能。
- 它们的实现非常强大并且完全可移植。

然而，这些容器类也有一些缺点，包括：

- 头文件很隐晦，难以理解（尽管文档相当不错）。
- 通用容器类通常比专门为解决特定问题而设计的数据结构慢。
- 通用容器可能比定制设计的数据结构消耗更多的内存。
- C++ 标准库进行了大量动态内存分配，有时很难以适合高性能、内存有限的控制台游戏的方式控制其对内存的需求。

We cannot cover all possible data structures here, but let's look at a few common ways in which game engine programmers tend to tackle containers.

6.3.4.1 To Build or Not to Build

We will not discuss the details of how to implement all of these data types and algorithms here—a plethora of books and online resources are available for that purpose. However, we *will* concern ourselves with the question of where to obtain implementations of the types and algorithms that you need. As game engine designers, we have basically three choices:

1. Build the needed data structures manually.
2. Make use of the STL-style containers provided by the C++ standard library.
3. Rely on a third-party library such as Boost (http://www.boost.org).

The C++ standard library and third-party libraries like Boost are attractive options, because they provide a rich and powerful set of container classes covering pretty much every type of data structure imaginable. In addition, these packages provide a powerful suite of template-based *generic algorithms*—implementations of common algorithms, such as finding an element in a container, which can be applied to virtually any type of data object. However, these implementations may not be appropriate for some kinds of game engines. Let's take a moment to investigate some of the pros and cons of each approach.

The C++ Standard Library

The benefits of the C++ standard library's STL-style container classes include:

- They offer a rich set of features.
- Their implementations are robust and fully portable.

However, these container classes also have some drawbacks, including:

- The header files are cryptic and difficult to understand (although the documentation is quite good).
- General-purpose container classes are often slower than a data structure that has been crafted specifically to solve a particular problem.
- A generic container may consume more memory than a custom-designed data structure.
- The C++ standard library does a lot of dynamic memory allocation, and it's sometimes challenging to control its appetite for memory in a way that is suitable for high-performance, memory-limited console games.

• 标准 C++ 库提供的模板分配器系统不够灵活，无法允许这些容器与某些类型的内存分配器一起使用，例如基于堆栈的分配器（参见第 6.2.1.1 节）。

PC 版《荣誉勋章：太平洋突击》引擎大量使用了当时被称为标准模板库 (STL) 的库。虽然 MOHPA 确实存在帧率问题，但团队设法解决了 STL 带来的性能问题（主要是通过严格限制和控制其使用）。OGRE 是一个流行的面向对象渲染库，本书中的一些示例就使用了它，它也大量使用了 STL 风格的容器。然而，顽皮狗禁止在游戏运行时代码中使用 STL 容器（尽管我们允许在离线工具代码中使用它们）。您的情况可能有所不同：在游戏引擎项目中使用 C++ 标准库提供的 STL 风格容器当然可行，但应谨慎使用。

促进

Boost 项目最初由 C++ 标准委员会库工作组成员发起，但现□□在是一个开源项目，拥有来自世界各地的众多贡献者。该项目的目标是开发能够扩展标准 C++ 库并与之协同工作的库，供商业和非商业用途使用。自 C++11 起，Boost 的许多库已被纳入 C++ 标准库，标准委员会的库技术报告 (TR2) 中也包含了更多组件，这标志着 Boost 朝着成为未来 C++ 标准迈出了一步。以下是 Boost 的简要概述：

• Boost 提供了许多 C++ 标准库中没有的有用功能。

• 在某些情况下，Boost 为 C++ 标准库中某些类的设计或实现问题提供了替代方案或解决方法。

• Boost 在处理一些非常复杂的问题（例如智能指针）方面表现出色。（请记住，智能指针非常复杂，而且会严重影响性能。通常情况下，使用句柄更为可取；详情请参阅 16.5 节。）

• Boost 库的文档通常非常出色。文档不仅解释了每个库的功能和使用方法，而且

- The templated allocator system provided by the standard C++ library isn't flexible enough to allow these containers to be used with certain kinds of memory allocators, such as stack-based allocators (see Section 6.2.1.1).

The *Medal of Honor: Pacific Assault* engine for the PC made heavy use of what was known at the time as the standard template library (STL). And while *MOHPA* did have its share of frame rate problems, the team was able to work around the performance problems caused by STL (primarily by carefully limiting and controlling its use). OGRE, the popular object-oriented rendering library that we use for some of the examples in this book, also makes heavy use of STL-style containers. However, at Naughty Dog we prohibit the use of STL containers in runtime game code (although we do permit their use in offline tools code). Your mileage may vary: Using the STL-style containers provided by the C++ standard library on a game engine project is certainly feasible, but it should be used with care.

Boost

The Boost project was started by members of the C++ Standards Committee Library Working Group, but it is now an open source project with many contributors from across the globe. The aim of the project is to produce libraries that extend and work together with the standard C++ library, for both commercial and non-commercial use. Many of the Boost libraries have already been included in the C++ standard library as of C++11, and more components are included in the Standards Committee's Library Technical Report (TR2), which is a step toward becoming part of a future C++ standard. Here is a brief summary of what Boost brings to the table:

- Boost provides a lot of useful facilities not available in the C++ standard library.
- In some cases, Boost provides alternatives or work-arounds for problems with the design or implementation of some classes in the C++ standard library.
- Boost does a great job of handling some very complex problems, like smart pointers. (Bear in mind that smart pointers are complex beasts, and they can be performance hogs. Handles are usually preferable; see Section 16.5 for details.)
- The Boost libraries' documentation is usually excellent. Not only does the documentation explain what each library does and how to use it, but

在大多数情况下，它还对构建库时的设计决策、约束和需求进行了深入的讨论。因此，阅读 Boost 文档是学习软件设计原则的好方法。

如果您已经在使用 C++ 标准库，那么 Boost 可以作为其许多功能的出色扩展和/或替代方案。但是，请注意以下注意事项：

- 大多数核心 Boost 类都是模板，因此使用它们只需要一组合适的头文件。然而，一些 Boost 库会生成相当大的 .lib 文件，可能不适合在小型游戏项目中使用。
- 虽然全球 Boost 社区是一个优秀的支持网络，但 Boost 库本身并不提供任何保证。如果您遇到错误，最终还是需要您的团队自行解决或修复。
- Boost 库根据 Boost 软件许可证分发。
  仔细阅读许可信息（http://www.boost.org/more/license_info.html）以确保它适合您的引擎。

蠢事

Folly 是由 Andrei Alexandrescu 和 Facebook 工程师开发的一个开源库。它的目标是扩展标准 C++ 库和 Boost 库（而不是与这些库竞争），并注重易用性和高性能软件的开发。您可以通过在线搜索题为“Folly：Facebook 开源库”的文章来了解相关信息，该文章托管在 https://www.facebook.com/ 上。该库本身可以在 GitHub 上找到：https://github.com/facebook/folly。

洛基

C++ 编程中有一个相当深奥的分支，称为模□□板元编程。其核心思想是利用 C++ 的模板特性，让编译器完成许多原本需要在运行时完成的工作，实际上是“欺骗”编译器去做一些它原本设计时没有考虑的事情。这可以带来一些非常强大且实用的编程工具。

迄今为止，最著名、可能也是最强大的 C++ 模板元编程库是 Loki，它是由 Andrei 设计和编写的库

in most cases it also provides an excellent in-depth discussion of the design decisions, constraints and requirements that went into constructing the library. As such, reading the Boost documentation is a great way to learn about the principles of software design.

If you are already using the C++ standard library, then Boost can serve as an excellent extension and/or alternative to many of its features. However, be aware of the following caveats:

- Most of the core Boost classes are templates, so all that one needs in order to use them is the appropriate set of header files. However, some of the Boost libraries build into rather large .lib files and may not be feasible for use in very small-scale game projects.
- While the worldwide Boost community is an excellent support network, the Boost libraries come with no guarantees. If you encounter a bug, it will ultimately be your team's responsibility to work around it or fix it.
- The Boost libraries are distributed under the Boost Software License. Read the license information (http://www.boost.org/more/license_info.html) carefully to be sure it is right for your engine.

Folly

Folly is an open source library developed by Andrei Alexandrescu and the engineers at Facebook. Its goal is to extend the standard C++ library and the Boost library (rather than to compete with these libraries), with an emphasis on ease of use and the development of high-performance software. You can read about it by searching online for the article entitled "Folly: The Facebook Open Source Library" which is hosted at https://www.facebook.com/. The library itself is available on GitHub here: https://github.com/facebook/folly.

Loki

There is a rather esoteric branch of C++ programming known as *template metaprogramming*. The core idea is to use the compiler to do a lot of the work that would otherwise have to be done at runtime by exploiting the template feature of C++ and in effect "tricking" the compiler into doing things it wasn't originally designed to do. This can lead to some startlingly powerful and useful programming tools.

By far the most well-known and probably most powerful template metaprogramming library for C++ is Loki, a library designed and written by Andrei

Alexandrescu（其主页为 http://www.erdani.org）。该库可从 SourceForge 获取，网址为 http://loki-lib.sourceforge.net。

Loki 功能极其强大；它的代码体系引人入胜，值得研究和学习。然而，它的两个主要弱点与实际操作息息相关：(a) 它的代码阅读和使用起来可能令人望而生畏，更不用说真正理解了；(b) 它的一些组件依赖于利用编译器的“副作用”行为，这些行为需要仔细定制才能在新的编译器上运行。因此，Loki 的使用可能有些困难，而且它的可移植性不如一些“功能不那么强大”的同类产品。Loki 并不适合胆小的人。话虽如此，Loki 的一些概念，例如基于策略的编程，可以应用于任何 C++ 项目，即使您不使用 Loki 库本身。我强烈建议所有软件工程师阅读 Andrei 的开创性著作《现代 C++ 设计》[3]，Loki 库就是由此诞生的。

6.3.5 动态数组和块分配

固定大小的 C 风格数组在游戏编程中被广泛使用，因为它们不需要内存分配，是连续的，因此对缓存友好，并且支持许多常见操作，例如附加数据和非常有效地搜索。

当无法预先确定数组的大小时，程序员往往会选择使用链表或动态数组。如果我们希望保持定长数组的性能和内存特性，那么动态数组通常是首选的数据结构。

实现动态数组的最简单方法是首先分配一个 n 个元素的缓冲区，然后仅当尝试向其中添加超过 n 个元素时才扩大列表。这为我们提供了固定大小数组的优点，但没有上限。通过分配一个新的更大的缓冲区，将数据从原始缓冲区复制到新缓冲区，然后释放原始缓冲区来实现增长。缓冲区的大小以某种有序的方式增加，例如每次增长时向其中添加 n，或者每次增长时将其翻倍。我遇到的大多数实现都不会缩小数组，只会增长它（除了将数组清除为零大小这个值得注意的例外，这可能会或可能不会释放缓冲区）。因此，数组的大小成为一种“高水位线”。std::vector 类就是这样工作的。

当然，如果你能为数据设定一个高水位线，那么在引擎启动时只分配一个该大小的缓冲区可能更好。由于重新分配和数据复制的开销，动态数组的增长可能会非常昂贵。这些因素的影响取决于

Alexandrescu (whose home page is at http://www.erdani.org). The library can be obtained from SourceForge at http://loki-lib.sourceforge.net.

Loki is extremely powerful; it is a fascinating body of code to study and learn from. However, its two big weaknesses are of a practical nature: (a) its code can be daunting to read and use, much less truly understand, and (b) some of its components are dependent upon exploiting “side-effect” behaviors of the compiler that require careful customization in order to be made to work on new compilers. So Loki can be somewhat tough to use, and it is not as portable as some of its “less-extreme” counterparts. Loki is not for the faint of heart. That said, some of Loki’s concepts such as *policy-based programming* can be applied to any C++ project, even if you don’t use the Loki library per se. I highly recommend that all software engineers read Andrei’s ground-breaking book, *Modern C++ Design* [3], from which the Loki library was born.

6.3.5 Dynamic Arrays and Chunky Allocation

Fixed size C-style arrays are used quite a lot in game programming, because they require no memory allocation, are contiguous and hence cache-friendly, and support many common operations such as appending data and searching very efficiently.

When the size of an array cannot be determined a priori, programmers tend to turn either to *linked lists* or *dynamic arrays*. If we wish to maintain the performance and memory characteristics of fixed-length arrays, then the dynamic array is often the data structure of choice.

The easiest way to implement a dynamic array is to allocate an n-element buffer initially and then *grow* the list only if an attempt is made to add more than n elements to it. This gives us the favorable characteristics of a fixed size array but with no upper bound. Growing is implemented by allocating a new larger buffer, copying the data from the original buffer into the new buffer, and then freeing the original buffer. The size of the buffer is increased in some orderly manner, such as adding n to it on each grow, or doubling it on each grow. Most of the implementations I’ve encountered never shrink the array, only grow it (with the notable exception of clearing the array to zero size, which might or might not free the buffer). Hence the size of the array becomes a sort of “high water mark.” The `std::vector` class works in this manner.

Of course, if you can establish a high water mark for your data, then you’re probably better off just allocating a single buffer of that size when the engine starts up. Growing a dynamic array can be incredibly costly due to reallocation and data copying costs. The impact of these things depends on the sizes of the

涉及缓冲区。增长还可能导致在丢弃的缓冲区被释放时产生碎片。因此，与所有分配内存的数据结构一样，使用动态数组时必须谨慎。动态数组可能最适合在开发过程中使用，因为此时您尚不确定所需的缓冲区大小。一旦确定了合适的内存预算，它们随时可以转换为固定大小的数组。

6.3.6 字典和哈希表

字典是由键值对组成的表。给定键，可以快速查找字典中的值。键和值可以是任何数据类型。这种数据结构通常以二叉搜索树或哈希表的形式实现。

在二叉树实现中，键值对存储在二叉树的节点中，并且树的维护顺序是按键排序的。按键查找值需要执行 O(log n) 的二分查找。

在哈希表实现中，值存储在固定大小的表中，其中表中的每个槽位代表一个或多个键。要将键值对插入哈希表，首先通过称为哈希的过程将键转换为整数形式（如果它还不是整数）。然后，通过将哈希键对表的大小取模来计算哈希表的索引。最后，将键值对存储在与该索引对应的槽位中。回想一下，模运算符（C/C++ 中的 % ）求整数键除以表大小后的余数。因此，如果哈希表有五个槽位，则键 3 将存储在索引 3 处（ 3 % 5 == 3 ），而键 6 将存储在索引 1 处（ 6 % 5 == 1 ）。在没有冲突的情况下，查找键值对是 O(1) 操作。

6.3.6.1 冲突：开放哈希表和封闭哈希表

有时，两个或多个键最终会占据哈希表中的同一个位置。这被称为冲突。解决冲突的基本方法有两种，从而产生了两种不同类型的哈希表：

- 开放式。在开放式哈希表中（参见图 6.8），冲突的解决方式是在每个索引处存储多个键值对（通常以链表的形式）。这种方法易于实现，并且对可存储的键值对数量没有上限。但是，每当向表中添加新的键值对时，都需要动态分配内存。

- 封闭式。在封闭式哈希表中（见图 6.9），冲突通过探测来解决，直到找到一个空槽。（“探测”指的是

buffers involved. Growing can also lead to fragmentation when discarded buffers are freed. So, as with all data structures that allocate memory, caution must be exercised when working with dynamic arrays. Dynamic arrays are probably best used during development, when you are as yet unsure of the buffer sizes you'll require. They can always be converted into fixed size arrays once suitable memory budgets have been established.

6.3.6 Dictionaries and Hash Tables

A dictionary is a table of key-value pairs. A value in the dictionary can be looked up quickly, given its key. The keys and values can be of any data type. This kind of data structure is usually implemented either as a binary search tree or as a hash table.

In a binary tree implementation, the key-value pairs are stored in the nodes of the binary tree, and the tree is maintained in key-sorted order. Looking up a value by key involves performing an $O(\log n)$ binary search.

In a hash table implementation, the values are stored in a fixed size table, where each slot in the table represents one or more keys. To insert a key-value pair into a hash table, the key is first converted into integer form via a process known as *hashing* (if it is not already an integer). Then an *index* into the hash table is calculated by taking the hashed key *modulo* the size of the table. Finally, the key-value pair is stored in the slot corresponding to that index. Recall that the *modulo* operator (`%` in C/C++) finds the remainder of dividing the integer key by the table size. So if the hash table has five slots, then a key of 3 would be stored at index 3 (`3 % 5 == 3`), while a key of 6 would be stored at index 1 (`6 % 5 == 1`). Finding a key-value pair is an $O(1)$ operation in the absence of collisions.

6.3.6.1 Collisions: Open and Closed Hash Tables

Sometimes two or more keys end up occupying the same slot in the hash table. This is known as a *collision*. There are two basic ways to *resolve* a collision, giving rise to two different kinds of hash tables:

- *Open*. In an open hash table (see Figure 6.8), collisions are resolved by simply storing more than one key-value pair at each index, usually in the form of a linked list. This approach is easy to implement and imposes no upper bound on the number of key-value pairs that can be stored. However, it does require memory to be allocated dynamically whenever a new key-value pair is added to the table.
- *Closed*. In a closed hash table (see Figure 6.9), collisions are resolved via a process of *probing* until a vacant slot is found. (“Probing” means ap-

使用定义明确的算法来搜索空闲的槽位。这种方法实现起来稍微困难一些，并且对表中可容纳的键值对的数量设置了上限（因为每个槽位只能容纳一个键值对）。但这种哈希表的主要优点是它占用的内存量是固定的，不需要动态内存分配。因此，在控制台引擎中，它通常是一个不错的选择。

令人困惑的是，封闭哈希表有时被称为使用开放寻址，而开放哈希表则被称为使用一种称为链接的寻址方法，由于表中每个插槽处都有链接列表而得名。

6.3.6.2 哈希

哈希处理是将任意数据类型的键转换为整数的过程，该整数可以作为表的索引，对表大小取模。从数学上讲，给定一个键 k，我们希望使用哈希函数 H 生成一个整数哈希值 h，然后找到表中的索引 i，如下所示：

$$h = H(k),$$
$$i = h \bmod N,$$

其中 N 是表中的槽数，符号 mod 表示模运算，即求商 h / N 的余数。

如果键是唯一的整数，则哈希函数可以是恒等函数，H(k)=k。如果键是唯一的32位浮点数，则哈希函数可能只是将32位浮点数的位模式重新解释为32位整数。

```
U32 hashFloat(float f)
{
```

图 6.8. 开放哈希表。

plying a well-defined algorithm to search for a free slot.) This approach is a bit more difficult to implement, and it imposes an upper limit on the number of key-value pairs that can reside in the table (because each slot can hold only one key-value pair). But the main benefit of this kind of hash table is that it uses up a fixed amount of memory and requires no dynamic memory allocation. Therefore, it is often a good choice in a console engine.

Confusingly, closed hash tables are sometimes said to use *open addressing*, while open hash tables are said to use an addressing method known as *chaining*, so named due to the linked lists at each slot in the table.

6.3.6.2 Hashing

Hashing is the process of turning a key of some arbitrary data type into an integer, which can be used modulo the table size as an index into the table. Mathematically, given a key k, we want to generate an integer hash value h using the hash function H and then find the index i into the table as follows:

$$h = H(k),$$
$$i = h \bmod N,$$

where N is the number of slots in the table, and the symbol mod represents the *modulo* operation, i.e., finding the remainder of the quotient h/N.

If the keys are unique integers, the hash function can be the identity function, $H(k) = k$. If the keys are unique 32-bit floating-point numbers, a hash function might simply reinterpret the bit pattern of the 32-bit float as if it were a 32-bit integer.

```
U32 hashFloat(float f)
{
```

Figure 6.8. An open hash table.

图 6.9。封闭的哈希表。

```
    union
    {
        float m_asFloat;
        U32   m_asU32;
    } u;

    u.m_asFloat = f;
    return u.m_asU32;
}
```

如果键是字符串，我们可以采用字符串哈希函数，将字符串中所有字符的 ASCII 或 UTF 代码组合成一个 32 位整数值。

哈希函数 H(k) 的质量对哈希表的效率至关重要。一个“好的”哈希函数是指能够将所有有效键的集合均匀分布在哈希表中，从而最大限度地降低冲突可能性的函数。哈希函数还必须计算速度足够快，并且具有确定性，即每次使用相同的输入调用它时，它都必须产生完全相同的输出。

字符串可能是你遇到的最常见的密钥类型，因此了解一个“好的”字符串哈希函数特别有帮助。表 6.1 列出了一些著名的哈希算法、它们的吞吐量评级（基于基准测量，然后转换为低、中或高评级）以及它们在 SMHasher 测试 (https://github.com/aappleby/smhasher) 中的得分。请注意，表中列出的相对吞吐量仅用于粗略比较。许多因素都会影响哈希函数的吞吐量，包括运行它的硬件和输入数据的属性。加密哈希函数的速度故意设计得很慢，因为它们的重点是生成一个极不可能与其他输入字符串的哈希值冲突的哈希值，并且确定

Figure 6.9. A closed hash table.

```
    union
    {
        float m_asFloat;
        U32   m_asU32;
    } u;

    u.m_asFloat = f;
    return u.m_asU32;
}
```

If the key is a string, we can employ a *string hashing function*, which combines the ASCII or UTF codes of all the characters in the string into a single 32-bit integer value.

The *quality* of the hashing function $H(k)$ is crucial to the efficiency of the hash table. A “good” hashing function is one that distributes the set of all valid keys evenly across the table, thereby minimizing the likelihood of collisions. A hash function must also be reasonably quick to calculate, and *deterministic* in the sense that it must produce the exact same output every time it is called with an indentical input.

Strings are probably the most prevalent type of key you’ll encounter, so it’s particularly helpful to know a “good” string hashing function. Table 6.1 lists a number of well-known hashing algorithms, their throughput ratings (based on benchmark measurements and then converted into a rating of Low, Medium or High) and their score on the SMHasher test (https://github.com/aappleby/smhasher). Please note that the relative throughputs listed in the table are for rough comparison purposes only. Many factors contribute to the throughput of a hash function, including the hardware on which it is run and the properties of the input data. Cryptographic hashes are deliberately slow, as their focus is on producing a hash that is extremely unlikely to collide with the hashes of other input strings, and for which the task of determining

姓名	吞吐量	分数	加密?
xx哈希	高的	10	No
MurmurHash 3a	高的	10	No
SBox	中等的	9	不‡
Lookup3	中等的	9	No
CityHash64	中等的	10	No
CRC32	低的	9	No
MD5-32	低的	10	是的
SHA1-32	低的	10	是的

表 6.1. 知名哈希算法在相对吞吐量和 SMHasher 测试中的得分比较。‡请注意，SBox 本身并不是加密哈希，而是密码学中使用的对称密钥算法的一个组成部分。

产生给定哈希值的字符串在计算上极其困难。

有关哈希函数的更多信息，请参阅 Paul Hsieh 撰写的精彩文章，网址为 http://www.azillionmonkeys.com/qed/hash.html。

6.3.6.3 实现封闭哈希表

在封闭式哈希表中，键值对直接存储在表中，而不是存储在每个表条目的链表中。这种方法允许程序员预先定义哈希表将使用的确切内存量。当遇到冲突时，就会出现问题——两个键最终想要存储在表中的同一个位置。为了解决这个问题，我们使用了一个称为探测的过程。

最简单的方法是线性探测。假设我们的哈希函数已经得到了一个表索引 i，但是该位置已经被占用；我们只需尝试（i + 1）、（i + 2）等等位置，直到找到一个空位置（当 i = N 时，绕回到表的开头）。线性探测的另一种变体是交替向前和向后搜索（i + 1）、（i - 1）、（i + 2）、（i - 2）等等，确保将结果索引取模到表的有效范围内。

线性探测容易导致键值对“聚集”。为了避免这种情况，我们可以使用一种称为二次探测的算法。我们从已占用的表索引 i 开始，使用探测序列 ij = （i ± j 2），其中 j = 1, 2, 3, . . . 换句话说，我们尝试（i + 1 2），（i - 1 2），（i + 2 2），（i - 2 2）等等。

Name	Throughput	Score	Cryptographic?
xxHash	High	10	No
MurmurHash 3a	High	10	No
SBox	Medium	9	No‡
Lookup3	Medium	9	No
CityHash64	Medium	10	No
CRC32	Low	9	No
MD5-32	Low	10	Yes
SHA1-32	Low	10	Yes

Table 6.1. Comparison of well-known hashing algorithms in terms of their relative throughput and their scores on the SMHasher test. ‡Note that SBox is not itself a cryptographic hash, but it is one component of the symmetric key algorithms used in cryptography.

a string that would produce a given hash value is extremely computationally difficult.

For more information on hash functions, see the excellent article by Paul Hsieh available at http://www.azillionmonkeys.com/qed/hash.html.

6.3.6.3 Implementing a Closed Hash Table

In a closed hash table, the key-value pairs are stored directly in the table, rather than in a linked list at each table entry. This approach allows the programmer to define a priori the exact amount of memory that will be used by the hash table. A problem arises when we encounter a *collision*—two keys that end up wanting to be stored in the same slot in the table. To address this, we use a process known as *probing*.

The simplest approach is *linear probing*. Imagine that our hashing function has yielded a table index of i, but that slot is already occupied; we simply try slots $(i+1)$, $(i+2)$ and so on until an empty slot is found (wrapping around to the start of the table when $i = N$). Another variation on linear probing is to alternate searching forwards and backwards, $(i+1)$, $(i-1)$, $(i+2)$, $(i-2)$ and so on, making sure to modulo the resulting indices into the valid range of the table.

Linear probing tends to cause key-value pairs to “clump up.” To avoid these clusters, we can use an algorithm known as *quadratic probing*. We start at the occupied table index i and use the sequence of probes $i_j = (i \pm j^2)$ for $j = 1, 2, 3, \ldots$. In other words, we try $(i+1^2)$, $(i-1^2)$, $(i+2^2)$, $(i-2^2)$ and so

记住始终将结果索引模到表的有效范围内。

使用封闭哈希时，最好将表大小设为质数。将质数表大小与二次探测结合使用，往往能够以最小的聚类程度实现对可用表槽的最佳覆盖。请参阅 http://stackoverflow.com/questions/1145217/why-should-hash

-functions-use-a-prime-number-modulus 很好地讨论了为什么素数哈希表大小更可取。

6.3.6.4 罗宾汉哈希

Robin Hood 哈希是另一种用于封闭哈希表的探测方法，近年来越来越流行。这种探测方案可以提高封闭哈希表的性能，即使哈希表几乎已满。有关 Robin Hood 哈希工作原理的详细讨论，请参阅 https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/。

6.4 字符串

字符串几乎在每个软件项目中都随处可见，游戏引擎也不例外。表面上看，字符串似乎是一种简单、基本的数据类型。但是，当你开始在项目中使用字符串时，你很快就会发现各种各样的设计问题和限制，所有这些都必须仔细考虑。

6.4.1 字符串的问题

关于字符串最根本的问题是它们应该如何在程序中存储和管理。在 C 和 C++ 中，字符串甚至不是原子类型——它们是作为字符数组实现的。字符串的长度可变意味着我们要么必须对字符串的大小进行硬编码限制，要么需要动态分配字符串缓冲区。

另一个与字符串相关的大问题是本地化——即调整软件以支持其他语言版本的过程。这也称为国际化，简称 I18N。任何以英语显示给用户的字符串都必须翻译成您计划支持的任何语言。（当然，程序内部使用但从未显示给用户的字符串无需本地化。）这不仅涉及确保您能够表示您计划支持的所有语言的所有字符字形（通过一组合适的字体），还意味着确保您的游戏能够处理不同的文本方向。例如，传统的

on, remembering to always modulo the resulting index into the valid range of the table.

When using closed hashing, it is a good idea to make your table size a *prime number*. Using a prime table size in conjunction with quadratic probing tends to yield the best coverage of the available table slots with minimal clustering. See http://stackoverflow.com/questions/1145217/why-should-hash-functions-use-a-prime-number-modulus for a good discussion of why prime hash table sizes are preferable.

6.3.6.4 Robin Hood Hashing

Robin Hood hashing is another probing method for closed hash tables that has gained popularity recently. This probing scheme improves the performance of a closed hash table, even when the table is nearly full. For a good discussion of how Robin Hood hashing works, see https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/.

6.4 Strings

Strings are ubiquitous in almost every software project, and game engines are no exception. On the surface, the string may seem like a simple, fundamental data type. But, when you start using strings in your projects, you will quickly discover a wide range of design issues and constraints, all of which must be carefully accounted for.

6.4.1 The Problem with Strings

The most fundamental question about strings is how they should be stored and managed in your program. In C and C++, strings aren't even an atomic type—they are implemented as *arrays of characters*. The variable length of strings means we either have to hard-code limitations on the sizes of our strings, or we need to dynamically allocate our string buffers.

Another big string-related problem is that of *localization*—the process of adapting your software for release in other languages. This is also known as *internationalization*, or I18N for short. Any string that you display to the user in English must be translated into whatever languages you plan to support. (Strings that are used internally to the program but are never displayed to the user are exempt from localization, of course.) This not only involves making sure that you can represent all the character glyphs of all the languages you plan to support (via an appropriate set of fonts), but it also means ensuring that your game can handle different text orientations. For example, traditional

中文文本是竖排的，而不是横排的（尽管现代中文和日语通常是横排的，并且是从左到右书写的），而有些语言，比如希伯来语，则是从右到左阅读的。你的游戏还需要妥善处理翻译后的字符串可能比英文字符串长很多或短很多的情况。

最后，需要注意的是，字符串在游戏引擎内部用于资源文件名和对象 ID 等信息。例如，当游戏设计师布置关卡时，使用有意义的名称（例如“PlayerCamera”、“enemy-tank-01”或“explosionTrigger”）来识别关卡中的对象会非常方便。

我们的引擎如何处理这些内部字符串通常会对游戏性能产生深远的影响。这是因为字符串在运行时处理起来开销巨大。比较或复制 int 或 float 可以通过简单的机器语言指令完成。另一方面，比较字符串需要使用类似 strcmp() 的函数（其中 n 是字符串的长度）对字符数组进行 O(n) 的扫描。复制字符串需要进行 O(n) 的内存复制，更不用说可能需要为复制动态分配内存。在我参与的一个项目中，我们分析了游戏的性能，结果发现 strcmp() 和 strcpy() 是开销最大的两个函数！通过消除不必要的字符串操作并使用本节概述的一些技术，我们几乎能够从性能分析中消除这些函数，并显著提高游戏的帧速率。（我从许多不同工作室的开发人员那里听到过类似的故事。）

6.4.2 字符串类

许多 C++ 程序员更喜欢使用字符串类，例如 C++ 标准库的 std::string，而不是直接处理字符数组。这样的类可以使程序员更方便地使用字符串。但是，字符串类可能具有隐藏的成本，这些成本在游戏分析之前很难发现。例如，使用 C 样式字符数组将字符串传递给函数很快，因为第一个字符的地址通常在硬件寄存器中传递。另一方面，如果函数声明或使用不当，传递字符串对象可能会产生一个或多个复制构造函数的开销。复制字符串可能涉及动态内存分配，导致看似无害的函数调用最终花费数千个机器周期。

由于字符串类存在诸多问题，我通常倾向于在游戏运行时代码中避免使用它们。但是，如果你强烈希望使用字符串

Chinese text is oriented vertically instead of horizontally (although modern Chinese and Japanese are commonly written horizontally and left-to-right), and some languages like Hebrew read right-to-left. Your game also needs to gracefully deal with the possibility that a translated string will be either much longer or much shorter than its English counterpart.

Finally, it's important to realize that strings are used internally within a game engine for things like resource file names and object ids. For example, when a game designer lays out a level, it's highly convenient to permit him or her to identify the objects in the level using meaningful names, like "PlayerCamera," "enemy-tank-01" or "explosionTrigger."

How our engine deals with these internal strings often has pervasive ramifications on the performance of the game. This is because strings are inherently expensive to work with at runtime. Comparing or copying `ints` or `floats` can be accomplished via simple machine language instructions. On the other hand, comparing strings requires an $O(n)$ scan of the character arrays using a function like `strcmp()` (where n is the length of the string). Copying a string requires an $O(n)$ memory copy, not to mention the possibility of having to dynamically allocate the memory for the copy. During one project I worked on, we profiled our game's performance only to discover that `strcmp()` and `strcpy()` were the top two most expensive functions! By eliminating unnecessary string operations and using some of the techniques outlined in this section, we were able to all but eliminate these functions from our profile and increase the game's frame rate significantly. (I've heard similar stories from developers at a number of different studios.)

6.4.2 String Classes

Many C++ programmers prefer to use a *string class*, such as the C++ standard library's `std::string`, rather than deal directly with character arrays. Such classes can make working with strings much more convenient for the programmer. However, a string class can have hidden costs that are difficult to see until the game is profiled. For example, passing a string to a function using a C-style character array is fast because the address of the first character is typically passed in a hardware register. On the other hand, passing a string object might incur the overhead of one or more copy constructors, if the function is not declared or used properly. Copying strings might involve dynamic memory allocation, causing what looks like an innocuous function call to end up costing literally thousands of machine cycles.

Because of the myriad issues with string classes, I generally prefer to avoid them in runtime game code. However, if you feel a strong urge to use a string

类，请确保选择或实现一个具有可接受的运行时性能特征的类，并确保所有使用它的程序员都了解其成本。了解你的字符串类：它是否将所有字符串缓冲区视为只读？它是否利用了写时复制优化？（请参阅 http://en.wikipedia.org/wiki/Copy-on-write。）在 C++11 中，它是否提供移动构造函数？它是否拥有与字符串关联的内存，或者它可以引用不属于它的内存？（有关字符串类中内存所有权问题的更多信息，请参阅 http://www.boost.org/doc/libs/1_57_0/libs/utility/doc/html/string_ref.html。）根据经验，始终通过引用传递字符串对象，而不要通过值传递（因为后者通常会产生字符串复制成本）。尽早并经常分析你的代码，以确保你的字符串类不会成为帧率下降的主要原因！

在我看来，在存储和管理文件系统路径时，使用专门的字符串类是合理的。在这种情况下，一个假设的 Path 类可以比原始的 C 语言字符数组更有价值。例如，它可以提供从路径中提取文件名、文件扩展名或目录的函数。它还可以通过自动将 Windows 风格的反斜杠转换为 UNIX 风格的正斜杠或其他操作系统的路径分隔符来隐藏操作系统差异。编写一个能够以跨平台方式提供此类功能的 Path 类在游戏引擎环境中可能非常有价值。（有关此主题的更多详细信息，请参阅第 7.1.1.4 节。）

6.4.3 唯一标识符

任何虚拟游戏世界中的对象都需要以某种方式进行唯一标识。例如，在《吃豆人》中，我们可能会遇到名为“pac_man”、“blinky”、“pinky”、“inky”和“clyde”的游戏对象。唯一的对象标识符使游戏设计师能够追踪构成游戏世界的无数对象，并允许引擎在运行时找到并操作这些对象。此外，构建游戏对象的资源（网格、材质、纹理、音频剪辑、动画等等）也都需要唯一的标识符。

字符串似乎是此类标识符的自然选择。资源通常存储在磁盘上的独立文件中，因此通常可以通过其文件路径（当然是字符串）唯一地标识它们。游戏对象是由游戏设计师创建的，因此他们很自然地会为对象分配易于理解的字符串名称，而不是必须记住整数对象索引或 64 位或 128 位全局唯一标识符 (GUID)。然而，唯一标识符之间的比较速度至关重要。

class, make sure you pick or implement one that has acceptable runtime performance characteristics—and be sure all programmers that use it are aware of its costs. Know your string class: Does it treat all string buffers as read-only? Does it utilize the *copy on write* optimization? (See http://en.wikipedia.org/wiki/Copy-on-write.) In C++11, does it provide a move constructor? Does it own the memory associated with the string, or can it reference memory that it does not own? (See http://www.boost.org/doc/libs/1_57_0/libs/utility/doc/html/string_ref.html for more on the issue of memory ownership in string classes.) As a rule of thumb, always pass string objects by reference, never by value (as the latter often incurs string-copying costs). Profile your code early and often to ensure that your string class isn't becoming a major source of lost frame rate!

One situation in which a specialized string class does seem justifiable to me is when storing and managing file system paths. Here, a hypothetical `Path` class could add significant value over a raw C-style character array. For example, it might provide functions for extracting the filename, file extension or directory from the path. It might hide operating system differences by automatically converting Windows-style backslashes to UNIX-style forward slashes or some other operating system's path separator. Writing a `Path` class that provides this kind of functionality in a cross-platform way could be highly valuable within a game engine context. (See Section 7.1.1.4 for more details on this topic.)

6.4.3 Unique Identifiers

The *objects* in any virtual game world need to be uniquely identified in some way. For example, in Pac Man we might encounter game objects named "pac_man," "blinky," "pinky," "inky" and "clyde." Unique object identifiers allow game designers to keep track of the myriad objects that make up their game worlds and also permit those objects to be found and operated on at runtime by the engine. In addition, the *assets* from which our game objects are constructed—meshes, materials, textures, audio clips, animations and so on—all need unique identifiers as well.

Strings seem like a natural choice for such identifiers. Assets are often stored in individual files on disk, so they can usually be identified uniquely by their file paths, which of course are strings. And game objects are created by game designers, so it is natural for them to assign their objects understandable string names, rather than have to remember integer object indices, or 64- or 128-bit globally unique identifiers (GUIDs). However, the speed with which *comparisons* between unique identifiers can be made is of paramount impor-

游戏中的场景，strcmp() 根本不够用。我们需要一种鱼与熊掌兼得的方法——既能获得字符串的描述性和灵活性，又能保持整数的速度。

6.4.3.1 散列字符串 ID

一个好的解决方案是对字符串进行哈希处理。正如我们所见，哈希函数将字符串映射到一个半唯一的整数上。字符串哈希码可以像其他整数一样进行比较，因此比较速度很快。如果我们将实际的字符串存储在哈希表中，那么原始字符串总是可以从哈希码中恢复出来。这对于调试非常有用，并且允许将经过哈希处理的字符串显示在屏幕上或日志文件中。游戏程序员有时会使用术语“字符串 ID”来指代这样的哈希字符串。虚幻引擎使用术语“名称”（由类 FName 实现）。

与任何哈希系统一样，冲突是有可能的（即两个不同的字符串可能最终得到相同的哈希码）。但是，使用合适的哈希函数，我们几乎可以保证在游戏中使用的所有合理输入字符串都不会发生冲突。毕竟，32 位哈希码代表超过 40 亿个可能的值。因此，如果我们的哈希函数能够很好地将字符串均匀分布在这个非常大的范围内，那么就不太可能发生冲突。在顽皮狗，我们开始使用 CRC-32 算法的变体来对字符串进行哈希处理，在《神秘海域》和《最后生还者》的多年开发过程中，我们只遇到了少数几次冲突。当确实发生冲突时，修复它只需稍微修改其中一个字符串即可（例如，在其中一个字符串后附加“2”或“b”，或者使用完全不同但同义的字符串）。话虽如此，顽皮狗已经将《最后生还者第二部》和我们未来的所有游戏的哈希函数转移到 64 位；考虑到我们在任何一款游戏中使用的字符串的数量和典型长度，这应该基本上消除了哈希冲突的可能性。

6.4.3.2 一些实现思路

从概念上讲，对字符串运行哈希函数来生成字符串 ID 非常简单。然而，实际上，重要的是要考虑何时计算哈希值。大多数使用字符串 ID 的游戏引擎都会在运行时进行哈希计算。在顽皮狗，我们允许对字符串进行运行时哈希计算，但我们也使用 C++11 的用户定义字面量功能，在编译时将语法“any_string”_sid 直接转换为哈希整数值。这使得字符串 ID 可以在任何可以使用整数常量的地方使用，包括 switch 语句的常量 case 标签。（在运行时生成字符串 ID 的函数调用的结果不是

tance in a game, so `strcmp()` simply doesn't cut it. We need a way to have our cake and eat it too—a way to get all the descriptiveness and flexibility of a string, but with the speed of an integer.

6.4.3.1 Hashed String Ids

One good solution is to *hash* our strings. As we've seen, a hash function maps a string onto a semi-unique integer. String hash codes can be compared just like any other integers, so comparisons are fast. If we store the actual strings in a hash table, then the original string can always be recovered from the hash code. This is useful for debugging purposes and to permit hashed strings to be displayed on-screen or in log files. Game programmers sometimes use the term *string id* to refer to such a hashed string. The Unreal engine uses the term *name* instead (implemented by class `FName`).

As with any hashing system, *collisions* are a possibility (i.e., two different strings might end up with the same hash code). However, with a suitable hash function, we can all but guarantee that collisions will not occur for all reasonable input strings we might use in our game. After all, a 32-bit hash code represents more than four billion possible values. So, if our hash function does a good job of distributing strings evenly throughout this very large range, we are unlikely to collide. At Naughty Dog, we started out using a variant of the CRC-32 algorithm to hash our strings, and we encountered only a handful of collisions during many years of development on *Uncharted* and *The Last of Us*. And when a collision did occur, fixing it was a simple matter of slightly altering one of the strings (e.g., append a "2" or a "b" to one of the strings, or use a totally different but synonymous string). That being said, Naughty Dog has moved to a 64-bit hashing function for *The Last of Us Part II* and all of our future game titles; this should essentially eliminate the possibility of hash collisions, given the quantity and typical lengths of the strings we use in any one game.

6.4.3.2 Some Implementation Ideas

Conceptually, it's easy enough to run a hash function on your strings in order to generate string ids. Practically speaking, however, it's important to consider *when* the hash will be calculated. Most game engines that use string ids do the hashing at runtime. At Naughty Dog, we permit runtime hashing of strings, but we also use C++11's *user-defined literals* feature to transform the syntax `"any_string"_sid` directly into a hashed integer value at compile time. This permits string ids to be used anywhere that an integer manifest constant can be used, including the constant `case` labels of a `switch` statement. (The result of a function call that generates a string id at runtime is not

一个常量，因此不能用作案例标签。）

从字符串生成字符串 ID 的过程有时被称为“驻留字符串”，因为除了对其进行哈希处理之外，该字符串通常还会被添加到全局字符串表中。这样，以后就可以从哈希码中恢复原始字符串。您可能还希望您的工具能够将字符串哈希处理为字符串 ID。这样，当工具生成供引擎使用的数据时，字符串就已经经过哈希处理了。

驻留字符串的主要问题是操作速度很慢。

哈希函数必须对字符串运行，这可能是一个昂贵的方案，尤其是在需要驻留大量字符串的情况下。此外，还必须为字符串分配内存，并将其复制到查找表中。因此（如果您不是在编译时生成字符串 ID），通常最好只驻留每个字符串一次，并将结果保存起来以备后用。例如，最好编写这样的代码，因为后一种实现会导致每次调用函数 f() 时都无谓地重新驻留字符串。

```
static StringId   sid_foo = internString("foo");
static StringId   sid_bar = internString("bar");

// ...

void f(StringId id)
{
    if (id == sid_foo)
    {
        // handle case of id == "foo"
    }
    else if (id == sid_bar)
    {
        // handle case of id == "bar"
    }
}
```

以下方法效率较低：

```
void f(StringId id)
{
    if (id == internString("foo"))
    {
        // handle case of id == "foo"
    }
```


a constant, so it cannot be used as a `case` label.)

The process of generating a string id from a string is sometimes called *interning* the string, because in addition to hashing it, the string is typically also added to a global string table. This allows the original string to be recovered from the hash code later. You may also want your tools to be capable of hashing strings into string ids. That way, when the tool generates data for consumption by your engine, the strings will already have been hashed.

The main problem with interning a string is that it is a slow operation. The hashing function must be run on the string, which can be an expensive proposition, especially when a large number of strings are being interned. In addition, memory must be allocated for the string, and it must be copied into the lookup table. As a result (if you are not generating string ids at compile-time), it is usually best to intern each string only once and save off the result for later use. For example, it would be preferable to write code like this because the latter implementation causes the strings to be unnecessarily re-interned every time the function `f()` is called.

```
static StringId   sid_foo = internString("foo");
static StringId   sid_bar = internString("bar");

// ...

void f(StringId id)
{
    if (id == sid_foo)
    {
        // handle case of id == "foo"
    }
    else if (id == sid_bar)
    {
        // handle case of id == "bar"
    }
}
```

The following approach is less efficient:

```
void f(StringId id)
{
    if (id == internString("foo"))
    {
        // handle case of id == "foo"
    }
```

```
        else if (id == internString("bar"))
        {
            // handle case of id == "bar"
        }
    }
```

这是 internString() 的一个可能的实现。

字符串.h

```
typedef U32 StringId;

extern StringId internString(const char* str);
```

字符串.cpp

```
static HashTable<StringId, const char*> gStringIdTable;

StringId internString(const char* str)
{
    StringId sid = hashCrc32(str);

    HashTable<StringId, const char*>::iterator it
        = gStringIdTable.find(sid);

    if (it == gStringTable.end())
    {
        // This string has not yet been added to the
        // table. Add it, being sure to copy it in case
        // the original was dynamically allocated and
        // might later be freed.
        gStringTable[sid] = strdup(str);
    }

    return sid;
}
```

虚幻引擎采用的另一个想法是将字符串 ID 和指向相应 C 风格字符数组的指针包装在一个微型类中。在虚幻引擎中，这个类被称为 FName 。在顽皮狗，我们也采取了同样的措施，将字符串 ID 包装在一个 StringId 类中。我们定义了一个宏，以便 SID("any_string") 生成这个类的一个实例，其哈希值由我们用户定义的字符串字面量语法生成。

`"any_string"_sid`.


```
        else if (id == internString("bar"))
        {
            // handle case of id == "bar"
        }
    }
```

Here's one possible implementation of `internString()`.

stringid.h

```
typedef U32 StringId;

extern StringId internString(const char* str);
```

stringid.cpp

```
static HashTable<StringId, const char*> gStringIdTable;

StringId internString(const char* str)
{
    StringId sid = hashCrc32(str);

    HashTable<StringId, const char*>::iterator it
        = gStringIdTable.find(sid);

    if (it == gStringTable.end())
    {
        // This string has not yet been added to the
        // table. Add it, being sure to copy it in case
        // the original was dynamically allocated and
        // might later be freed.
        gStringTable[sid] = strdup(str);
    }

    return sid;
}
```

Another idea employed by the Unreal Engine is to wrap the string id and a pointer to the corresponding C-style character array in a tiny class. In the Unreal Engine, this class is called `FName`. At Naughty Dog we do the same and wrap our string ids in a `StringId` class. We define a macro so that `SID("any_string")` produces an instance of this class, with its hashed value produced by our user-defined string literal syntax `"any_string"_sid`.

使用调试内存存储字符串

使用字符串 ID 时，字符串本身仅供人类使用。游戏发行时，几乎肯定不需要这些字符串——游戏本身应该只使用这些 ID。因此，最好将字符串表存储在零售版游戏中不存在的内存区域中。例如，PS3 开发套件有 256 MiB 的零售内存，以及零售版中没有的 256 MiB 的“调试”内存。如果我们将字符串存储在调试内存中，就无需担心它们对最终发行版游戏的内存占用的影响。（我们只需注意，切勿编写依赖于可用字符串的生产代码！）

6.4.4 本地化

游戏（或任何软件项目）的本地化是一项艰巨的任务。最好的处理方式是从第一天开始就进行规划，并在开发的每个阶段都考虑到这一点。然而，我们通常不会像希望的那样经常这样做。以下是一些技巧，希望能帮助你规划游戏引擎项目的本地化。有关软件本地化的深入探讨，请参阅[34]。

6.4.4.1 Unicode

大多数英语软件开发人员的问题在于，他们从出生起（或大约从出生起！）就被训练成将字符串视为八位 ASCII 字符代码（即遵循 ANSI 标准的字符）的数组。ANSI 字符串对于字母表简单的语言（例如英语）来说非常有效。但是，对于字母表复杂的语言（包含比英语 26 个字母多得多的字符，有时甚至包含完全不同的字形）来说，ANSI 字符串就显得力不从心了。为了克服 ANSI 标准的局限性，Unicode 字符集系统应运而生。

Unicode 的基本思想是将全球常用语言中的每个字符或字形分配给一个唯一的十六进制代码，称为代码点。在内存中存储字符串时，我们会选择一种特定的编码方式（即表示每个字符 Unicode 代码点的特定方式），并遵循这些规则，在内存中设置代表该字符串的位序列。UTF-8 和 UTF-16 是两种常见的编码方式。您应该选择最适合您需求的特定编码标准。

请立即放下这本书，阅读 Joel Spolsky 的文章《每个软件开发人员绝对、肯定必须了解的 Unicode 和字符集最低要求（绝无借口！）》。文章地址：http://www.joelonsoftware.com/articles/Unicode.html。

（完成后，请再次拿起这本书！）

Using Debug Memory for Strings

When using string ids, the strings themselves are only kept around for human consumption. When you ship your game, you almost certainly won't need the strings—the game itself should only ever use the ids. As such, it's a good idea to store your string table in a region of memory that won't exist in the retail game. For example, a PS3 development kit has 256 MiB of retail memory, plus an additional 256 MiB of "debug" memory that is not available on a retail unit. If we store our strings in debug memory, we needn't worry about their impact on the memory footprint of the final shipping game. (We just need to be careful never to write production code that depends on the strings being available!)

6.4.4 Localization

Localization of a game (or any software project) is a big undertaking. It is a task best handled by planning for it from day one and accounting for it at every step of development. However, this is not done as often as we all would like. Here are some tips that should help you plan your game engine project for localization. For an in-depth treatment of software localization, see [34].

6.4.4.1 Unicode

The problem for most English-speaking software developers is that they are trained from birth (or thereabouts!) to think of strings as arrays of eight-bit ASCII character codes (i.e., characters following the ANSI standard). ANSI strings work great for a language with a simple alphabet, like English. But, they just don't cut it for languages with complex alphabets containing a great many more characters, sometimes totally different glyphs than English's 26 letters. To address the limitations of the ANSI standard, the Unicode character set system was devised.

The basic idea behind Unicode is to assign every character or glyph from every language in common use around the globe to a unique hexadecimal code known as a *code point*. When storing a string of characters in memory, we select a particular *encoding*—a specific means of representing the Unicode code points for each character—and following those rules, we lay down a sequence of bits in memory that represent the string. UTF-8 and UTF-16 are two common encodings. You should select the specific encoding standard that best suits your needs.

Please set down this book right now and read the article entitled, "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" by Joel Spolsky. You can find it here: http://www.joelonsoftware.com/articles/Unicode.html. (Once you've done that, please pick up the book again!)

UTF-32

最简单的 Unicode 编码是 UTF-32。在这种编码中，每个 Unicode 码位都被编码为一个 32 位（4 字节）的值。这种编码浪费了大量空间，原因有二：首先，西欧语言中的大多数字符串不使用任何最高值码位，因此每个字符通常平均至少浪费 16 位（2 字节）。其次，最高的 Unicode 码位是 0x10FFFF，所以即使我们想要创建一个使用所有可能的 Unicode 字形的字符串，每个字符仍然只需要 21 位，而不是 32 位。

话虽如此，UTF-32 确实有其自身的优势。它是一种固定长度的编码，这意味着每个字符在内存中占用相同数量的位（准确地说是 32 位）。因此，我们可以通过将任何 UTF-32 字符串的长度（以字节为单位）除以 4 来确定其长度。

UTF-8

在 UTF-8 编码方案中，字符串中每个字符的代码点以八位（一个字节）的粒度存储，但某些代码点占用多个字节。因此，UTF-8 字符串占用的字节数不一定等于字符串的字符长度。这被称为可变长度编码或多字节字符集 (MBCS)，因为字符串中的每个字符可能占用一个或多个字节的存储空间。

UTF-8 编码的一大优势在于它与 ANSI 编码向后兼容。这是因为前 127 个 Unicode 码位在数字上与旧的 ANSI 字符码相对应。这意味着每个 ANSI 字符在 UTF-8 中都恰好用一个字节表示，并且 ANSI 字符串可以无需修改地解释为 UTF-8 字符串。

为了表示更高值的代码点，UTF-8 标准使用多字节字符。每个多字节字符都以最高有效位为 1 的字节开头（即，其值在 128 到 255 之间，含 128 和 255）。这种高值字节永远不会出现在 ANSI 字符串中，因此在区分单字节字符和多字节字符时不会产生歧义。

UTF-16

UTF-16 编码采用了一种稍微简单一些但开销更大的方法。UTF-16 字符串中的每个字符都由一个或两个 16 位值表示。UTF-16 编码被称为宽字符集 (WCS)，因为每个字符至少有 16 位宽，而不是 Unicode 字符集使用的 8 位。

UTF-32

The simplest Unicode encoding is UTF-32. In this encoding, each Unicode code point is encoded into a 32-bit (4-byte) value. This encoding wastes a lot of space, for two reasons: First, most strings in Western European languages do not use any of the highest-valued code points, so an average of at least 16 bits (2 bytes) is usually wasted per character. Second, the highest Unicode code point is 0x10FFFF, so even if we wanted to create a string that uses every possible Unicode glyph, we'd still only need 21 bits per character, not 32.

That said, UTF-32 does have simplicity in its favor. It is a *fixed-length* encoding, meaning that every character occupies the same number of bits in memory (32 bits to be precise). As such, we can determine the length of any UTF-32 string by taking its length in bytes and dividing by four.

UTF-8

In the UTF-8 encoding scheme, the code points for each character in a string are stored using eight-bit (one-byte) granularity, but some code points occupy more than one byte. Hence the number of bytes occupied by a UTF-8 character string is not necessarily the length of the string in characters. This is known as a *variable-length* encoding, or a *multibyte character set* (MBCS), because each character in a string may take one or more bytes of storage.

One of the big benefits of the UTF-8 encoding is that it is backwards-compatible with the ANSI encoding. This works because the first 127 Unicode code points correspond numerically to the old ANSI character codes. This means that every ANSI character will be represented by exactly one byte in UTF-8, and a string of ANSI characters can be interpreted as a UTF-8 string without modification.

To represent higher-valued code points, the UTF-8 standard uses multibyte characters. Each multibyte character starts with a byte whose most-significant bit is 1 (i.e., its value lies in the range 128–255, inclusive). Such high-valued bytes will never appear in an ANSI string, so there is no ambiguity when distinguishing between single-byte characters and multibyte characters.

UTF-16

The UTF-16 encoding employs a somewhat simpler, albeit more expensive approach. Each character in a UTF-16 string is represented by either one or two 16-bit values. The UTF-16 encoding is known as a wide character set (WCS) because each character is at least 16 bits wide, instead of the eight bits used by

"常规" ANSI 字符及其 UTF-8 对应字符。

在 UTF-16 中，所有可能的 Unicode 码点集合被划分为 17 个平面，每个平面包含 2^16 个码点。第一个平面称为基本多文种平面 (BMP)。它包含各种语言中最常用的码点。因此，许多 UTF-16 字符串可以完全由第一个平面中的码点表示，这意味着此类字符串中的每个字符仅由一个 16 位值表示。但是，如果字符串中需要使用其他平面（称为补充平面）中的字符，则该字符由两个连续的 16 位值表示。

UCS-2（双字节通用字符集）编码是 UTF-16 编码的一个有限子集，仅利用了基本的多语言页面。因此，它无法表示 Unicode 码位大于 0xFFFF 的字符。这简化了格式，因为每个字符都保证恰好占用 16 位（两个字节）。换句话说，UCS-2 是一种定长字符编码，而 UTF-8 和 UTF-16 通常都是变长编码。

如果我们事先知道 UTF-16 字符串仅使用 BMP 中的代码点（或者我们处理的是 UCS-2 编码的字符串），那么我们可以通过简单地将字节数除以二来确定字符串中的字符数。当然，如果 UTF-16 字符串中使用了补充平面，这个简单的"技巧"就不再有效了。

请注意，UTF-16 编码可以是小端序或大端序（参见第 3.3.2.1 节），具体取决于目标 CPU 的原生字节序。在磁盘上存储 UTF-16 文本时，通常会在文本数据前面加上字节顺序标记 (BOM)，以指示单个 16 位字符是以小端序还是大端序存储的。（当然，UTF-32 编码的字符串数据也是如此。）

6.4.4.2 char 与 wchar_t

标准 C/C++ 库定义了两种用于处理字符串的数据类型 — char 和 wchar_t 。char 类型旨在与旧式 ANSI 字符串和多字节字符集 (MBCS) 一起使用，包括（但不限于）UTF-8。wchar_t 类型是一种"宽"字符类型，旨在能够在单个整数中表示任何有效的代码点。因此，它的大小是编译器和系统特定的。在完全不支持 Unicode 的系统上，它可能为 8 位。如果假定所有宽字符都采用 UCS-2 编码，或者如果使用像 UTF-16 这样的多字编码，则它可能为 16 位。或者，如果 UTF-32 是所选的"宽"字符编码，则它可能为 32 位。

由于 wchar_t 定义中固有的歧义性，如果你

"regular" ANSI `chars` and their UTF-8 counterparts.

In UTF-16, the set of all possible Unicode code points is divided into 17 *planes* containing 2^{16} code points each. The first plane is known as the *basic multilingual plane* (BMP). It contains the most commonly used code points across a wide range of languages. As such, many UTF-16 strings can be represented entirely by code points within the first plane, meaning that each character in such a string is represented by only *one* 16-bit value. However, if a character from one of the other planes (known as *supplementary planes*) is required in a string, it is represented by two consecutive 16-bit values.

The UCS-2 (2-byte universal character set) encoding is a limited subset of the UTF-16 encoding, utilizing *only* the basic multilingual page. As such, it cannot represent characters whose Unicode code points are numerically higher than 0xFFFF. This simplifies the format, because every character is guaranteed to occupy exactly 16 bits (two bytes). In other words, UCS-2 is a *fixed-length* character encoding, while in general UTF-8 and UTF-16 are *variable-length* encodings.

If we know a priori that a UTF-16 string only utilizes code points from the BMP (or if we are dealing with a UCS-2 encoded string), we can determine the number of characters in the string by simply dividing the number of bytes by two. Of course, if supplemental planes are used in a UTF-16 string, this simple "trick" no longer works.

Note that a UTF-16 encoding can be little-endian or big-endian (see Section 3.3.2.1), depending on the native endianness of your target CPU. When storing UTF-16 text on-disc, it's common to precede the text data with a *byte order mark* (BOM) indicating whether the individual 16-bit characters are stored in little- or big-endian format. (This is true of UTF-32 encoded string data as well, of course.)

6.4.4.2 `char` versus `wchar_t`

The standard C/C++ library defines two data types for dealing with character strings—`char` and `wchar_t`. The `char` type is intended for use with legacy ANSI strings and with multibyte character sets (MBCS), including (but not limited to) UTF-8. The `wchar_t` type is a "wide" character type, intended to be capable of representing any valid code point in a single integer. As such, its size is compiler- and system-specific. It could be eight bits on a system that does not support Unicode at all. It could be 16 bits if the UCS-2 encoding is assumed for all wide characters, or if a multi-word encoding like UTF-16 is being employed. Or it could be 32 bits if UTF-32 is the "wide" character encoding of choice.

Because of this inherent ambiguity in the definition of `wchar_t`, if you

如果您需要编写真正可移植的字符串处理代码，您需要定义自己的字符数据类型，并提供一个函数库来处理您需要支持的任何 Unicode 编码。但是，如果您针对的是特定的平台和编译器，您可以在该特定实现的限制范围内编写代码，但会损失一些可移植性。

以下文章很好地概述了使用 wchar_t 数据类型的优缺点：http://icu-project.org/docs/papers/unicode_ wchar_t.html。

6.4.4.3 Windows 下的 Unicode

在 Windows 下，wchar_t 数据类型专用于 UTF-16 编码的 Unicode 字符串，而 char 类型用于 ANSI 字符串和旧版 Windows 代码页字符串编码。因此，在阅读 Windows API 文档时，“Unicode”一词始终与“宽字符集”（WCS）和 UTF-16 编码同义。这有点令人困惑，因为 Unicode 字符串通常可以用“非宽”多字节 UTF-8 格式进行编码。

Windows API 定义了三组字符/字符串操作函数：一组用于单字节字符集 ANSI 字符串 (SBCS)，一组用于多字节字符集 (MBCS) 字符串，一组用于宽字符集字符串。ANSI 函数本质上是我们从小就熟悉的老式“C 风格”字符串函数。MBCS 字符串函数处理各种多字节编码，主要用于处理旧版 Windows 代码页编码。WCS 函数处理 Unicode UTF-16 字符串。

在整个 Windows API 中，前缀或后缀“w”、“wcs”或“W”表示宽字符集 (UTF-16) 编码；前缀或后缀“mb”表示多字节编码；前缀或后缀“a”或“A”，或者没有任何前缀或后缀，表示 ANSI 或 Windows 代码页编码。C++ 标准库使用类似的约定 - 例如，std::string 是其 ANSI 字符串类，而 std::wstring 是其宽字符等效类。不幸的是，函数名称并不总是 100% 一致。这会导致不了解情况的程序员产生一些混淆。（但您不是其中之一！）表 6.2 列出了一些示例。

Windows 还提供了在 ANSI 字符串、多字节字符串和宽 UTF-16 字符串之间进行转换的函数。例如，wcstombs() 函数会根据当前活动的区域设置将宽 UTF-16 字符串转换为多字节字符串。

Windows API 使用一个小的预处理器技巧，允许您编写至少表面上可在宽（Unicode）和非宽之间移植的代码

need to write truly portable string-handling code, you'll need to define your own character data type(s) and provide a library of functions for dealing with whatever Unicode encoding(s) you need to support. However, if you are targeting a specific platform and compiler, you can write your code within the limits of that particular implementation, at the loss of some portability.

The following article does a good job of outlining the pros and cons of using the `wchar_t` data type: http://icu-project.org/docs/papers/unicode_wchar_t.html.

6.4.4.3 Unicode under Windows

Under Windows, the `wchar_t` data type is used exclusively for UTF-16 encoded Unicode strings, and the `char` type is used for ANSI strings and legacy *Windows code page* string encodings. When reading the Windows API docs, the term "Unicode" is therefore always synonymous with "wide character set" (WCS) and UTF-16 encoding. This is a bit confusing, because of course Unicode strings can in general be encoded in the "non-wide" multibyte UTF-8 format.

The Windows API defines three sets of character/string manipulation functions: one set for single-byte character set ANSI strings (SBCS), one set for multibyte character set (MBCS) strings, and one set for wide character set strings. The ANSI functions are essentially the old-school "C-style" string functions we all grew up with. The MBCS string functions handle a variety of multibyte encodings and are primarily designed for dealing with legacy Windows code pages encodings. The WCS functions handle Unicode UTF-16 strings.

Throughout the Windows API, a prefix or suffix of "w," "wcs" or "W" indicates a wide character set (UTF-16) encoding; a prefix or suffix of "mb" indicates a multibyte encoding; and a prefix or suffix of "a" or "A," or the lack of any prefix or suffix, indicates an ANSI or Windows code pages encoding. The C++ standard library uses a similar convention—for example, `std::string` is its ANSI string class, while `std::wstring` is its wide character equivalent. Unfortunately, the names of the functions aren't always 100% consistent. This all leads to some confusion among programmers who aren't in the know. (But *you* aren't one of those programmers!) Table 6.2 lists some examples.

Windows also provides functions for translating between ANSI character strings, multibyte strings and wide UTF-16 strings. For example, `wcstombs()` converts a wide UTF-16 string into a multibyte string according to the currently active *locale* setting.

The Windows API uses a little preprocessor trick to allow you to write code that is at least superficially portable between wide (Unicode) and non-wide

ANSI	WCS	MBCS
strcmp()	wcscmp()	_mbscmp()
strcpy()	wcscpy()	_mbscpy()
strlen()	wcslen()	_mbstrlen()

表 6.2. 用于 ANSI、宽和多字节字符集的一些常见 C 标准库字符串函数的变体。

(ANSI/MBCS) 字符串编码。在“ANSI 模式”下构建应用程序时，通用字符数据类型 TCHAR 被定义为 char 的 typedef；在“Unicode 模式”下构建应用程序时，它被定义为 wchar_t 的 typedef。宏 _T() 用于在“Unicode 模式”下编译时将八位字符串文字（例如，char* s = "this is a string"; ）转换为宽字符串文字（例如，wchar_t* s = L"this is a string"; ）。同样，提供了一套“伪” API 函数，可以根据您是否在“Unicode 模式”下构建，自动转换为相应的 8 位或 16 位变体。这些与字符集无关的神奇函数要么没有前缀或后缀，要么带有“t”、“tcs”或“T”前缀或后缀。

所有这些函数的完整文档都可以在微软的 MSDN 网站上找到。以下是 strcmp() 及其同类函数的文档链接，您可以通过页面左侧的树形视图或搜索栏轻松导航到其他相关的字符串操作函数：http://msdn2.microsoft.com/en-us/library/kk6xf663(VS.

80).aspx。

6.4.4.4 控制台上的 Unicode

Xbox 360 软件开发工具包 (XDK) 几乎完全使用 WCS 字符串，所有字符串甚至包括文件路径等内部字符串。这无疑是解决本地化问题的有效方法之一，并且能够在整个 XDK 中实现高度一致的字符串处理。然而，UTF-16 编码会浪费一些内存，因此不同的游戏引擎可能会采用不同的约定。在顽皮狗，我们在整个引擎中使用 8 位字符串，并使用 UTF-8 编码处理外语。编码的选择并不特别重要，只要在项目早期就尽可能选择一种并始终如一地使用即可。

6.4.4.5 其他本地化问题

即使你已经将软件适配到使用 Unicode 字符，仍然有许多其他本地化问题需要解决。首先，

ANSI	WCS	MBCS
strcmp()	wcscmp()	_mbscmp()
strcpy()	wcscpy()	_mbscpy()
strlen()	wcslen()	_mbstrlen()

Table 6.2. Variants of some common C standard library string functions for use with ANSI, wide and multibyte character sets.

(ANSI/MBCS) string encodings. The generic character data type `TCHAR` is defined to be a `typedef` to `char` when building your application in “ANSI mode,” and it’s defined to be a `typedef` to `wchar_t` when building your application in “Unicode mode.” The macro `_T()` is used to convert an eight-bit string literal (e.g., `char* s = "this is a string";`) into a wide string literal (e.g., `wchar_t* s = L"this is a string";`) when compiling in “Unicode mode.” Likewise, a suite of “fake” API functions are provided that “automagically” morph into their appropriate 8-bit or 16-bit variant, depending on whether you are building in “Unicode mode” or not. These magic character-set-independent functions are either named with no prefix or suffix, or with a “t,” “tcs” or “T” prefix or suffix.

Complete documentation for all of these functions can be found on Microsoft’s MSDN website. Here’s a link to the documentation for `strcmp()` and its ilk, from which you can quite easily navigate to the other related string-manipulation functions using the tree view on the left-hand side of the page, or via the search bar: http://msdn2.microsoft.com/en-us/library/kk6xf663(VS.80).aspx.

6.4.4.4 Unicode on Consoles

The Xbox 360 software development kit (XDK) uses WCS strings pretty much exclusively, for all strings—even for internal strings like file paths. This is certainly one valid approach to the localization problem, and it makes for very consistent string handling throughout the XDK. However, the UTF-16 encoding is a bit wasteful on memory, so different game engines may employ different conventions. At Naughty Dog, we use eight-bit `char` strings throughout our engine, and we handle foreign languages via a UTF-8 encoding. The choice of encoding is not particularly important, as long as you select one as early in the project as possible and stick with it consistently.

6.4.4.5 Other Localization Concerns

Even once you have adapted your software to use Unicode characters, there is still a host of other localization problems to contend with. For one thing,

Id	英语	法语
p1分数	“玩家 1 得分”	“玩家 1 得分”
p2分数	“玩家 2 得分”	“玩家 2 得分”
p1wins	“玩家一获胜！”	“玩家一获胜！”
p2wins	“二号玩家获胜！”	“二号玩家获胜！”

表 6.3. 用于本地化的字符串数据库示例。

字符串并非唯一会出现本地化问题的地方。音频片段（包括录音）也必须翻译。纹理中可能包含需要翻译的英文单词。许多符号在不同的文化中含义不同。即使是像“禁止吸烟”标志这样无害的符号，也可能在另一种文化中被误解。此外，一些市场对不同游戏分级的界限划分也有所不同。例如，在日本，青少年级游戏不允许出现任何形式的血腥场面，而在北美，少量红色血迹飞溅则被认为是可以接受的。

对于字符串，还有其他细节需要考虑。您需要管理游戏中所有人类可读字符串的数据库，以便所有字符串都能被可靠地翻译。软件必须根据用户的安装设置显示正确的语言。不同语言的字符串格式可能完全不同——例如，中文有时是竖排的，而希伯来语则是从右到左阅读的。不同语言的字符串长度差异很大。您还需要决定是发行包含所有语言的单张 DVD 或蓝光光盘，还是针对特定地区发行不同的光盘。

本地化系统中最关键的组件是包含人类可读字符串的中央数据库和通过 ID 查找这些字符串的游戏内系统。例如，假设您想要一个平视显示器，使用“玩家 1 得分：”和“玩家 2 得分：”标签列出每个玩家的得分，并在回合结束时显示文本“玩家 1 获胜”或“玩家 2 获胜”。这四个字符串将以游戏开发者可以理解的唯一 ID 存储在本地化数据库中。因此，我们的数据库可能分别使用 ID“p1score”、“p2score”、“p1wins”和“p2wins”。将游戏的字符串翻译成法语后，我们的数据库将类似于表 6.3 中所示的简单示例。可以为游戏支持的每种新语言添加附加列。

这个数据库的具体格式由你决定。它可以像 Microsoft Excel 工作表一样简单，可以保存为逗号分隔值 (CSV) 文件并由游戏引擎解析；也可以像功能齐全的 Or- 一样复杂。

Id	English	French
p1score	“Player 1 Score”	“Joueur 1 Score”
p2score	“Player 2 Score”	“Joueur 2 Score”
p1wins	“Player one wins!”	“Joueur un gagne!”
p2wins	“Player two wins!”	“Joueur deux gagne!”

Table 6.3. Example of a string database used for localization.

strings aren’t the only place where localization issues arise. Audio clips including recorded voices must be translated. Textures may have English words painted into them that require translation. Many symbols have different meanings in different cultures. Even something as innocuous as a no-smoking sign might be misinterpreted in another culture. In addition, some markets draw the boundaries between the various game-rating levels differently. For example, in Japan a Teen-rated game is not permitted to show blood of any kind, whereas in North America small red blood spatters are considered acceptable.

For strings, there are other details to worry about as well. You will need to manage a database of all human-readable strings in your game, so that they can all be reliably translated. The software must display the proper language given the user’s installation settings. The formatting of the strings may be totally different in different languages—for example, Chinese is sometimes written vertically, and Hebrew reads right-to-left. The lengths of the strings will vary greatly from language to language. You’ll also need to decide whether to ship a single DVD or Blu-ray disc that contains all languages or ship different discs for particular territories.

The most crucial components in your localization system will be the central database of human-readable strings and an in-game system for looking up those strings by id. For example, let’s say you want a heads-up display that lists the score of each player with “Player 1 Score:” and “Player 2 Score:” labels and that also displays the text “Player 1 Wins” or “Player 2 Wins” at the end of a round. These four strings would be stored in the localization database under unique ids that are understandable to you, the developer of the game. So our database might use the ids “p1score,” “p2score,” “p1wins” and “p2wins,” respectively. Once our game’s strings have been translated into French, our database would look something like the simple example shown in Table 6.3. Additional columns can be added for each new language your game supports.

The exact format of this database is up to you. It might be as simple as a Microsoft Excel worksheet that can be saved as a comma-separated values (CSV) file and parsed by the game engine or as complex as a full-fledged Or-

字符串数据库。字符串数据库的具体细节对游戏引擎来说并不重要，只要它能够读取游戏支持的语言对应的字符串 ID 和 Unicode 字符串即可。（然而，从实际角度来看，数据库的具体细节可能非常重要，这取决于游戏工作室的组织结构。拥有内部翻译人员的小型工作室可能可以使用位于网络驱动器上的 Excel 电子表格。但在英国、欧洲、南美和日本设有分支机构的大型工作室可能会发现某种分布式数据库更为合适。）

在运行时，你需要提供一个简单的函数，该函数返回“当前”语言的 Unicode 字符串，并传入该字符串的唯一 ID。该函数可以像这样声明：

```
wchar_t getLocalizedString(const char* id);
```

它的用法可能如下：

```
void drawScoreHud(const Vector3& score1Pos,
                  const Vector3& score2Pos)
{
    renderer.displayTextOrtho(getLocalizedString("p1score"),
                              score1Pos);

    renderer.displayTextOrtho(getLocalizedString("p2score"),
                              score2Pos);

    // ...
}
```

当然，您需要某种方法来全局设置“当前”语言。这可以通过配置设置来实现，该设置会在游戏安装过程中固定下来。或者，您可以允许用户通过游戏内菜单随时更改当前语言。无论哪种方式，设置都不难实现；它可以简单到只需一个全局整数变量，指定要读取的字符串表中列的索引（例如，第一列可能是英语，第二列是法语，第三列是西班牙语，等等）。

一旦建立了这个基础架构，程序员必须记住永远不要向用户显示原始字符串。他们必须始终使用数据库中字符串的 ID，并调用查找函数来检索所需的字符串。

6.4.4.6 案例研究：顽皮狗的本地化工具

在顽皮狗，我们使用内部开发的本地化数据库。本地化工具的后端由位于

acle database. The specifics of the string database are largely unimportant to the game engine, as long as it can read in the string ids and the corresponding Unicode strings for whatever language(s) your game supports. (However, the specifics of the database may be *very* important from a practical point of view, depending upon the organizational structure of your game studio. A small studio with in-house translators can probably get away with an Excel spreadsheet located on a network drive. But a large studio with branch offices in Britain, Europe, South America and Japan would probably find some kind of distributed database a great deal more amenable.)

At runtime, you'll need to provide a simple function that returns the Unicode string in the "current" language, given the unique id of that string. The function might be declared like this:

```
wchar_t getLocalizedString(const char* id);
```

and it might be used like this:

```
void drawScoreHud(const Vector3& score1Pos,
                  const Vector3& score2Pos)
{
    renderer.displayTextOrtho(getLocalizedString("p1score"),
                              score1Pos);

    renderer.displayTextOrtho(getLocalizedString("p2score"),
                              score2Pos);

    // ...
}
```

Of course, you'll need some way to set the "current" language globally. This might be done via a configuration setting, which is fixed during the installation of the game. Or you might allow users to change the current language on the fly via an in-game menu. Either way, the setting is not difficult to implement; it can be as simple as a global integer variable specifying the index of the column in the string table from which to read (e.g., column one might be English, column two French, column three Spanish and so on).

Once you have this infrastructure in place, your programmers must remember to *never display a raw string to the user*. They must always use the id of a string in the database and call the look-up function in order to retrieve the string in question.

6.4.4.6 Case Study: Naughty Dog's Localization Tool

At Naughty Dog, we use a localization database that we developed in-house. The localization tool's back end consists of a MySQL database located on a

图 6.10。顽皮狗本地化工具的主窗口，显示了菜单和 HUD 中使用的纯文本资源列表。用户刚刚搜索了名为 MENU_NEWGAME 的资源。

图 6.11.详细资产视图，显示 MENU_NEWGAME 字符串。

Figure 6.10. Naughty Dog's localization tool's main window, showing a list of pure text assets used in the menus and HUD. The user has just performed a search for an asset called `MENU_NEWGAME`.

Figure 6.11. Detailed asset view, showing the `MENU_NEWGAME` string.

该服务器不仅可供顽皮狗内部的开发者访问，也可供我们合作的外部公司访问，以便将我们的文本和语音音频片段翻译成我们游戏支持的各种语言。前端是一个与数据库“对话”的 Web 界面，允许用户查看所有文本和音频资产、编辑其内容、为每个资产提供翻译、按 ID 或内容搜索资产等等。

在顽皮狗的本地化工具中，每个资源要么是字符串（用于菜单或HUD），要么是带有可选字幕文本的语音音频片段（用于游戏内对话或过场动画）。每个资源都有一个唯一标识符，该标识符以散列字符串ID表示（参见第6.4.3.1节）。如果菜单或HUD需要使用某个字符串，我们会通过其ID查找，并返回一个适合在屏幕上显示的Unicode (UTF-8)字符串。如果必须播放一段对话，我们同样会通过其ID查找音频片段，并使用引擎中的数据查找其对应的字幕（如果有）。字幕的处理方式与菜单或HUD字符串相同，因为它由本地化工具的API返回为适合显示的UTF-8字符串。

图 6.10 展示了本地化工具的主界面，本例中显示的是 Chrome 网络浏览器。在此图中，您可以看到用户输入了 ID MENU_NEWGAME 来查找字符串“NEW GAME”（用于游戏主菜单中启动新游戏）。图 6.11 展示了 MENU_NEWGAME 资源的详细视图。如果用户点击资源详细信息窗口左上角的“文本翻译”按钮，则会显示图 6.12 所示的屏幕，允许用户输入或编辑该字符串的各种翻译。图 6.13 展示了本地化工具主页上的另一个选项卡，这次列出了音频语音资源。最后，图 6.14 展示了语音资源 BADA_GAM_MIL_ESCAPE_OVERPASS_001（“我们错过了所有行动”）的详细资源视图，其中显示了这句对话被翻译成部分支持的语言。

6.5 发动机配置

游戏引擎非常复杂，它们不可避免地会有大量可配置的选项。其中一些选项会通过游戏中的一个或多个选项菜单向玩家展示。例如，游戏可能会显示与图形质量、音乐和音效音量或控制器配置相关的选项。其他选项仅为游戏开发团队创建，在游戏发行前要么被隐藏，要么被完全移除。例如，玩家角色的

server that is accessible both to the developers within Naughty Dog and also to the various external companies with which we work to translate our text and speech audio clips into the various languages our games support. The front end is a web interface that “speaks” to the database, allowing users to view all of the text and audio assets, edit their contents, provide translations for each asset, search for assets by id or by content and so on.

In Naughty Dog’s localization tool, each asset is either a string (for use in the menus or HUD) or a speech audio clip with optional subtitle text (for use as in-game dialog or within cinematics). Each asset has a unique identifier, which is represented as a hashed string id (see Section 6.4.3.1). If a string is required for use in the menus or HUD, we look it up by its id and get back a Unicode (UTF-8) string suitable for display on-screen. If a line of dialog must be played, we likewise look up the audio clip by its id and use the data in-engine to look up its corresponding subtitle (if any). The subtitle is treated just like a menu or HUD string, in that it is returned by the localization tool’s API as a UTF-8 string suitable for display.

Figure 6.10 shows the main interface of the localization tool, in this case displayed in the Chrome web browser. In this image, you can see that the user has typed in the id `MENU_NEWGAME` in order to look up the string “NEW GAME” (used on the game’s main menu for launching a new game). Figure 6.11 shows the detailed view of the `MENU_NEWGAME` asset. If the user hits the “Text Translations” button in the upper-left corner of the asset details window, the screen shown in Figure 6.12 comes up, allowing the user to enter or edit the various translations of the string. Figure 6.13 shows another tab on the localization tool’s main page, this time listing audio speech assets. Finally, Figure 6.14 depicts the detailed asset view for the speech asset `BADA_GAM_MIL_ESCAPE_OVERPASS_001` (“We missed all the action”), showing translations of this line of dialog into some of the supported languages.

6.5 Engine Configuration

Game engines are complex beasts, and they invariably end up having a large number of configurable options. Some of these options are exposed to the player via one or more options menus in-game. For example, a game might expose options related to graphics quality, the volume of music and sound effects, or controller configuration. Other options are created for the benefit of the game development team only and are either hidden or stripped out of the game completely before it ships. For example, the player character’s

图 6.12. 将字符串“NEW GAME”翻译成顽皮狗的《最后生还者》支持的所有语言。

最大步行速度可能会作为一个选项公开，以便在开发过程中进行微调，但在发货前可能会更改为硬编码值。

6.5.1 加载和保存选项

可配置选项可以简单地实现为全局变量或单例类的成员变量。但是，除非可配置选项的值可以配置，并存储在硬盘、存储卡或其他存储介质上，以便游戏稍后检索，否则可配置选项的作用并不大。加载和保存配置选项的方法有很多：

- 文本配置文件。迄今为止，保存和加载配置选项的最常用方法是将它们放入一个或多个文本

Figure 6.12. Text translations of the string “NEW GAME” into all languages supported by Naughty Dog's *The Last of Us*.

maximum walk speed might be exposed as an option so that it can be fine-tuned during development, but it might be changed to a hard-coded value prior to ship.

6.5.1 Loading and Saving Options

A configurable option can be implemented trivially as a global variable or a member variable of a singleton class. However, configurable options are not particularly useful unless their values can be configured, stored on a hard disk, memory card or other storage medium, and later retrieved by the game. There are a number of simple ways to load and save configuration options:

- *Text configuration files*. By far the most common method of saving and loading configuration options is by placing them into one or more text

图 6.13.顽皮狗本地化工具的主窗口，这次显示了带有字幕文本的语音音频资产列表。

图 6.14。详细资产视图显示语音资产 BADA_GAM_MIL_ESCAPE_OVERPASS_001（“我们错过了所有动作”）的录制翻译。

Figure 6.13. Naughty Dog's localization tool's main window again, this time showing a list of speech audio assets with accompanying subtitle text.

Figure 6.14. Detailed asset view showing recorded translations for the speech asset `BADA_GAM_MIL_ESCAPE_OVERPASS_001` ("We missed all the action").

文件。这些文件的格式因引擎而异，但通常非常简单。例如，Windows INI 文件（由 OGRE 渲染器使用）由按逻辑部分分组的键值对的扁平列表组成。JSON 格式是可配置游戏选项文件的另一种常见选择。XML 也是另一种可行的选择，尽管如今大多数开发人员发现 JSON 比 XML 更简洁、更易于阅读。

• 压缩二进制文件。大多数现代游戏机都配备了硬盘驱动器，但老款游戏机却无法提供这种便利。因此，自超级任天堂娱乐系统 (SNES) 以来，所有游戏机都配备了专有的可移动存储卡，允许读写数据。游戏选项有时会与已保存的游戏一起存储在这些卡上。压缩二进制文件是存储卡的首选格式，因为这些卡上的可用存储空间通常非常有限。

• Windows 注册表。Microsoft Windows 操作系统提供了一个全局选项数据库，称为注册表。它以树状结构存储，其中内部节点（称为注册表项）类似于文件夹，而叶节点则将各个选项存储为键值对。话虽如此，我不建议使用 Windows 注册表来存储引擎配置信息。注册表是一个庞大的数据库，很容易损坏、丢失（Windows 重新安装时），或者与文件系统中的文件不同步。有关 Windows 注册表弱点的更多信息，请参阅 https://blog.coding horror.com/was-the-windows-registry-a-good-idea/。

• 命令行选项。可以通过命令行查看选项设置。引擎可能提供一种机制，允许通过命令行控制游戏中的任何选项，也可能只在此显示一小部分游戏选项。

• 环境变量。在运行 Windows、Linux 或 MacOS 的个人计算机上，环境变量有时也用于存储配置选项。

• 在线用户资料。随着 Xbox Live 等在线游戏社区的出现，每个用户都可以创建个人资料，并用它来保存成就、已购买和可解锁的游戏功能、游戏选项以及其他信息。这些数据存储在中央服务器上，玩家可以在任何有互联网连接的地方访问。

files. The format of these files varies widely from engine to engine, but it is usually very simple. For example, Windows INI files (which are used by the OGRE renderer) consist of flat lists of key-value pairs grouped into logical sections. The JSON format is another common choice for configurable game options files. XML is another viable option, although most developers these days find JSON to be less verbose and easier to read than XML.

- *Compressed binary files*. Most modern consoles have hard disk drives in them, but older consoles could not afford this luxury. As a result, all game consoles since the Super Nintendo Entertainment System (SNES) have come equipped with proprietary removable memory cards that permit both reading and writing of data. Game options are sometimes stored on these cards, along with saved games. Compressed binary files are the format of choice on a memory card, because the storage space available on these cards is often very limited.

- *The Windows registry*. The Microsoft Windows operating system provides a global options database known as the registry. It is stored as a tree, where the interior nodes (known as *registry keys*) act like file folders, and the leaf nodes store the individual options as key-value pairs. That being said, I don't recommend using the Windows registry for storing engine configuration information. The registry is a monolithic database that can easily be corrupted, lost (when Windows is reinstalled), or thrown out-of-sync with the files in the filesystem. For more on the weaknesses of the Windows registry, see https://blog.coding horror.com/was-the-windows-registry-a-good-idea/.

- *Command line options*. The command line can be scanned for option settings. The engine might provide a mechanism for controlling any option in the game via the command line, or it might expose only a small subset of the game's options here.

- *Environment variables*. On personal computers running Windows, Linux or MacOS, environment variables are sometimes used to store configuration options as well.

- *Online user profiles*. With the advent of online gaming communities like Xbox Live, each user can create a profile and use it to save achievements, purchased and unlockable game features, game options and other information. The data are stored on a central server and can be accessed by the player wherever an Internet connection is available.

6.5.2 每个用户的选项

大多数游戏引擎会区分全局选项和每个用户选项。这是必要的，因为大多数游戏允许每个玩家根据自己的喜好配置游戏。这在游戏开发过程中也是一个很有用的概念，因为它允许每个程序员、美术师和设计师自定义自己的工作环境，而不会影响其他团队成员。

显然，存储每个用户选项时必须谨慎，确保每个玩家只能“看到”自己的选项，而看不到同一台电脑或主机上其他玩家的选项。在主机游戏中，用户通常可以将其游戏进度以及控制器偏好设置等每个用户的选项保存在存储卡或硬盘的“插槽”中。这些“插槽”通常以相关介质上的文件形式实现。

在 Windows 计算机上，每个用户在 C:\Users 下都有一个文件夹，其中包含用户桌面、“我的文档”文件夹、Internet 浏览历史记录以及临时文件等信息。一个名为 AppData 的隐藏子文件夹用于按应用程序存储每个用户的信息；每个应用程序都会在 AppData 下创建一个文件夹，并可用它来存储所需的任何用户信息。

Windows 游戏有时会将每个用户的配置数据存储在注册表中。注册表以树状结构排列，根节点的顶级子节点之一 HKEY_CURRENT_USER 存储着当前登录用户的设置。每个用户在注册表中都有自己的子树（存储在顶级子树 HKEY_USERS 下），而 HKEY_CURRENT_USER 实际上只是当前用户子树的别名。因此，游戏和其他应用程序只需在 HKEY_CURRENT_USER 子树下的键中读写每个用户的配置选项即可管理这些配置选项。

6.5.3 一些实际引擎中的配置管理

在本节中，我们将简要了解一些真实的游戏引擎如何管理其配置选项。

6.5.3.1 示例：Quake 的 Cvars

Quake 系列引擎使用一种称为控制台变量（简称 cvar）的配置管理系统。cvar 是一个浮点或字符串全局变量，其值可以在 Quake 的游戏控制台中查看和修改。某些 cvar 的值可以保存到磁盘，之后由引擎重新加载。

在运行时，cvar 存储在一个全局链表中。每个 cvar 都是一个动态分配的 struct cvar_t 实例，其中包含变量的

6.5.2 Per-User Options

Most game engines differentiate between global options and per-user options. This is necessary because most games allow each player to configure the game to his or her liking. It is also a useful concept during development of the game, because it allows each programmer, artist and designer to customize his or her work environment without affecting other team members.

Obviously care must be taken to store per-user options in such a way that each player “sees” only his or her options and not the options of other players on the same computer or console. In a console game, the user is typically allowed to save his or her progress, along with per-user options such as controller preferences, in “slots” on a memory card or hard disk. These slots are usually implemented as files on the media in question.

On a Windows machine, each user has a folder under *C:\Users* containing information such as the user’s desktop, his or her My Documents folder, his or her Internet browsing history and temporary files and so on. A hidden subfolder named *AppData* is used to store per-user information on a per-application basis; each application creates a folder under *AppData* and can use it to store whatever per-user information it requires.

Windows games sometimes store per-user configuration data in the registry. The registry is arranged as a tree, and one of the top-level children of the root node, called `HKEY_CURRENT_USER`, stores settings for whichever user happens to be logged on. Every user has his or her own subtree in the registry (stored under the top-level subtree `HKEY_USERS`), and `HKEY_CURRENT_USER` is really just an alias to the current user’s subtree. So games and other applications can manage per-user configuration options by simply reading and writing them to keys under the `HKEY_CURRENT_USER` subtree.

6.5.3 Configuration Management in Some Real Engines

In this section, we’ll take a brief look at how some real game engines manage their configuration options.

6.5.3.1 Example: Quake’s Cvars

The Quake family of engines uses a configuration management system known as *console variables*, or *cvars* for short. A cvar is just a floating-point or string global variable whose value can be inspected and modified from within Quake’s in-game console. The values of some cvars can be saved to disk and later reloaded by the engine.

At runtime, cvars are stored in a global linked list. Each cvar is a dynamically allocated instance of `struct cvar_t`, which contains the variable’s

名称、其值（字符串或浮点数）、一组标志位以及指向所有 cvar 链表中下一个 cvar 的指针。通过调用 Cvar_Get() 访问 cvar，如果变量不存在，则创建该变量；调用 Cvar_Set() 修改该变量。其中一个位标志 CVAR_ARCHIVE 控制是否将 cvar 保存到名为 config.cfg 的配置文件中。如果设置了此标志，则 cvar 的值将在游戏的多次运行中保持不变。

6.5.3.2 示例：OGRE

OGRE 渲染引擎使用 Windows INI 格式的文本文件集合来配置其选项。默认情况下，这些选项存储在三个文件中，每个文件都与可执行程序位于同一文件夹中：

- plugins.cfg 包含指定启用哪些可选引擎插件以及在磁盘上查找它们的位置的选项。
- resources.cfg 包含一个搜索路径，指定游戏资产（又名
  可以找到媒体（又称资源）。
- ogre.cfg 包含一组丰富的选项，指定要使用的渲染器（DirectX 或 OpenGL）以及首选的视频模式、屏幕尺寸等。

OGRE 本身并不提供存储每个用户配置选项的机制。然而，OGRE 源代码是免费提供的，因此可以很容易地将其更改为在用户主目录中搜索配置文件，而不是在包含可执行文件的文件夹中。Ogre::ConfigFile 类也使得编写读写全新配置文件的代码变得非常简单。

6.5.3.3 示例：《神秘海域》和《最后生还者》系列

Naughty Dog 的引擎使用了许多配置机制。

游戏内菜单设置

顽皮狗引擎支持强大的游戏内菜单系统，允许开发者控制全局配置选项并调用命令。可配置选项的数据类型必须相对简单（主要为布尔值、整数和浮点变量），但这一限制并没有阻止顽皮狗的开发者创建数百个实用的菜单驱动选项。

每个配置选项都实现为全局变量，或者单例结构体或类的成员。创建控制某个选项的菜单选项时，会提供该变量的地址，菜单项会直接控制其值。例如，以下函数会创建一个

name, its value as a string or float, a set of flag bits, and a pointer to the next cvar in the linked list of all cvars. Cvars are accessed by calling `Cvar_Get()`, which creates the variable if it doesn't already exist and modified by calling `Cvar_Set()`. One of the bit flags, `CVAR_ARCHIVE`, controls whether or not the cvar will be saved into a configuration file called *config.cfg*. If this flag is set, the value of the cvar will persist across multiple runs of the game.

6.5.3.2 Example: OGRE

The OGRE rendering engine uses a collection of text files in Windows INI format for its configuration options. By default, the options are stored in three files, each of which is located in the same folder as the executable program:

- *plugins.cfg* contains options specifying which optional engine plug-ins are enabled and where to find them on disk.
- *resources.cfg* contains a *search path* specifying where game assets (a.k.a. media, a.k.a. resources) can be found.
- *ogre.cfg* contains a rich set of options specifying which renderer (DirectX or OpenGL) to use and the preferred video mode, screen size, etc.

Out of the box, OGRE provides no mechanism for storing per-user configuration options. However, the OGRE source code is freely available, so it would be quite easy to change it to search for its configuration files in the user's home directory instead of in the folder containing the executable. The `Ogre::ConfigFile` class makes it easy to write code that reads and writes brand new configuration files as well.

6.5.3.3 Example: The Uncharted and The Last of Us Series

Naughty Dog's engine makes use of a number of configuration mechanisms.

In-Game Menu Settings

The Naughty Dog engine supports a powerful in-game menu system, allowing developers to control global configuration options and invoke commands. The data types of the configurable options must be relatively simple (primarily Boolean, integer and floating-point variables), but this limitation has not prevented the developers at Naughty Dog from creating literally hundreds of useful menu-driven options.

Each configuration option is implemented as a global variable, or a member of a singleton struct or class. When the menu option that controls an option is created, the address of the variable is provided, and the menu item directly controls its value. As an example, the following function creates a

子菜单项包含一些顽皮狗轨道车辆的选项（这些简单的车辆行驶在样条线上，几乎在每款顽皮狗游戏中都有使用，从《神秘海域：德雷克的宝藏》中的“Out of the Frying Pan”吉普车追逐关卡到《神秘海域 4》中的卡车车队/吉普车追逐序列）。它定义了控制三个全局变量的菜单项：两个布尔值和一个浮点值。这些菜单项被收集到一个菜单中，并返回一个特殊菜单项，选择该菜单项时会显示该菜单项。据推测，调用此函数的代码会将此菜单项添加到它正在构建的父菜单中。

```
DMENU::ItemSubmenu * CreateRailVehicleMenu()
{
    extern bool g_railVehicleDebugDraw2D;
    extern bool g_railVehicleDebugDrawCameraGoals;
    extern float g_railVehicleFlameProbability;

    DMENU::Menu * pMenu
        = new DMENU::Menu("RailVehicle");

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw 2D Spring Graphs",
            DMENU::ToggleBool,
            &g_railVehicleDebugDraw2D));

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw Goals (Untracked)",
            DMENU::ToggleBool,
            &g_railVehicleDebugDrawCameraGoals));

    DMENU::ItemFloat * pItemFloat;
    pItemFloat = new DMENU::ItemFloat(
        "FlameProbability",
        DMENU::EditFloat, 5, "%5.2f",
        &g_railVehicleFlameProbability);

    pItemFloat->SetRangeAndStep(0.0f, 1.0f, 0.1f, 0.01f);
    pMenu->PushBackItem(pItemFloat);

    DMENU::ItemSubmenu * pSubmenuItem;
    pSubmenuItem = new DMENU::ItemSubmenu(
        "RailVehicle...", pMenu);

    return pSubmenuItem;
}
```

当选择相应的菜单项时，只需用 Dualshock 游戏手柄上的圆圈按钮标记即可保存任何选项的值。

submenu item containing some options for Naughty Dog's *rail vehicles* (simple vehicles that ride on splines which have been used in pretty much every Naughty Dog game, from the "Out of the Frying Pan" jeep chase level in *Uncharted: Drake's Fortune* to the truck convoy / jeep chase sequence in *Uncharted 4*). It defines menu items controlling three global variables: two Booleans and one floating-point value. The items are collected onto a menu, and a special item is returned that will bring up the menu when selected. Presumably the code calling this function adds this item to the parent menu that it is building.

```
DMENU::ItemSubmenu * CreateRailVehicleMenu()
{
    extern bool g_railVehicleDebugDraw2D;
    extern bool g_railVehicleDebugDrawCameraGoals;
    extern float g_railVehicleFlameProbability;

    DMENU::Menu * pMenu
        = new DMENU::Menu("RailVehicle");

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw 2D Spring Graphs",
            DMENU::ToggleBool,
            &g_railVehicleDebugDraw2D));

    pMenu->PushBackItem(
        new DMENU::ItemBool("Draw Goals (Untracked)",
            DMENU::ToggleBool,
            &g_railVehicleDebugDrawCameraGoals));

    DMENU::ItemFloat * pItemFloat;
    pItemFloat = new DMENU::ItemFloat(
        "FlameProbability",
        DMENU::EditFloat, 5, "%5.2f",
        &g_railVehicleFlameProbability);

    pItemFloat->SetRangeAndStep(0.0f, 1.0f, 0.1f, 0.01f);
    pMenu->PushBackItem(pItemFloat);

    DMENU::ItemSubmenu * pSubmenuItem;
    pSubmenuItem = new DMENU::ItemSubmenu(
        "RailVehicle...", pMenu);

    return pSubmenuItem;
}
```

The value of any option can be saved by simply marking it with the circle button on the Dualshock joypad when the corresponding menu item is se-

已选择。菜单设置保存在 INI 风格的文本文件中，允许保存的全局变量在游戏多次运行后保留其值。能够控制每个菜单项保存哪些选项非常有用，因为任何未保存的选项都将采用程序员指定的默认值。如果程序员更改了默认值，所有用户都会“看到”新值，除非用户为该特定选项保存了自定义值。

命令行参数

顽皮狗引擎会扫描命令行，查找一组预定义的特殊选项。您可以指定要加载的关卡名称以及其他一些常用参数。

方案数据定义

顽皮狗引擎（用于制作《神秘海域》和《最后生还者》系列）中的绝大多数引擎和游戏配置信息都是使用一种名为 Scheme 的类 Lisp 语言指定的。使用专有数据编译器，Scheme 语言定义的数据结构会被转换为引擎可加载的二进制文件。数据编译器还会生成包含 Scheme 中定义的每种数据类型的 C 结构体声明的头文件。这些头文件使引擎能够正确解释加载的二进制文件中的数据。这些二进制文件甚至可以即时重新编译和加载，允许开发人员修改 Scheme 中的数据并立即查看更改的效果（只要不添加或删除数据成员，因为这需要重新编译引擎）。

下面的示例演示了如何创建一个指定动画属性的数据结构。然后，它将三个不同的动画导出到游戏中。您可能从未读过 Scheme 代码，但对于这个相对简单的例子来说，它应该很容易理解。您会注意到一个奇怪的现象：Scheme 符号中允许使用连字符，因此 simple-animation 是一个单独的符号（这与 C/C++ 中 simple-animation 是两个变量 simple 和 animation 的减法不同）。

简单动画.scm

```
;; Define a new data type called simple-animation.
(deftype simple-animation ()
    (
        (name            string)
        (speed           float   :default 1.0)
```


lected. The menu settings are saved in an INI-style text file, allowing the saved global variables to retain the values across multiple runs of the game. The ability to control which options are saved on a *per-menu-item basis* is highly useful, because any option that is not saved will take on its programmer-specified default value. If a programmer changes a default, all users will “see” the new value, unless of course a user has saved a custom value for that particular option.

Command Line Arguments

The Naughty Dog engine scans the command line for a predefined set of special options. The name of the level to load can be specified, along with a number of other commonly used arguments.

Scheme Data Definitions

The vast majority of engine and game configuration information in the Naughty Dog engine (used to produce the *Uncharted* and *The Last of Us* series) is specified using a Lisp-like language called Scheme. Using a proprietary data compiler, data structures defined in the Scheme language are transformed into binary files that can be loaded by the engine. The data compiler also spits out header files containing C `struct` declarations for every data type defined in Scheme. These header files allow the engine to properly interpret the data contained in the loaded binary files. The binary files can even be recompiled and reloaded on the fly, allowing developers to alter the data in Scheme and see the effects of their changes immediately (as long as data members are not added or removed, as that would require a recompile of the engine).

The following example illustrates the creation of a data structure specifying the properties of an animation. It then exports three unique animations to the game. You may have never read Scheme code before, but for this relatively simple example it should be pretty self-explanatory. One oddity you'll notice is that hyphens are permitted within Scheme symbols, so `simple-animation` is a single symbol (unlike in C/C++ where `simple-animation` would be the subtraction of two variables, `simple` and `animation`).

simple-animation.scm

```
;; Define a new data type called simple-animation.
(deftype simple-animation ()
    (
        (name            string)
        (speed           float   :default 1.0)
```

```
        (fade-in-seconds  float   :default 0.25)
        (fade-out-seconds float   :default 0.25)
    )
)

;; Now define three instances of this data structure...
(define-export anim-walk
    (new simple-animation
        :name "walk"
        :speed 1.0
    )
)

(define-export anim-walk-fast
    (new simple-animation
        :name "walk"
        :speed 2.0
    )
)

(define-export anim-jump
    (new simple-animation
        :name "jump"
        :fade-in-seconds 0.1
        :fade-out-seconds 0.1
    )
)
```

该 Scheme 代码将生成以下 C/C++ 头文件：

简单动画.h

```
// WARNING: This file was automatically generated from
// Scheme. Do not hand-edit.

struct SimpleAnimation
{
    const char* m_name;
    float       m_speed;
    float       m_fadeInSeconds;
    float       m_fadeOutSeconds;
};
```

在游戏中，可以通过调用 LookupSymbol() 函数来读取数据，该函数根据返回的数据类型进行模板化，如下所示：


```
        (fade-in-seconds  float   :default 0.25)
        (fade-out-seconds float   :default 0.25)
    )
)

;; Now define three instances of this data structure...
(define-export anim-walk
    (new simple-animation
        :name "walk"
        :speed 1.0
    )
)

(define-export anim-walk-fast
    (new simple-animation
        :name "walk"
        :speed 2.0
    )
)

(define-export anim-jump
    (new simple-animation
        :name "jump"
        :fade-in-seconds 0.1
        :fade-out-seconds 0.1
    )
)
```

This Scheme code would generate the following C/C++ header file:

simple-animation.h

```
// WARNING: This file was automatically generated from
// Scheme. Do not hand-edit.

struct SimpleAnimation
{
    const char* m_name;
    float       m_speed;
    float       m_fadeInSeconds;
    float       m_fadeOutSeconds;
};
```

In-game, the data can be read by calling the `LookupSymbol()` function, which is templated on the data type returned, as follows:

```
#include "simple-animation.h"
void someFunction()
{
    SimpleAnimation* pWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk"));

    SimpleAnimation* pFastWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk-fast"));

    SimpleAnimation* pJumpAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-jump"));

    // use the data here...
}
```

该系统为程序员提供了极大的灵活性，可以定义各种配置数据——从简单的布尔值、浮点数和字符串选项，到复杂的嵌套互连数据结构。它用于指定详细的动画树、物理参数、玩家机制等等。


```
#include "simple-animation.h"
void someFunction()
{
    SimpleAnimation* pWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk"));

    SimpleAnimation* pFastWalkAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-walk-fast"));

    SimpleAnimation* pJumpAnim
        = LookupSymbol<SimpleAnimation*>(
            SID("anim-jump"));

    // use the data here...
}
```

This system gives the programmers a great deal of flexibility in defining all sorts of configuration data—from simple Boolean, floating-point and string options all the way to complex, nested, interconnected data structures. It is used to specify detailed animation trees, physics parameters, player mechanics and so on.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

7
资源和文件系统

游戏本质上是多媒体体验。因此，游戏引擎需要能够加载和管理各种不同类型的媒体 - 纹理位图、3D 网格数据、动画、音频剪辑、碰撞和物理数据、游戏世界布局等等。此外，由于内存通常很少，游戏引擎需要确保在任何给定时间只有一个媒体文件副本加载到内存中。例如，如果五个网格共享相同的纹理，那么我们希望内存中只有一个该纹理的副本，而不是五个。大多数游戏引擎采用某种资源管理器（又名资产管理器或媒体管理器）来加载和管理构成现代 3 D 游戏的大量资源。

每个资源管理器都大量使用文件系统。在个人计算机上，文件系统通过操作系统调用库暴露给程序员。然而，游戏引擎通常会将原生文件系统 API“包装”在引擎特定的 API 中，主要有两个原因。首先，引擎可能是跨平台的，在这种情况下，游戏引擎的文件系统 API 可以保护软件的其余部分免受不同目标硬件平台之间差异的影响。其次，操作系统的文件系统 API 可能无法提供游戏引擎所需的所有工具。例如，许多引擎支持文件流（即在游戏运行时“动态”加载数据的能力），但大多数操作系统并不提供流文件系统 API。

7
Resources and the File System

Games are by nature multimedia experiences. A game engine therefore needs to be capable of loading and managing a wide variety of different kinds of media—texture bitmaps, 3D mesh data, animations, audio clips, collision and physics data, game world layouts, and the list goes on. Moreover, because memory is usually scarce, a game engine needs to ensure that only one copy of each media file is loaded into memory at any given time. For example, if five meshes share the same texture, then we would like to have only one copy of that texture in memory, not five. Most game engines employ some kind of *resource manager* (a.k.a. *asset manager*, a.k.a. *media manager*) to load and manage the myriad resources that make up a modern 3D game.

Every resource manager makes heavy use of the file system. On a personal computer, the file system is exposed to the programmer via a library of operating system calls. However, game engines often “wrap” the native file system API in an engine-specific API, for two primary reasons. First, the engine might be cross-platform, in which case the game engine’s file system API can shield the rest of the software from differences between different target hardware platforms. Second, the operating system’s file system API might not provide all the tools needed by a game engine. For example, many engines support file *streaming* (i.e., the ability to load data “on the fly” while the game is running), yet most operating systems don’t provide a streaming file system API

开箱即用。主机游戏引擎还需要提供对各种可移动和不可移动介质的访问，从记忆棒到可选硬盘，从 DVD-ROM 或蓝光光盘到网络文件系统（例如 Xbox Live 或 PlayStation Network，PSN）。各种介质之间的差异同样可以“隐藏”在游戏引擎的文件系统 API 背后。

在本章中，我们将首先探讨现代 3D 游戏引擎中常见的各种文件系统 API。然后，我们将了解典型的资源管理器的工作原理。

7.1 文件系统

游戏引擎的文件系统 API 通常涉及以下功能领域：

- 操作文件名和路径，
- 打开、关闭、读取和写入单个文件，
- 扫描目录的内容，以及
- 处理异步文件 I/O 请求（用于流式传输）。

我们将在以下章节中简要介绍这些内容。

7.1.1 文件名和路径

路径是描述文件或目录在文件系统层次结构中位置的字符串。每个操作系统使用的路径格式略有不同，但路径在每个操作系统上的结构基本相同。路径通常采用以下形式：

```
volume/directory1/directory2/.../directoryN/file-name
```

or

```
volume/directory1/directory2/.../directory(N-1)/directoryN
```

换句话说，路径通常由一个可选的卷说明符后跟一系列路径组件组成，这些组件之间由保留的路径分隔符（例如正斜杠或反斜杠（/ 或 \））分隔。每个组件都指定一个目录，该目录沿着从根目录到相关文件或目录的路径延伸。如果路径指定了文件的位置，则路径中的最后一个组件是文件名；否则，它指定目标目录。根目录通常由一个路径表示，该路径由一个可选的卷说明符后跟一个路径分隔符组成（例如，在 UNIX 上为 / ，在 Windows 上为 C:\）。

out of the box. Console game engines also need to provide access to a variety of removable and non-removable media, from memory sticks to optional hard drives to a DVD-ROM or Blu-ray fixed disk to network file systems (e.g., Xbox Live or the PlayStation Network, PSN). The differences between various kinds of media can likewise be “hidden” behind a game engine’s file system API.

In this chapter, we’ll first explore the kinds of file system APIs found in modern 3D game engines. Then we’ll see how a typical resource manager works.

7.1 File System

A game engine’s file system API typically addresses the following areas of functionality:

- manipulating file names and paths,
- opening, closing, reading and writing individual files,
- scanning the contents of a directory, and
- handling asynchronous file I/O requests (for streaming).

We’ll take a brief look at each of these in the following sections.

7.1.1 File Names and Paths

A *path* is a string describing the location of a file or directory within a file system hierarchy. Each operating system uses a slightly different path format, but paths have essentially the same structure on every operating system. A path generally takes the following form:

```
volume/directory1/directory2/.../directoryN/file-name
```

or

```
volume/directory1/directory2/.../directory(N-1)/directoryN
```

In other words, a path generally consists of an optional *volume specifier* followed by a sequence of *path components* separated by a reserved path separator character such as the forward or backward slash (/ or \). Each component names a directory along the route from the root directory to the file or directory in question. If the path specifies the location of a file, the last component in the path is the file name; otherwise it names the target directory. The root directory is usually indicated by a path consisting of the optional volume specifier followed by a single path separator character (e.g., / on UNIX, or `C:\` on Windows).

7.1.1.1 不同操作系统之间的差异

每个操作系统都会在此通用路径结构上引入一些细微的差异。以下是 Microsoft DOS、Microsoft Windows、UNIX 系列操作系统和 Apple Macintosh OS 之间的一些主要区别：

• UNIX 使用正斜杠（/）作为路径分隔符，而 DOS 和旧版本的 Windows 使用反斜杠（\）作为路径分隔符。较新版本的 Windows 允许使用正斜杠或反斜杠来分隔路径，但某些应用程序仍然无法接受正斜杠。

• Mac OS 8 和 9 使用冒号 (:) 作为路径分隔符。Mac OS X 基于 BSD UNIX，因此支持 UNIX 的正斜杠表示法。

• 某些文件系统会区分路径和文件名的大小写（例如 UNIX 及其变体），而其他文件系统则不区分大小写（例如 Windows）。这在开发过程中处理跨多个操作系统的文件或编写跨平台游戏时可能会导致问题。（例如，名为 EnemyAnims.json 的资源文件是否应被视为与名为 enemyanims.json 的资源文件等效？）

• UNIX 及其变体不支持将卷作为单独的目录层次结构。整个文件系统包含在一个单一的整体层次结构中，并且本地磁盘驱动器、网络驱动器和其他资源的挂载方式使其看起来像是主层次结构中的子树。因此，UNIX 路径永远不会包含卷说明符。

• 在 Microsoft Windows 上，卷可以通过两种方式指定。本地磁盘驱动器使用单个字母加冒号（例如，常见的 C:）来指定。远程网络共享可以挂载为本地磁盘，也可以通过卷说明符来引用，该卷说明符由两个反斜杠后跟远程计算机名称以及该计算机上共享目录或资源的名称组成（例如，\\some-computer\some-share）。这种双反斜杠表示法是通用命名约定 (UNC) 的一个示例。

• 在 DOS 和早期版本的 Windows 中，文件名最多可以包含 8 个字符，并带有一个由 3 个字符组成的扩展名，扩展名与主文件名之间用一个点分隔。扩展名描述了文件的类型，例如，.txt 表示文本文件，.exe 表示可执行文件。在最近的 Windows 实现中，文件名可以包含任意数量的点（就像在 UNIX 下一样），但 .txt 之后的字符

7.1.1.1 Differences across Operating Systems

Each operating system introduces slight variations on this general path structure. Here are some of the key differences between Microsoft DOS, Microsoft Windows, the UNIX family of operating systems and Apple Macintosh OS:

- UNIX uses a forward slash (/) as its path component separator, while DOS and older versions of Windows used a backslash (\) as the path separator. Recent versions of Windows allow either forward or backward slashes to be used to separate path components, although some applications still fail to accept forward slashes.
- Mac OS 8 and 9 use the colon (`:`) as the path separator character. Mac OS X is based on BSD UNIX, so it supports UNIX's forward slash notation.
- Some filesystems consider paths and filenames to be case-sensitive (like UNIX and its variants), while others are case-insensitive (like Windows). This can cause problems when dealing with files across multiple operating systems during development, or when writing a cross-platform game. (For example, should an asset file named `EnemyAnims.json` be considered equivalent to an asset file named `enemyanims.json` or not?)
- UNIX and its variants don't support volumes as separate directory hierarchies. The entire file system is contained within a single monolithic hierarchy, and local disk drives, network drives and other resources are *mounted* so that they appear to be subtrees within the main hierarchy. As a result, UNIX paths never have a volume specifier.
- On Microsoft Windows, volumes can be specified in two ways. A local disk drive is specified using a single letter followed by a colon (e.g., the ubiquitous `C:`). A remote network share can either be mounted so that it looks like a local disk, or it can be referenced via a volume specifier consisting of two backslashes followed by the remote computer name and the name of a shared directory or resource on that machine (e.g., `\\some-computer\some-share`). This double backslash notation is an example of the Universal Naming Convention (UNC).
- Under DOS and early versions of Windows, a file name could be up to eight characters in length, with a three-character *extension* which was separated from the main file name by a dot. The extension described the file's type, for example `.txt` for a text file or `.exe` for an executable file. In recent Windows implementations, file names can contain any number of dots (as they can under UNIX), but the characters after the

最后的点仍然被包括 Windows 资源管理器在内的许多应用程序解释为文件的扩展名。

• 每个操作系统都不允许在文件和目录名称中使用某些字符。例如，冒号不能出现在 Windows 或 DOS 路径中的任何位置，除非作为驱动器号卷说明符的一部分。某些操作系统允许这些保留字符的子集出现在路径中，只要该路径被整个引号引起来，或者在违规字符前加上反斜杠或其他保留转义字符进行转义即可。例如，在 Windows 下，文件和目录名称可以包含空格，但在某些情况下，此类路径必须用双引号引起来。

• UNIX 和 Windows 都具有当前工作目录 (CWD)（也称为当前工作目录 (PWD)）的概念。在这两个操作系统上，都可以在命令 shell 中使用 cd（更改目录）命令设置 CWD；在 Windows 下，可以通过输入不带参数的 cd 命令来查询 CWD；在 UNIX 下，可以通过执行 pwd 命令来查询 CWD。在 UNIX 下，只有一个 CWD。在 Windows 下，每个卷都有自己私有的 CWD。

• 支持多个卷的操作系统（例如 Windows）也具有当前工作卷的概念。在 Windows 命令外壳中，可以通过输入驱动器号和冒号，然后按 Enter 键来设置当前卷（例如，C: <Enter>）。

• 游戏机通常还会使用一组预定义的路径前缀来表示多个卷。例如，PlayStation 3 使用前缀 /dev_bdvd/ 来指代蓝光光驱，而 /dev_hdd x / 则指代一个或多个硬盘（其中 x 是设备的索引）。在 PS3 开发套件中，/app_home/ 会映射到用于开发的主机上的用户定义路径。在开发过程中，游戏通常从 /app_home/ 读取其资源，而不是从蓝光光驱或硬盘读取。

7.1.1.2 绝对路径和相对路径

所有路径都是相对于文件系统中某个位置指定的。当路径相对于根目录指定时，我们称之为 绝对路径 。当路径相对于文件系统层次结构中的其他目录指定时，我们称之为 相对路径 。

在 UNIX 和 Windows 系统中，绝对路径都以路径分隔符（/ 或 \）开头，而相对路径则没有前导路径分隔符。在 Windows 系统中，绝对路径和相对路径都可以包含可选的卷

final dot are still interpreted as the file's extension by many applications including the Windows Explorer.

- Each operating system disallows certain characters in the names of files and directories. For example, a colon cannot appear anywhere in a Windows or DOS path except as part of a drive letter volume specifier. Some operating systems permit a subset of these reserved characters to appear in a path as long as the path is quoted in its entirety or the offending character is *escaped* by preceding it with a backslash or some other reserved *escape character*. For example, file and directory names may contain spaces under Windows, but such a path must be surrounded by double quotes in certain contexts.
- Both UNIX and Windows have the concept of a *current working directory* or CWD (also known as the *present working directory* or PWD). The CWD can be set from a command shell via the `cd` (*change directory*) command on both operating systems, and it can be queried by typing `cd` with no arguments under Windows or by executing the `pwd` command on UNIX. Under UNIX there is only one CWD. Under Windows, each volume has its own private CWD.
- Operating systems that support multiple volumes, like Windows, also have the concept of a *current working volume*. From a Windows command shell, the current volume can be set by entering its drive letter and a colon followed by the Enter key (e.g., `C:`<Enter>).
- Consoles often also employ a set of predefined path prefixes to represent multiple volumes. For example, PlayStation 3 uses the prefix `/dev_bdvd/` to refer to the Blu-ray disk drive, while `/dev_hdd`*x*`/` refers to one or more hard disks (where *x* is the index of the device). On a PS3 development kit, `/app_home/` maps to a user-defined path on whatever host machine is being used for development. During development, the game usually reads its assets from `/app_home/` rather than from the Blu-ray or the hard disk.

7.1.1.2 Absolute and Relative Paths

All paths are specified relative to some location within the file system. When a path is specified relative to the root directory, we call it an *absolute path*. When it is relative to some *other* directory in the file system hierarchy, we call it a *relative path*.

Under both UNIX and Windows, absolute paths start with a path separator character (/ or \), while relative paths have no leading path separator. On Windows, both absolute and relative paths may have an optional volume

说明符——如果省略卷，则假定路径指的是当前工作卷。

以下路径都是绝对路径：

视窗
- `C:\Windows\System32`
- D:\（D: 卷上的根目录）
- \（当前工作卷上的根目录）
- \game\assets\animation\walk.anim（当前工作体积）
- \\joe-dell\Shared_Files\Images\foo.jpg（网络路径）UNIX

- `/usr/local/bin/grep`
- `/game/src/audio/effects.cpp`
- /（根目录）以下路径都是相对的

:

视窗
- System32（相对于当前卷上的 CWD \Windows）
- X:animation\walk.anim（相对于 X: 卷上的 CWD \game\assets）UNIX

- bin/grep（相对于 CWD /usr/local ）
- src/audio/effects.cpp（相对于 CWD /game ）

7.1.1.3 搜索路径

请勿将“路径”与“搜索路径”混淆。路径是一个字符串，表示单个文件或目录在文件系统层次结构中的位置。搜索路径是一个包含路径列表的字符串，每个路径之间由特殊字符（例如冒号或分号）分隔，查找文件时会搜索这些路径。例如，当您从命令提示符运行任何程序时，操作系统会通过搜索 Shell 环境变量中包含的搜索路径中的每个目录来查找可执行文件。

一些游戏引擎也使用搜索路径来定位资源文件。例如，OGRE 渲染引擎使用名为 resources.cfg 的文本文件中包含的资源搜索路径。该文件提供了一个简单的目录和 ZIP 压缩包列表，在尝试查找资源时应按顺序搜索这些目录和 ZIP 压缩包。即便如此，在运行时搜索资源是一项耗时的工作。通常，我们完全可以预先知道资源的路径。假设情况确实如此，我们就可以完全避免搜索资源——这显然是一种更优的方法。

specifier—if the volume is omitted, then the path is assumed to refer to the current working volume.

The following paths are all absolute:

Windows
- `C:\Windows\System32`
- `D:\` (root directory on the `D:` volume)
- \ (root directory on the current working volume)
- `\game\assets\animation\walk.anim` (current working volume)
- `\\joe-dell\Shared_Files\Images\foo.jpg` (network path)

UNIX
- `/usr/local/bin/grep`
- `/game/src/audio/effects.cpp`
- / (root directory)

The following paths are all relative:

Windows
- `System32` (relative to CWD `\Windows` on the current volume)
- `X:animation\walk.anim` (relative to CWD `\game\assets` on the `X:` volume)

UNIX
- `bin/grep` (relative to CWD `/usr/local`)
- `src/audio/effects.cpp` (relative to CWD `/game`)

7.1.1.3 Search Paths

The term *path* should not be confused with the term *search path*. A *path* is a string representing the location of a single file or directory within the file system hierarchy. A *search path* is a string containing a list of paths, each separated by a special character such as a colon or semicolon, which is searched when looking for a file. For example, when you run any program from a command prompt, the operating system finds the executable file by searching each directory on the search path contained in the shell's environment variable.

Some game engines also use search paths to locate resource files. For example, the OGRE rendering engine uses a resource search path contained in a text file named `resources.cfg`. The file provides a simple list of directories and ZIP archives that should be searched in order when trying to find an asset. That said, searching for assets at runtime is a time-consuming proposition. Usually there's no reason our assets' paths cannot be known a priori. Presuming this is the case, we can avoid having to search for assets at all—which is clearly a superior approach.

7.1.1.4 路径 API

显然，路径比简单的字符串复杂得多。程序员在处理路径时可能需要做很多事情，例如隔离目录、文件名和扩展名，规范化路径，在绝对路径和相对路径之间来回转换等等。拥有一个功能丰富的 API 来帮助完成这些任务将非常有帮助。

Microsoft Windows 为此提供了一个 API。它由动态链接库 shlwapi.dll 实现，并通过头文件 shlwapi.h 公开。此 API 的完整文档可在 Microsoft 开发者网络 (MSDN) 上找到，网址为：http://msdn2。

microsoft.com/en-us/library/bb773559(VS.85).aspx。

当然，shlwapi API 仅适用于 Win32 平台。索尼也提供了类似的 API 供 PlayStation 3 和 PlayStation 4 使用。但是，在编写跨平台游戏引擎时，我们不能直接使用平台特定的 API。游戏引擎可能并不需要像 shlwapi 这样的 API 提供的所有功能。出于这些原因，游戏引擎通常会实现一个精简的路径处理 API，以满足引擎的特定需求，并使其能够在引擎所针对的每个操作系统上运行。这样的 API 可以作为每个平台上原生 API 的精简包装来实现，也可以从头编写。

7.1.2 基本文件 I/O

C 标准库提供了两种用于打开、读取和写入文件内容的 API——一种是缓冲 API，另一种是非缓冲 API。每个文件 I/O API 都需要称为缓冲区的数据块，作为程序与磁盘文件之间字节传递的源或目标。当文件 I/O API 为您管理必要的输入和输出数据缓冲区时，我们称该 API 为缓冲 API。对于非缓冲 API，使用该 API 的程序员需要负责分配和管理数据缓冲区。C 标准库的缓冲文件 I/O 例程有时被称为流 I/O API，因为它们提供了一种抽象，使磁盘文件看起来像字节流。

表 7.1 列出了缓冲和非缓冲文件 I/O 的 C 标准库函数。

C 标准库 I/O 函数已有详尽的文档，因此我们在此不再赘述。更多信息，请参阅 http://msdn.microsoft.com/en-us/library/c565h7xx.aspx（了解 Microsoft 的缓冲（流 I/O）API 实现）和 http://msdn.microsoft.com/en-us/library/40bbyw78.aspx（了解 Microsoft 的实现）。

7.1.1.4 Path APIs

Clearly, paths are much more complex than simple strings. There are many things a programmer may need to do when dealing with paths, such as isolating the directory, filename and extension, canonicalizing a path, converting back and forth between absolute and relative paths and so on. It can be extremely helpful to have a feature-rich API to help with these tasks.

Microsoft Windows provides an API for this purpose. It is implemented by the dynamic link library `shlwapi.dll`, and it is exposed via the header file `shlwapi.h`. Complete documentation for this API is provided on the Microsoft Developer's Network (MSDN) at the following URL: http://msdn2.microsoft.com/en-us/library/bb773559(VS.85).aspx.

Of course, the `shlwapi` API is only available on Win32 platforms. Sony provides similar APIs for use on the PlayStation 3 and PlayStation 4. But when writing a cross-platform game engine, we cannot use platform-specific APIs directly. A game engine may not need all of the functions provided by an API like `shlwapi` anyway. For these reasons, game engines often implement a stripped-down path-handling API that meets the engine's particular needs and works on every operating system targeted by the engine. Such an API can be implemented as a thin wrapper around the native API on each platform or it can be written from scratch.

7.1.2 Basic File I/O

The C standard library provides two APIs for opening, reading and writing the contents of files—one buffered and the other unbuffered. Every file I/O API requires data blocks known as *buffers* to serve as the source or destination of the bytes passing between the program and the file on disk. We say a file I/O API is *buffered* when the API manages the necessary input and output data buffers for you. With an unbuffered API, it is the responsibility of the programmer using the API to allocate and manage the data buffers. The C standard library's buffered file I/O routines are sometimes referred to as the *stream I/O* API, because they provide an abstraction which makes disk files look like streams of bytes.

The C standard library functions for buffered and unbuffered file I/O are listed in Table 7.1.

The C standard library I/O functions are well-documented, so we will not repeat detailed documentation for them here. For more information, please refer to http://msdn.microsoft.com/en-us/library/c565h7xx.aspx for Microsoft's implementation of the buffered (stream I/O) API, and to http://msdn.microsoft.com/en-us/library/40bbyw78.aspx for Microsoft's imple-

手术	缓冲 API	无缓冲 API
打开文件	`fopen()`	`open()`
关闭文件	`fclose()`	`close()`
从文件读取	`fread()`	`read()`
写入文件	`fwrite()`	`write()`
寻找偏移量	`fseek()`	`seek()`
返回电流偏移	`ftell()`	`tell()`
读一行	`fgets()`	无
写一行	`fputs()`	无
读取格式化的字符串	`fscanf()`	无
写入格式化字符串	`fprintf()`	无
查询文件状态	`fstat()`	`stat()`

表 7.1. C 标准库中的缓冲和非缓冲文件操作。

无缓冲（低级 I/O）API 的说明。

在 UNIX 及其变体中，C 标准库的无缓冲 I/O 路由是原生操作系统调用。然而，在 Microsoft Windows 上，这些例程仅仅是对更低级 API 的包装。Win32 函数 CreateFile() 用于创建或打开文件进行写入或读取，ReadFile() 和 WriteFile() 分别用于读取和写入数据，CloseFile() 用于关闭打开的文件句柄。与 C 标准库函数相比，使用低级系统调用的优势在于它们可以公开原生文件系统的所有细节。例如，使用 Windows 原生 API 时，您可以查询和控制文件的安全属性——而这是 C 标准库无法实现的。

一些游戏团队发现管理自己的缓冲区很有用。例如，EA 的《红色警戒 3》团队发现，将数据写入日志文件会导致性能显著下降。他们修改了日志系统，使其输出累积到内存缓冲区中，只有当缓冲区填满时才将其写入磁盘。然后，他们将缓冲区转储例程移至单独的线程，以避免拖慢主游戏循环。

7.1.2.1 是否包装

游戏引擎可以使用 C 标准库的文件 I/O 函数或操作系统的原生 API 来编写。然而，许多游戏引擎将文件 I/O API 封装在自定义 I/O 函数库中。封装操作系统的 I/O API 至少有三个优点。首先，即使原生库在特定平台上不一致或存在错误，引擎程序员也可以保证所有目标平台上的行为一致。

Operation	Buffered API	Unbuffered API
Open a file	`fopen()`	`open()`
Close a file	`fclose()`	`close()`
Read from a file	`fread()`	`read()`
Write to a file	`fwrite()`	`write()`
Seek to an offset	`fseek()`	`seek()`
Return current offset	`ftell()`	`tell()`
Read a single line	`fgets()`	n/a
Write a single line	`fputs()`	n/a
Read formatted string	`fscanf()`	n/a
Write formatted string	`fprintf()`	n/a
Query file status	`fstat()`	`stat()`

Table 7.1. Buffered and unbuffered file operations in the C standard library.

mentation of the unbuffered (low-level I/O) API.

On UNIX and its variants, the C standard library's unbuffered I/O routes are native operating system calls. However, on Microsoft Windows these routines are merely wrappers around an even lower-level API. The Win32 function `CreateFile()` creates or opens a file for writing or reading, `ReadFile()` and `WriteFile()` read and write data, respectively, and `CloseFile()` closes an open file handle. The advantage to using low-level system calls as opposed to C standard library functions is that they expose all of the details of the native file system. For example, you can query and control the security attributes of files when using the Windows native API—something you cannot do with the C standard library.

Some game teams find it useful to manage their own buffers. For example, the *Red Alert 3* team at Electronic Arts observed that writing data into log files was causing significant performance degradation. They changed the logging system so that it accumulated its output into a memory buffer, writing the buffer out to disk only when it was filled. Then they moved the buffer dump routine out into a separate thread to avoid stalling the main game loop.

7.1.2.1 To Wrap or Not to Wrap

A game engine can be written to use the C standard library's file I/O functions or the operating system's native API. However, many game engines *wrap* the file I/O API in a library of custom I/O functions. There are at least three advantages to wrapping the operating system's I/O API. First, the engine programmers can guarantee identical behavior across all target platforms, even when native libraries are inconsistent or buggy on a particular platform.

其次，API 可以精简到只包含引擎实际需要的功能，从而最大限度地减少维护工作。第三，可以提供扩展功能。例如，引擎的自定义包装器 API 可能能够处理硬盘上的文件、主机上的 DVDROM 或蓝光光盘、网络上的文件（例如，由 Xbox Live 或 PSN 管理的远程文件），以及 U 盘或其他可移动媒体上的文件。

7.1.2.2 同步文件 I/O

标准 C 库中的两个文件 I/O 库都是同步的，这意味着发出 I/O 请求的程序必须等到数据完全传输到媒体设备或从媒体设备传输出去后才能继续执行。以下代码片段演示了如何使用同步 I/O 函数 fread() 将文件的全部内容读入内存缓冲区。请注意，函数 syncReadFile() 直到所有数据都读入提供的缓冲区后才会返回。

```
bool syncReadFile(const char* filePath,
                  U8* buffer,
                  size_t bufferSize,
                  size_t& rBytesRead)
{
    FILE* handle = fopen(filePath, "rb");
    if (handle)
    {
        // BLOCK here until all data has been read.
        size_t bytesRead = fread(buffer, 1,
                                 bufferSize, handle);

        int err = ferror(handle); // get error if any

        fclose(handle);

        if (0 == err)
        {
            rBytesRead = bytesRead;
            return true;
        }
    }
    rBytesRead = 0;
    return false;
}

void main(int argc, const char* argv[])
{
```


Second, the API can be simplified down to only those functions actually required by the engine, which keeps maintenance efforts to a minimum. Third, extended functionality can be provided. For example, the engine's custom wrapper API might be capable of dealing with files on a hard disk, a DVD-ROM or Blu-ray disk on a console, files on a network (e.g., remote files managed by Xbox Live or PSN), and also with files on memory sticks or other kinds of removable media.

7.1.2.2 Synchronous File I/O

Both of the standard C library's file I/O libraries are *synchronous*, meaning that the program making the I/O request must wait until the data has been completely transferred to or from the media device before continuing. The following code snippet demonstrates how the entire contents of a file might be read into an in-memory buffer using the synchronous I/O function `fread()`. Notice how the function `syncReadFile()` does not return until all the data has been read into the buffer provided.

```
bool syncReadFile(const char* filePath,
                  U8* buffer,
                  size_t bufferSize,
                  size_t& rBytesRead)
{
    FILE* handle = fopen(filePath, "rb");
    if (handle)
    {
        // BLOCK here until all data has been read.
        size_t bytesRead = fread(buffer, 1,
                                 bufferSize, handle);

        int err = ferror(handle); // get error if any

        fclose(handle);

        if (0 == err)
        {
            rBytesRead = bytesRead;
            return true;
        }
    }
    rBytesRead = 0;
    return false;
}

void main(int argc, const char* argv[])
{
```

```
        U8 testBuffer[512];
        size_t bytesRead = 0;

        if (syncReadFile("C:\\testfile.bin",
                         testBuffer, sizeof(testBuffer),
                         bytesRead))
        {
            printf("success: read %u bytes\n", bytesRead);
            // contents of buffer can be used here...
        }
    }
```

7.1.3 异步文件 I/O

流式传输是指在主程序持续运行的同时，在后台加载数据的行为。许多游戏通过在游戏过程中从 DVD-ROM、蓝光光盘或硬盘驱动器流式传输即将开启的关卡数据，为玩家提供无缝、无加载画面的游戏体验。音频和纹理数据可能是最常见的流式传输数据类型，但任何类型的数据都可以流式传输，包括几何图形、关卡布局和动画剪辑。

为了支持流式传输，我们必须使用异步文件 I/O 库，即允许程序在满足其 I/O 请求的同时继续运行的库。某些操作系统提供了开箱即用的异步文件 I/O 库。例如，Windows 公共语言运行时（CLR，Visual BASIC、C#、托管 C++ 和 J# 等语言在其上实现的虚拟机）提供了 System.IO.BeginRead() 和 System.IO.BeginWrite() 等函数。PlayStation 3 和 PlayStation 4 提供了称为 fios 的异步 API。如果您的目标平台没有可用的异步文件 I/O 库，您可以自己编写一个。即使您不必从头开始编写，包装系统 API 以实现可移植性可能也是一个好主意。

以下代码片段演示了如何使用异步读取操作将文件的全部内容读入内存缓冲区。请注意，asyncReadFile() 函数会立即返回——直到 I/O 库调用回调函数 asyncRead Complete() 后，数据才会出现在缓冲区中。

```
AsyncRequestHandle g_hRequest; // async I/O request handle
U8 g_asyncBuffer[512];         // input buffer

static void asyncReadComplete(AsyncRequestHandle hRequest);
```



```
        U8 testBuffer[512];
        size_t bytesRead = 0;

        if (syncReadFile("C:\\testfile.bin",
                         testBuffer, sizeof(testBuffer),
                         bytesRead))
        {
            printf("success: read %u bytes\n", bytesRead);
            // contents of buffer can be used here...
        }
    }
```

7.1.3 Asynchronous File I/O

Streaming refers to the act of loading data in the background while the main program continues to run. Many games provide the player with a seamless, load-screen-free playing experience by streaming data for upcoming levels from the DVD-ROM, Blu-ray disk or hard drive while the game is being played. Audio and texture data are probably the most commonly streamed types of data, but any type of data can be streamed, including geometry, level layouts and animation clips.

In order to support streaming, we must utilize an *asynchronous* file I/O library, i.e., one which permits the program to continue to run while its I/O requests are being satisfied. Some operating systems provide an asynchronous file I/O library out of the box. For example, the Windows Common Language Runtime (CLR, the virtual machine upon which languages like Visual BASIC, C#, managed C++ and J# are implemented) provides functions like `System.IO.BeginRead()` and `System.IO.BeginWrite()`. An asynchronous API known as `fios` is available for the PlayStation 3 and PlayStation 4. If an asynchronous file I/O library is not available for your target platform, it is possible to write one yourself. And even if you don't have to write it from scratch, it's probably a good idea to wrap the system API for portability.

The following code snippet demonstrates how the entire contents of a file might be read into an in-memory buffer using an asynchronous read operation. Notice that the `asyncReadFile()` function returns immediately—the data is not present in the buffer until our callback function `asyncReadComplete()` has been called by the I/O library.

```
AsyncRequestHandle g_hRequest; // async I/O request handle
U8 g_asyncBuffer[512];         // input buffer

static void asyncReadComplete(AsyncRequestHandle hRequest);
```

```
void main(int argc, const char* argv[])
{
    // NOTE: This call to asyncOpen() might itself be an
    // asynchronous call, but we'll ignore that detail
    // here and just assume it's a blocking function.

    AsyncFileHandle hFile = asyncOpen(
        "C:\\testfile.bin");

    if (hFile)
    {
        // This function requests an I/O read, then
        // returns immediately (non-blocking).
        g_hRequest = asyncReadFile(
            hFile,                  // file handle
            g_asyncBuffer,          // input buffer
            sizeof(g_asyncBuffer),  // size of buffer
            asyncReadComplete);     // callback function
    }

    // Now go on our merry way...
    // (This loop simulates doing real work while we wait
    // for the I/O read to complete.)

    for (;;)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }
}

// This function will be called when the data has been read.
static void asyncReadComplete(AsyncRequestHandle hRequest)
{
    if (hRequest == g_hRequest
    && asyncWasSuccessful(hRequest))
    {
        // The data is now present in g_asyncBuffer[] and
        // can be used. Query for the number of bytes
        // actually read:
        size_t bytes = asyncGetBytesReadOrWritten(
                           hRequest);

        char msg[256];
        snprintf(msg, sizeof(msg),
            "async success, read %u bytes\n",
```



```
void main(int argc, const char* argv[])
{
    // NOTE: This call to asyncOpen() might itself be an
    // asynchronous call, but we'll ignore that detail
    // here and just assume it's a blocking function.

    AsyncFileHandle hFile = asyncOpen(
        "C:\\testfile.bin");

    if (hFile)
    {
        // This function requests an I/O read, then
        // returns immediately (non-blocking).
        g_hRequest = asyncReadFile(
            hFile,                  // file handle
            g_asyncBuffer,          // input buffer
            sizeof(g_asyncBuffer),  // size of buffer
            asyncReadComplete);     // callback function
    }

    // Now go on our merry way...
    // (This loop simulates doing real work while we wait
    // for the I/O read to complete.)

    for (;;)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }
}

// This function will be called when the data has been read.
static void asyncReadComplete(AsyncRequestHandle hRequest)
{
    if (hRequest == g_hRequest
    && asyncWasSuccessful(hRequest))
    {
        // The data is now present in g_asyncBuffer[] and
        // can be used. Query for the number of bytes
        // actually read:
        size_t bytes = asyncGetBytesReadOrWritten(
                           hRequest);

        char msg[256];
        snprintf(msg, sizeof(msg),
            "async success, read %u bytes\n",
```

```
            bytes);
        OutputDebugString(msg);
    }
}
```

大多数异步 I/O 库允许主程序在发出请求后等待一段时间，直到 I/O 操作完成。这在需要等待的 I/O 请求结果之前只能完成有限工作的情况下非常有用。以下代码片段演示了这一点。

```
U8 g_asyncBuffer[512];  // input buffer

void main(int argc, const char* argv[])
{
    AsyncRequestHandle hRequest = ASYNC_INVALID_HANDLE;
    AsyncFileHandle hFile = asyncOpen(
                                "C:\\testfile.bin");

    if (hFile)
    {
        // This function requests an I/O read, then
        // returns immediately (non-blocking).
        hRequest = asyncReadFile(
            hFile,                  // file handle
            g_asyncBuffer,          // input buffer
            sizeof(g_asyncBuffer),  // size of buffer
            nullptr);                  // no callback
    }

    // Now do some limited amount of work...
    for (int i = 0; i < 10; i++)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }

    // We can't do anything further until we have that
    // data, so wait for it here.
    asyncWait(hRequest);

    if (asyncWasSuccessful(hRequest))
    {
        // The data is now present in g_asyncBuffer[] and
        // can be used. Query for the number of bytes
        // actually read:
        size_t bytes = asyncGetBytesReadOrWritten(
                                        hRequest);
```



```
            bytes);
        OutputDebugString(msg);
    }
}
```

Most asynchronous I/O libraries permit the main program to wait for an I/O operation to complete some time after the request was made. This can be useful in situations where only a limited amount of work can be done before the results of a pending I/O request are needed. This is illustrated in the following code snippet.

```
U8 g_asyncBuffer[512];  // input buffer

void main(int argc, const char* argv[])
{
    AsyncRequestHandle hRequest = ASYNC_INVALID_HANDLE;
    AsyncFileHandle hFile = asyncOpen(
                                "C:\\testfile.bin");

    if (hFile)
    {
        // This function requests an I/O read, then
        // returns immediately (non-blocking).
        hRequest = asyncReadFile(
            hFile,                  // file handle
            g_asyncBuffer,          // input buffer
            sizeof(g_asyncBuffer),  // size of buffer
            nullptr);                  // no callback
    }

    // Now do some limited amount of work...
    for (int i = 0; i < 10; i++)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }

    // We can't do anything further until we have that
    // data, so wait for it here.
    asyncWait(hRequest);

    if (asyncWasSuccessful(hRequest))
    {
        // The data is now present in g_asyncBuffer[] and
        // can be used. Query for the number of bytes
        // actually read:
        size_t bytes = asyncGetBytesReadOrWritten(
                                        hRequest);
```

```
            char msg[256];
            snprintf(msg, sizeof(msg),
                "async success, read %u bytes\n",
                bytes);
            OutputDebugString(msg);
        }
    }
```

一些异步 I/O 库允许程序员预估某个异步操作需要多长时间才能完成。一些 API 还允许你为请求设置截止时间（这可以有效地提高该请求相对于其他待处理请求的优先级），并指定当请求错过截止时间时将发生什么（例如，取消请求、通知程序并继续尝试等）。

7.1.3.1 优先级

重要的是要记住，文件 I/O 是一个实时系统，与游戏的其他部分一样受截止期限的限制。因此，异步 I/O 操作通常具有不同的优先级。例如，如果我们从硬盘或蓝光光盘流式传输音频并即时播放，则加载下一个充满音频数据的缓冲区的优先级显然高于加载纹理或游戏关卡的某个数据块。异步 I/O 系统必须能够暂停低优先级的请求，以便高优先级的 I/O 请求有机会在其截止期限内完成。

7.1.3.2 异步文件 I/O 的工作原理

异步文件 I/O 的工作原理是在单独的线程中处理 I/O 请求。主线程调用一些函数，这些函数只是将请求放入队列，然后立即返回。同时，I/O 线程从队列中获取请求，并使用 read() 或 fread() 等阻塞 I/O 例程按顺序处理它们。请求完成后，将调用主线程提供的回调函数，通知它操作已完成。如果主线程选择等待 I/O 请求完成，则通过信号量来处理。（每个请求都有一个关联的信号量，主线程可以进入休眠状态，等待 I/O 线程在请求□□完成后发出信号量信号。有关信号量的更多信息，请参见第 4.6.4 节。）

几乎任何你能想到的同步操作都可以通过将代码移到单独的线程中或在物理上独立的处理器上运行（例如在 PlayStation 4 上的一个 CPU 内核上）来转换为异步操作。有关更多详细信息，请参阅第 8.6 节。


```
            char msg[256];
            snprintf(msg, sizeof(msg),
                "async success, read %u bytes\n",
                bytes);
            OutputDebugString(msg);
        }
    }
```

Some asynchronous I/O libraries allow the programmer to ask for an estimate of how long a particular asynchronous operation will take to complete. Some APIs also allow you to set *deadlines* on a request (which effectively prioritizes the request relative to other pending requests), and to specify what happens when a request misses its deadline (e.g., cancel the request, notify the program and keep trying, etc.)

7.1.3.1 Priorities

It's important to remember that file I/O is a real-time system, subject to deadlines just like the rest of the game. Therefore, asynchronous I/O operations often have varying priorities. For example, if we are streaming audio from the hard disk or Blu-ray and playing it on the fly, loading the next buffer-full of audio data is clearly higher priority than, say, loading a texture or a chunk of a game level. Asynchronous I/O systems must be capable of suspending lower-priority requests, so that higher-priority I/O requests have a chance to complete within their deadlines.

7.1.3.2 How Asynchronous File I/O Works

Asynchronous file I/O works by handling I/O requests in a separate thread. The main thread calls functions that simply place requests on a queue and then return immediately. Meanwhile, the I/O thread picks up requests from the queue and handles them sequentially using blocking I/O routines like `read()` or `fread()`. When a request is completed, a callback provided by the main thread is called, thereby notifying it that the operation is done. If the main thread chooses to wait for an I/O request to complete, this is handled via a *semaphore*. (Each request has an associated semaphore, and the main thread can put itself to sleep waiting for that semaphore to be signaled by the I/O thread upon completion of the request. See Section 4.6.4 for more on semaphores.)

Virtually *any* synchronous operation you can imagine can be transformed into an asynchronous operation by moving the code into a separate thread—or by running it on a physically separate processor, such as on one of the CPU cores on the PlayStation 4. See Section 8.6 for more details.

7.2 资源管理器

每个游戏都由各种各样的资源（有时称为资源或媒体）构成。例如，网格、材质、纹理、着色器程序、动画、音频剪辑、关卡布局、碰撞基元、物理参数等等。游戏资源必须进行管理，这既包括用于创建资源的离线工具，也包括运行时的加载、卸载和操作。因此，每个游戏引擎都拥有某种类型的资源管理器。

每个资源管理器都由两个独立但集成的组件组成。一个组件管理用于创建资源并将其转换为引擎可用形式的离线工具链。另一个组件在运行时管理资源，确保在游戏需要之前将资源加载到内存中，并在不再需要时将其从内存中卸载。

在某些引擎中，资源管理器是一个设计简洁、统一、集中的子系统，用于管理游戏使用的所有类型的资源。在其他引擎中，资源管理器本身并非作为单个子系统存在，而是分布在一系列不同的子系统中，这些子系统可能是由不同的人在引擎漫长而丰富多彩的历史中不同时期编写的。但无论如何实现，资源管理器总是承担着某些职责，并解决一系列众所周知的问题。在本节中，我们将探讨典型游戏引擎资源管理器的功能和一些实现细节。

7.2.1 离线资源管理和工具链

7.2.1.1 资产的修订控制

在小型游戏项目中，可以通过将松散的文件存储在具有临时目录结构的共享网络驱动器上来管理游戏资源。但对于包含大量且种类繁多的资源的现代商业 3D 游戏来说，这种方法并不可行。对于这样的项目，团队需要一种更规范的方式来跟踪和管理资源。

一些游戏团队使用源代码版本控制系统来管理他们的资源。美术源文件（Maya 场景、Photoshop PSD 文件、Illustrator 文件等）由美术师提交到 Perforce 或类似的软件包中。这种方法效果不错，尽管有些游戏团队会构建自定义资产管理工具，以帮助降低美术师的学习难度。这些工具可能是对商业版本控制系统的简单包装，也可能是完全定制的。

7.2 The Resource Manager

Every game is constructed from a wide variety of *resources* (sometimes called *assets* or *media*). Examples include meshes, materials, textures, shader programs, animations, audio clips, level layouts, collision primitives, physics parameters, and the list goes on. A game's resources must be managed, both in terms of the offline tools used to create them, and in terms of loading, unloading and manipulating them at runtime. Therefore, every game engine has a *resource manager* of some kind.

Every resource manager is comprised of two distinct but integrated components. One component manages the chain of offline tools used to create the assets and transform them into their engine-ready form. The other component manages the resources at runtime, ensuring that they are loaded into memory in advance of being needed by the game and making sure they are unloaded from memory when no longer needed.

In some engines, the resource manager is a cleanly designed, unified, centralized subsystem that manages all types of resources used by the game. In other engines, the resource manager doesn't exist as a single subsystem per se, but rather is spread across a disparate collection of subsystems, perhaps written by different individuals at various times over the engine's long and sometimes colorful history. But no matter how it is implemented, a resource manager invariably takes on certain responsibilities and solves a well-understood set of problems. In this section, we'll explore the functionality and some of the implementation details of a typical game engine resource manager.

7.2.1 Offline Resource Management and the Tool Chain

7.2.1.1 Revision Control for Assets

On a small game project, the game's assets can be managed by keeping loose files sitting around on a shared network drive with an ad hoc directory structure. This approach is not feasible for a modern commercial 3D game, comprised of a massive number and variety of assets. For such a project, the team requires a more formalized way to track and manage its assets.

Some game teams use a source code revision control system to manage their resources. Art source files (Maya scenes, Photoshop PSD files, Illustrator files, etc.) are checked in to Perforce or a similar package by the artists. This approach works reasonably well, although some game teams build custom asset management tools to help flatten the learning curve for their artists. Such tools may be simple wrappers around a commercial revision control system, or they might be entirely custom.

处理数据大小

美术资源版本控制的最大问题之一是数据量巨大。C++ 和脚本源代码文件虽然较小，但相对于它们对项目的影响而言，美术文件往往要大得多。由于许多源代码控制系统的工作原理是将文件从中央存储库复制到用户本地计算机，因此庞大的资源文件可能会使这些包几乎完全失效。

我见过不同工作室针对这个问题采用过许多不同的解决方案。有些工作室会选择 Alienbrain 等专门设计用于处理海量数据的商业版本控制系统。有些团队则干脆“自食其果”，让他们的版本控制工具在本地复制资源。只要磁盘空间足够大，网络带宽充足，这种方法就能奏效，但效率低下，还会拖慢团队速度。有些团队会在其版本控制工具的基础上构建复杂的系统，以确保特定最终用户只能获得其实际需要的文件的本地副本。在这种模式下，用户要么无法访问存储库的其余部分，要么在需要时通过共享网络驱动器访问。

在顽皮狗，我们使用一个专有工具，该工具利用 UNIX 符号链接几乎消除了数据复制，同时允许每个用户拥有资产存储库的完整本地视图。只要文件未检出进行编辑，它就是指向共享网络驱动器上主文件的符号链接。符号链接在本地磁盘上占用的空间很小，因为它只不过是一个目录条目。当用户检出文件进行编辑时，符号链接会被删除，文件的本地副本会替换它。当用户完成编辑并检入文件时，本地副本将成为新的主副本，其修订历史记录会在主数据库中更新，本地文件也会变回符号链接。这个系统运行良好，但它需要团队从头构建自己的修订控制系统；我还没听说过有哪个商业工具可以像这样工作。此外，符号链接是 UNIX 的一个特性——这样的工具可能可以用 Windows 连接（Windows 中与符□□号链接等同的）来构建，但我还没有看到有人尝试过。

7.2.1.2 资源数据库

正如我们将在下一节深入探讨的那样，大多数资源并非以其原始格式被游戏引擎使用。它们需要经过某种资源调节管道，其作用是将资源转换为引擎所需的二进制格式。对于每个经过

Dealing with Data Size

One of the biggest problems in the revision control of art assets is the sheer amount of data. Whereas C++ and script source code files are small, relative to their impact on the project, art files tend to be much, much larger. Because many source control systems work by copying files from the central repository down to the user's local machine, the sheer size of the asset files can render these packages almost entirely useless.

I've seen a number of different solutions to this problem employed at various studios. Some studios turn to commercial revision control systems like Alienbrain that have been specifically designed to handle very large data sizes. Some teams simply "take their lumps" and allow their revision control tool to copy assets locally. This can work, as long as your disks are big enough and your network bandwidth sufficient, but it can also be inefficient and slow the team down. Some teams build elaborate systems on top of their revision control tool to ensure that a particular end user only gets local copies of the files he or she actually needs. In this model, the user either has no access to the rest of the repository or can access it on a shared network drive when needed.

At Naughty Dog we use a proprietary tool that makes use of UNIX symbolic links to virtually eliminate data copying, while permitting each user to have a complete local view of the asset repository. As long as a file is not checked out for editing, it is a symlink to a master file on a shared network drive. A symbolic link occupies very little space on the local disk, because it is nothing more than a directory entry. When the user checks out a file for editing, the symlink is removed, and a local copy of the file replaces it. When the user is done editing and checks the file in, the local copy becomes the new master copy, its revision history is updated in a master database, and the local file turns back into a symlink. This system works very well, but it requires the team to build their own revision control system from scratch; I am unaware of any commercial tool that works like this. Also, symbolic links are a UNIX feature—such a tool could probably be built with Windows junctions (the Windows equivalent of a symbolic link), but I haven't seen anyone try it as yet.

7.2.1.2 The Resource Database

As we'll explore in depth in the next section, most assets are not used in their original format by the game engine. They need to pass through some kind of asset conditioning pipeline, whose job it is to convert the assets into the binary format needed by the engine. For every resource that passes through

在资源调节管线中，有一些元数据描述了该资源应该如何处理。压缩纹理位图时，我们需要知道哪种压缩类型最适合该特定图像。导出动画时，我们需要知道应该导出 Maya 中哪些范围的帧。从包含多个角色的 Maya 场景中导出角色网格时，我们需要知道哪个网格对应于游戏中的哪个角色。

为了管理所有这些元数据，我们需要某种数据库。如果我们制作的是一款非常小的游戏，这个数据库可能就存储在开发者自己的大脑中。我现在都能听到他们说："记住：玩家的动画需要设置'翻转X'标志，但其他角色不能设置它……或者……老鼠……是不是反过来？"

显然，对于任何规模可观的游戏，我们根本无法以这种方式依赖开发人员的记忆。首先，庞大的资源数量很快就会让人应接不暇。手动处理单个资源文件也过于耗时，在成熟的商业游戏制作中并不实用。因此，每个专业的游戏团队都拥有某种半自动化的资源流程，而驱动这些流程的数据则存储在某种资源数据库中。

在不同的游戏引擎中，资源数据库的形式差异巨大。在某个引擎中，描述资源构建方式的元数据可能嵌入到源资源本身中（例如，它可能以所谓的盲数据存储在 Maya 文件中）。在另一个引擎中，每个源资源文件可能附带一个小的文本文件，用于描述如何处理它。还有一些引擎将其资源构建元数据编码到一组 XML 文件中，这些文件可能被封装在某种自定义的图形用户界面中。有些引擎使用真正的关系数据库，例如 Microsoft Access、MySQL，甚至可能是像 Oracle 这样的重量级数据库。

无论其形式如何，资源数据库都必须提供以下基本功能：

- 能够以某种一致的方式处理多种类型的资源，理想情况下（但肯定不一定）。
- 创造新资源的能力。
- 删除资源的能力。
- 检查和修改现有资源的能力。
- 能够将资源的源文件从磁盘上的一个位置移动到另一个位置。（这非常有用，因为艺术家和游戏设计师

the asset conditioning pipeline, there is some amount of *metadata* that describes *how* that resource should be processed. When compressing a texture bitmap, we need to know what *type* of compression best suits that particular image. When exporting an animation, we need to know what range of frames in Maya should be exported. When exporting character meshes out of a Maya scene containing multiple characters, we need to know which mesh corresponds to which character in the game.

To manage all of this metadata, we need some kind of database. If we are making a very small game, this database might be housed in the brains of the developers themselves. I can hear them now: "Remember: the player's animations need to have the 'flip X' flag set, but the other characters must *not* have it set…or…rats…is it the other way around?"

Clearly for any game of respectable size, we simply cannot rely on the memories of our developers in this manner. For one thing, the sheer volume of assets becomes overwhelming quite quickly. Processing individual resource files by hand is also far too time-consuming to be practical on a full-fledged commercial game production. Therefore, every professional game team has some kind of semiautomated resource pipeline, and the data that drive the pipeline is stored in some kind of *resource database*.

The resource database takes on vastly different forms in different game engines. In one engine, the metadata describing how a resource should be built might be embedded into the source assets themselves (e.g., it might be stored as so-called blind data within a Maya file). In another engine, each source resource file might be accompanied by a small text file that describes how it should be processed. Still other engines encode their resource building metadata in a set of XML files, perhaps wrapped in some kind of custom graphical user interface. Some engines employ a true relational database, such as Microsoft Access, MySQL or conceivably even a heavyweight database like Oracle.

Whatever its form, a resource database must provide the following basic functionality:

- The ability to deal with multiple *types* of resources, ideally (but certainly not necessarily) in a somewhat consistent manner.
- The ability to create new resources.
- The ability to delete resources.
- The ability to inspect and modify existing resources.
- The ability to move a resource's source file(s) from one location to another on-disk. (This is very helpful because artists and game designers

通常需要重新安排资产以反映□□不断变化的项目目标、重新考虑游戏设计、功能的添加和削减等）

• 资源能够交叉引用其他资源（例如，网格使用的材质，或17级所需的动画集合）。这些交叉引用通常会在运行时驱动资源构建过程和加载过程。

• 能够维护数据库内所有交叉引用的参照完整性，并在执行所有常见操作（例如删除或移动资源）时保持参照完整性。

• 能够维护修订历史记录，包括每次更改的人员和原因的记录。

• 如果资源数据库支持多种搜索或查询方式，也会非常有帮助。例如，开发者可能想知道某个动画在哪些关卡中使用，或者一组材质引用了哪些纹理。或者，他们可能只是想找到一个暂时想不起名字的资源。

从上面的列表可以看出，创建一个可靠且强大的资源数据库并非易事。如果设计合理、实施到位，资源数据库甚至可以决定一个团队能否成功推出热门游戏，还是一个团队在项目进行18个月的停滞不前后，最终被管理层逼着放弃项目（甚至更糟）。我深知这一点，因为我亲身经历过这两种情况。

7.2.1.3 一些成功的资源数据库设计

每个游戏团队在设计资源数据库时都会有不同的需求和决策。不过，以下是我亲身体验过的一些行之有效的设计，仅供参考。

虚幻引擎 4

虚幻引擎的资源数据库由其超级工具 UnrealEd 管理。UnrealEd 几乎负责所有工作，从资源元数据管理到资源创建、关卡布局等等。UnrealEd 有其缺点，但它最大的优势在于它是游戏引擎本身的一部分。这使得创建资源后可以立即查看其完整效果，与游戏内的实际效果完全相同。甚至可以在 UnrealEd 内部运行游戏，以便在自然环境中可视化资源，并查看它们在游戏中是否以及如何运行。

often need to rearrange assets to reflect changing project goals, rethinking of game designs, feature additions and cuts, etc.)

- The ability of a resource to cross-reference other resources (e.g., the material used by a mesh, or the collection of animations needed by level 17). These cross-references typically drive both the resource building process and the loading process at runtime.
- The ability to maintain *referential integrity* of all cross-references within the database and to do so in the face of all common operations such as deleting or moving resources around.
- The ability to maintain a revision history, complete with a log of who made each change and why.
- It is also very helpful if the resource database supports searching or querying in various ways. For example, a developer might want to know in which levels a particular animation is used or which textures are referenced by a set of materials. Or they might simply be trying to find a resource whose name momentarily escapes them.

It should be pretty obvious from looking at the above list that creating a reliable and robust resource database is no small task. When designed well and implemented properly, the resource database can quite literally make the difference between a team that ships a hit game and a team that spins its wheels for 18 months before being forced by management to abandon the project (or worse). I know this to be true, because I've personally experienced both.

7.2.1.3 Some Successful Resource Database Designs

Every game team will have different requirements and make different decisions when designing their resource database. However, for what it's worth, here are some designs that have worked well in my own experience.

Unreal Engine 4

Unreal's resource database is managed by their über-tool, UnrealEd. UnrealEd is responsible for literally everything, from resource metadata management to asset creation to level layout and more. UnrealEd has its drawbacks, but its single biggest benefit is that UnrealEd is a part of the game engine itself. This permits assets to be created and then immediately viewed in their full glory, exactly as they will appear in-game. The game can even be run from within UnrealEd in order to visualize the assets in their natural surroundings and see if and how they work in-game.

图 7.1.UnrealEd 的通用浏览器。

UnrealEd 的另一大优势是我所说的一站式购物。UnrealEd 的通用浏览器（如图 7.1 所示）允许开发者访问引擎所使用的所有资源。拥有一个单一、统一且高度一致的界面来创建和管理所有类型的资源，无疑是一大优势。考虑到大多数其他游戏引擎中的资源数据分散在无数不一致且通常难以理解的工具中，这一点尤为重要。能够在 UnrealEd 中轻松找到任何资源本身就是一大优势。

虚幻引擎比许多其他引擎更不容易出错，因为资源必须明确导入虚幻引擎的资源数据库。这使得在制作过程的早期就能检查资源的有效性。在大多数游戏引擎中，任何旧数据都会被放入资源数据库，只有在最终构建时才能知道这些数据是否有效——有时甚至直到实际在运行时加载到游戏中才能知道。但使用虚幻引擎，资源在导入虚幻编辑器后即可进行验证。这意味着创建资源的人可以立即收到反馈，了解其资源是否配置正确。

当然，虚幻引擎的方法也存在一些严重的缺陷。首先，

Figure 7.1. UnrealEd's Generic Browser.

Another big benefit of UnrealEd is what I would call *one-stop shopping*. UnrealEd's Generic Browser (depicted in Figure 7.1) allows a developer to access literally every resource that is consumed by the engine. Having a single, unified and reasonably consistent interface for creating and managing all types of resources is a big win. This is especially true considering that the resource data in most other game engines is fragmented across countless inconsistent and often cryptic tools. Just being able to *find* any resource easily in UnrealEd is a big plus.

Unreal can be less error-prone than many other engines, because assets must be explicitly imported into Unreal's resource database. This allows resources to be checked for validity very early in the production process. In most game engines, any old data can be thrown into the resource database, and you only know whether or not that data is valid when it is eventually built—or sometimes not until it is actually loaded into the game at runtime. But with Unreal, assets can be validated as soon as they are imported into UnrealEd. This means that the person who created the asset gets immediate feedback as to whether his or her asset is configured properly.

Of course, Unreal's approach has some serious drawbacks. For one thing,

所有资源数据都存储在少量大型包文件中。这些文件是二进制文件，因此很难通过 CVS、Subversion 或 Perforce 等版本控制包合并。当多个用户想要修改同一个包中的资源时，这会带来一些严重的问题。即使用户尝试修改不同的资源，一次也只有一个用户可以锁定该包，因此另一个用户必须等待。可以通过将资源划分为相对较小的、粒度更细的包来降低这个问题的严重性，但实际上无法完全消除。

UnrealEd 中的引用完整性相当不错，但仍然存在一些问题。当资源被重命名或移动时，所有对该资源的引用都会自动使用一个虚拟对象来维护，该对象会将旧资源重新映射到新的名称/位置。这些虚拟重映射对象的问题在于它们会一直存在并累积，有时会导致问题，尤其是在删除资源时。总的来说，虚幻的引用完整性相当不错，但并非完美无缺。

尽管存在一些问题，UnrealEd 仍然是我迄今为止使用过的最用户友好、集成度最高、最精简的资产创建工具包、资源数据库和资产调节管道。

顽皮狗的引擎

对于《神秘海域：德雷克的宝藏》，顽皮狗将其资源元数据存储在 MySQL 数据库中。他们编写了一个自定义图形用户界面来管理数据库内容。该工具允许美术师、游戏设计师和程序员创建新资源、删除现有资源以及检查和修改资源。这个 GUI 是系统的关键组件，因为它让用户无需学习通过 SQL 与关系数据库交互的复杂知识。

《神秘海域》最初使用的 MySQL 数据库既没有提供有用的数据库更改历史记录，也没有提供回滚“错误”更改的有效方法。它也不支持多个用户编辑同一资源，管理起来也很困难。顽皮狗后来放弃了 MySQL，转而使用基于 XML 文件的资源数据库，并由 Perforce 进行管理。

Builder 是顽皮狗的资源数据库 GUI，如图 7.2 所示。
该窗口分为两个主要部分：左侧是树状视图，显示游戏中的所有资源；右侧是属性窗口，用于查看和编辑树状视图中选定的资源。资源树包含用于组织资源的文件夹，以便美术师和游戏设计师可以按照自己认为合适的方式组织资源。可以在任何文件夹中创建和管理各种类型的资源，包括：

all resource data is stored in a small number of large package files. These files are binary, so they are not easily merged by a revision control package like CVS, Subversion or Perforce. This presents some major problems when more than one user wants to modify resources that reside in a single package. Even if the users are trying to modify *different* resources, only one user can lock the package at a time, so the other has to wait. The severity of this problem can be reduced by dividing resources into relatively small, granular packages, but it cannot practically be eliminated.

Referential integrity is quite good in UnrealEd, but there are still some problems. When a resource is renamed or moved around, all references to it are maintained automatically using a dummy object that remaps the old resource to its new name/location. The problem with these dummy remapping objects is that they hang around and accumulate and sometimes cause problems, especially if a resource is deleted. Overall, Unreal's referential integrity is quite good, but it is not perfect.

Despite its problems, UnrealEd is by far the most user-friendly, well-integrated and streamlined asset creation toolkit, resource database and asset conditioning pipeline that I have ever worked with.

Naughty Dog's Engine

For *Uncharted: Drake's Fortune*, Naughty Dog stored its resource metadata in a MySQL database. A custom graphical user interface was written to manage the contents of the database. This tool allowed artists, game designers and programmers alike to create new resources, delete existing resources and inspect and modify resources as well. This GUI was a crucial component of the system, because it allowed users to avoid having to learn the intricacies of interacting with a relational database via SQL.

The original MySQL database used on *Uncharted* did not provide a useful history of the changes made to the database, nor did it provide a good way to roll back "bad" changes. It also did not support multiple users editing the same resource, and it was difficult to administer. Naughty Dog has since moved away from MySQL in favor of an XML file-based asset database, managed under Perforce.

Builder, Naughty Dog's resource database GUI, is depicted in Figure 7.2. The window is broken into two main sections: a tree view showing all resources in the game on the left and a properties window on the right, allowing the resource(s) that are selected in the tree view to be viewed and edited. The resource tree contains folders for organizational purposes, so that the artists and game designers can organize their resources in any way they see fit. Various types of resources can be created and managed within any folder, includ-

图 7.2。顽皮狗离线资源数据库 Builder 的前端 GUI。

Actor、关卡以及构成它们的各种子资源（主要是网格、骨架和动画）。动画也可以分组到称为“捆绑包”的伪文件夹中。这样可以创建大量动画，并将其作为一个单元进行管理，从而避免在树状视图中拖拽单个动画而浪费大量时间。

《神秘海域》和《最后生还者》系列中使用的资源调节管线由一组从命令行运行的资源导出器、编译器和链接器组成。该引擎能够处理各种不同类型的数据对象，但这些数据对象被打包成两种类型的资源文件之一：Actor 和关卡。Actor 可以包含骨架、网格、材质、纹理和/或动画。关卡包含静态背景网格、材质和纹理，以及关卡布局信息。

Figure 7.2. The front-end GUI for Naughty Dog's offline resource database, Builder.

ing actors and levels, and the various subresources that comprise them (primarily meshes, skeletons and animations). Animations can also be grouped into pseudo-folders known as bundles. This allows large groups of animations to be created and then managed as a unit, and prevents a lot of wasted time dragging individual animations around in the tree view.

The asset conditioning pipeline employed on the *Uncharted* and *The Last of Us* series consists of a set of resource exporters, compilers and linkers that are run from the command line. The engine is capable of dealing with a wide variety of different kinds of data objects, but these are packaged into one of two types of resource files: actors and levels. An actor can contain skeletons, meshes, materials, textures and/or animations. A level contains static background meshes, materials and textures, and also level-layout information. To

要构建一个 Actor，只需在命令行中输入 ba name-of-actor 即可；要构建一个关卡，只需输入 bl name-of-level 即可。这些命令行工具会查询数据库，以确定如何构建相关的 Actor 或关卡。这包括如何从 Maya 和 Photoshop 等 DCC 工具导出资源、如何处理数据以及如何将其打包成游戏引擎可加载的二进制 .pak 文件。这比许多引擎要简单得多，因为许多引擎都需要美术师手动导出资源——这是一项耗时、繁琐且容易出错的任务。

顽皮狗使用的资源管道设计的好处包括：

- 精细资源。资源可以通过游戏中的逻辑实体（网格、材质、骨架和动画）进行操作。这些资源类型足够精细，团队几乎不会遇到两个用户同时编辑同一资源的冲突。

- 必要的功能（仅此而已）。Builder 工具提供了一组强大的功能，可以满足团队的需求，但顽皮狗并没有浪费任何资源来创建他们不需要的功能。

- 明显映射到源文件。用户可以非常快速地确定哪些源资产（原生 DCC 文件，如 Maya .ma 文件或 photoshop .psd 文件）构成了特定资源。

- 轻松更改 DCC 数据的导出和处理方式。只需单击相关资源，然后在资源数据库 GUI 中调整其处理属性即可。

- 轻松构建资源。只需在命令行中输入 ba 或 bl，然后输入资源名称即可。依赖关系系统会处理剩下的事情。

当然，顽皮狗的工具链也有一些缺点，包括：

- 缺乏可视化工具。预览资产的唯一方法是将其加载到游戏或模型/动画查看器中（这实际上只是游戏本身的一种特殊模式）。

- 工具尚未完全集成。顽皮狗使用一个工具来布局关卡，另一个工具来管理资源数据库中的大部分资源，第三个工具来设置材质和着色器（这不属于资源数据库前端）。资源的构建是通过命令行完成的。如果将所有这些功能集成到一个工具中，可能会更方便一些。然而，顽皮狗目前没有计划这样做，因为这样做的收益可能无法抵消成本。

build an actor, one simply types `ba` *name-of-actor* on the command line; to build a level, one types `bl` *name-of-level*. These command-line tools query the database to determine exactly *how* to build the actor or level in question. This includes information on how to export the assets from DCC tools like Maya and Photoshop, how to process the data, and how to package it into binary `.pak` files that can be loaded by the game engine. This is much simpler than in many engines, where resources have to be *exported manually* by the artists—a time-consuming, tedious and error-prone task.

The benefits of the resource pipeline design used by Naughty Dog include:

- *Granular resources*. Resources can be manipulated in terms of logical entities in the game—meshes, materials, skeletons and animations. These resource types are granular enough that the team almost never has conflicts in which two users want to edit the same resource simultaneously.
- *The necessary features (and no more)*. The Builder tool provides a powerful set of features that meet the needs of the team, but Naughty Dog didn't waste any resources creating features they didn't need.
- *Obvious mapping to source files*. A user can very quickly determine which source assets (native DCC files, like Maya .ma files or photoshop .psd files) make up a particular resource.
- *Easy to change how DCC data is exported and processed*. Just click on the resource in question and twiddle its processing properties within the resource database GUI.
- *Easy to build assets*. Just type `ba` or `bl` followed by the resource name on the command line. The dependency system takes care of the rest.

Of course, Naughty Dog's tool chain does have some drawbacks as well, including:

- *Lack of visualization tools*. The only way to preview an asset is to load it into the game or the model/animation viewer (which is really just a special mode of the game itself).
- *The tools aren't fully integrated*. Naughty Dog uses one tool to lay out levels, another to manage the majority of resources in the resource database, and a third to set up materials and shaders (this is not part of the resource database front end). Building the assets is done on the command line. It might be a bit more convenient if all of these functions were to be integrated into a single tool. However, Naughty Dog has no plans to do this, because the benefit would probably not outweigh the costs involved.

OGRE的资源管理系统

OGRE 是一个渲染引擎，而非一个功能齐全的游戏引擎。即便如此，OGRE 确实拥有一个相当完善且设计精良的运行时资源管理器。它使用简单一致的接口来加载几乎任何类型的资源。而且，该系统在设计时充分考虑了可扩展性。任何程序员都可以轻松地为一种全新的资源实现资源管理器，并将其轻松集成到 OGRE 的资源框架中。

OGRE 资源管理器的缺点之一是它只是一个运行时解决方案。OGRE 缺乏任何类型的离线资源数据库。OGRE 确实提供了一些导出器，能够将 Maya 文件转换为 OGRE 可以使用的网格（包含材质、着色器、骨架和可选动画）。但是，导出器必须在 Maya 内部手动运行。更糟糕的是，所有描述特定 Maya 文件导出和处理的元数据都必须由执行导出的用户输入。

总而言之，OGRE 的运行时资源管理器功能强大且设计精良。但如果工具方面能够拥有一个同样强大且现代化的资源数据库和资产调节管道，OGRE 将受益匪浅。

微软的XNA

XNA 是微软推出的一款游戏开发工具包，主要面向 PC 和 Xbox 360 平台。尽管微软已于 2014 年停止提供 XNA，但它仍然是学习游戏引擎的优秀资源。XNA 的资源管理系统非常独特，它利用 Visual Studio IDE 的项目管理和构建系统来管理和构建游戏中的资源。XNA 的游戏开发工具 Game Studio Express 只是 Visual Studio Express 的一个插件。

7.2.1.4 资产调节管道

在 1.7 节中，我们了解到资源数据通常使用 Maya、ZBrush、Photoshop 或 Houdini 等高级数字内容创作 (DCC) 工具创建。然而，这些工具使用的数据格式通常不适合游戏引擎直接使用。因此，大多数资源数据在传输到游戏引擎的过程中会经过资产调节管道 (ACP)。ACP 有时也称为资源调节管道 (RCP)，或简称为工具链。

每个资源管道都始于原生 DCC 格式的源资产集合（例如 Maya .ma 或 .mb 文件、Photoshop .psd 文件等）。这些

OGRE's Resource Manager System

OGRE is a rendering engine, not a full-fledged game engine. That said, OGRE does boast a reasonably complete and very well-designed runtime resource manager. A simple, consistent interface is used to load virtually any kind of resource. And the system has been designed with extensibility in mind. Any programmer can quite easily implement a resource manager for a brand new kind of asset and integrate it easily into OGRE's resource framework.

One of the drawbacks of OGRE's resource manager is that it is a runtime-only solution. OGRE lacks any kind of offline resource database. OGRE does provide some exporters that are capable of converting a Maya file into a mesh that can be used by OGRE (complete with materials, shaders, a skeleton and optional animations). However, the exporter must be run manually from within Maya itself. Worse, all of the metadata describing how a particular Maya file should be exported and processed must be entered by the user doing the export.

In summary, OGRE's runtime resource manager is powerful and well-designed. But, OGRE would benefit a great deal from an equally powerful and modern resource database and asset conditioning pipeline on the tools side.

Microsoft's XNA

XNA is a game development toolkit by Microsoft, targeted at the PC and Xbox 360 platforms. Although it was retired by Microsoft in 2014, it's still a good resource for learning about game engines. XNA's resource management system is unique, in that it leverages the project management and build systems of the Visual Studio IDE to manage and build the assets in the game as well. XNA's game development tool, Game Studio Express, is just a plug-in to Visual Studio Express.

7.2.1.4 The Asset Conditioning Pipeline

In Section 1.7, we learned that resource data is typically created using advanced digital content creation (DCC) tools like Maya, ZBrush, Photoshop or Houdini. However, the data formats used by these tools are usually not suitable for direct consumption by a game engine. So the majority of resource data is passed through an *asset conditioning pipeline* (ACP) on its way to the game engine. The ACP is sometimes referred to as the *resource conditioning pipeline* (RCP), or simply the *tool chain*.

Every resource pipeline starts with a collection of *source assets* in native DCC formats (e.g., Maya .ma or .mb files, Photoshop .psd files, etc.). These

资产在进入游戏引擎的过程中通常要经过三个处理阶段：

1. 导出器。我们需要某种方法将数据从 DCC 的原生格式导出为我们可以操作的格式。这通常是通过为相应的 DCC 编写自定义插件来实现的。插件的作用是将数据导出为某种中间文件格式，以便传递到流程的后续阶段。大多数 DCC 应用程序都提供了相当便捷的机制来实现这一点。Maya 实际上提供了三种机制：C++ SDK、名为 MEL 的脚本语言，以及最近推出的 Python 接口。

   如果 DCC 应用程序不提供自定义钩子，我们始终可以将数据保存为 DCC 工具的原生格式之一。如果幸运的话，这些格式可能是开放格式、比较直观的文本格式或其他我们可以进行逆向工程的格式。假设情况如此，我们可以将文件直接传递到流水线的下一阶段。

2. 资源编译器。我们经常需要以各种方式“处理”从 DCC 应用程序导出的原始数据，以使其可用于游戏。例如，我们可能需要将网格的三角形重新排列成条带，或者压缩纹理位图，或者计算 Catmull-Rom 样条曲线各段的弧长。并非所有类型的资源都需要编译——有些资源导出后可能立即可用于游戏。

3. 资源链接器。有时，需要将多个资源文件合并成一个可用的包，然后才能由游戏引擎加载。这类似于将已编译的 C++ 程序的目标文件链接成可执行文件的过程，因此此过程有时也称为资源链接。例如，在构建 3D 模型等复杂的复合资源时□□，我们可能需要将来自多个导出的网格文件、多个材质文件、一个骨架文件和多个动画文件的数据合并成一个资源。并非所有类型的资源都需要链接——某些资源在导出或编译步骤后即可用于游戏。

资源依赖和构建规则

与编译 C 或 C++ 项目中的源文件然后将其链接到可执行文件中非常相似，资产调节管道处理源资产（以 Maya 几何和动画文件、Photoshop PSD 文件的形式，

assets are typically passed through three processing stages on their way to the game engine:

1. *Exporters*. We need some way of getting the data out of the DCC's native format and into a format that we can manipulate. This is usually accomplished by writing a custom plug-in for the DCC in question. It is the plug-in's job to export the data into some kind of intermediate file format that can be passed to later stages in the pipeline. Most DCC applications provide a reasonably convenient mechanism for doing this. Maya actually provides three: a C++ SDK, a scripting language called MEL and most recently a Python interface as well.

   In cases where a DCC application provides no customization hooks, we can always save the data in one of the DCC tool's native formats. With any luck, one of these will be an open format, a reasonably intuitive text format, or some other format that we can reverse engineer. Presuming this is the case, we can pass the file directly to the next stage of the pipeline.

2. *Resource compilers*. We often have to "massage" the raw data exported from a DCC application in various ways in order to make them game-ready. For example, we might need to rearrange a mesh's triangles into strips, or compress a texture bitmap, or calculate the arc lengths of the segments of a Catmull-Rom spline. Not all types of resources need to be compiled—some might be game-ready immediately upon being exported.

3. *Resource linkers*. Multiple resource files sometimes need to be combined into a single useful package prior to being loaded by the game engine. This mimics the process of linking together the object files of a compiled C++ program into an executable file, and so this process is sometimes called *resource linking*. For example, when building a complex composite resource like a 3D model, we might need to combine the data from multiple exported mesh files, multiple material files, a skeleton file and multiple animation files into a single resource. Not all types of resources need to be linked—some assets are game-ready after the export or compile steps.

Resource Dependencies and Build Rules

Much like compiling the source files in a C or C++ project and then linking them into an executable, the asset conditioning pipeline processes source assets (in the form of Maya geometry and animation files, Photoshop PSD files,

引擎会将原始音频片段、文本文件等转换为游戏可用格式，然后将它们链接在一起形成一个完整的整体，供引擎使用。就像计算机程序中的源文件一样，游戏资源通常具有相互依赖关系。（例如，一个网格引用一种或多种材质，而这些材质又引用各种纹理。）这些相互依赖关系通常会影响管道处理资源的顺序。（例如，我们可能需要先构建角色的骨架，然后才能处理该角色的任何动画。）此外，资源之间的依赖关系会告诉我们，当某个源资源发生变化时，哪些资源需要重建。

构建依赖关系不仅围绕资源本身的更改，还围绕数据格式的更改。例如，如果用于存储三角形网格的文件格式发生变化，则可能需要重新导出和/或重建整个游戏中的所有网格。某些游戏引擎采用的数据格式对版本变化具有很强的鲁棒性。例如，资源可能包含版本号，并且游戏引擎可能包含“知道”如何加载和使用旧资源的代码。这种策略的缺点是资源文件和引擎代码往往会变得庞大。当数据格式更改相对较少时，最好是硬着头皮在格式更改发生时重新处理所有文件。

每个资产调节管道都需要一组描述资产之间相互依赖关系的规则，以及某种可以使用这些信息来确保在修改源资产时以正确的顺序构建正确的资产的构建工具。一些游戏团队推出自己的构建系统。其他人使用成熟的工具，例如 `make`。无论选择哪种解决方案，团队都应该非常小心地对待他们的构建依赖关系系统。如果不这样做，对源资产的更改可能不会触发重建正确的资产。结果可能是游戏资产不一致，从而导致视觉异常甚至引擎崩溃。以我的个人经验来看，我目睹了无数的时间被浪费在追踪问题上，如果正确指定了资产相互依赖关系并实施了可靠地使用它们的构建系统，这些问题是可以避免的。

7.2.2 运行时资源管理

现在让我们将注意力转向资源数据库中的资产在运行时如何在引擎内加载、管理和卸载。

7.2.2.1 运行时资源管理器的职责

游戏引擎的运行时资源管理器承担着广泛的职责，所有职责都与将资源加载到内存的主要任务有关：

raw audio clips, text files, etc.), converts them into game-ready form and then links them together into a cohesive whole for use by the engine. And just like the source files in a computer program, game assets often have interdependencies. (For example, a mesh refers to one or more materials, which in turn refer to various textures.) These interdependencies typically have an impact on the order in which assets must be processed by the pipeline. (For example, we might need to build a character's skeleton before we can process any of that character's animations.) In addition, the dependencies between assets tell us which assets need to be rebuilt when a particular source asset changes.

Build dependencies revolve not only around changes to the assets themselves, but also around changes to data formats. If the format of the files used to store triangle meshes changes, for instance, all meshes in the entire game may need to be reexported and/or rebuilt. Some game engines employ data formats that are robust to version changes. For example, an asset may contain a version number, and the game engine may include code that "knows" how to load and make use of legacy assets. The downside of such a policy is that asset files and engine code tend to become bulky. When data format changes are relatively rare, it may be better to just bite the bullet and reprocess all the files when format changes do occur.

Every asset conditioning pipeline requires a set of rules that describe the interdependencies between the assets, and some kind of build tool that can use this information to ensure that the proper assets are built, in the proper order, when a source asset is modified. Some game teams roll their own build system. Others use an established tool, such as `make`. Whatever solution is selected, teams should treat their build dependency system with utmost care. If you don't, changes to sources, assets may not trigger the proper assets to be rebuilt. The result can be inconsistent game assets, which may lead to visual anomalies or even engine crashes. In my personal experience, I've witnessed countless hours wasted in tracking down problems that could have been avoided had the asset interdependencies been properly specified and the build system implemented to use them reliably.

7.2.2 Runtime Resource Management

Let us turn our attention now to how the assets in our resource database are loaded, managed and unloaded within the engine at runtime.

7.2.2.1 Responsibilities of the Runtime Resource Manager

A game engine's runtime resource manager takes on a wide range of responsibilities, all related to its primary mandate of loading resources into memory:

• 确保在任何给定时间内存中只存在每个唯一资源的一个副本。

• 管理每个资源的生命周期。

• 加载所需的资源并卸载不再需要的资源。

• 处理复合资源的加载。复合资源是由其他资源组成的资源。例如，3D 模型是一个复合资源，它由一个网格、一种或多种材质、一个或多个纹理以及可选的骨架和多个骨骼动画组成。

• 维护引用完整性。这包括内部引用完整性（单个资源内的交叉引用）和外部引用完整性（资源之间的交叉引用）。例如，模型引用其网格和骨架；网格引用其材质，而材质又引用纹理资源；动画引用骨架，最终将动画与一个或多个模型关联起来。加载复合资源时□□，资源管理器必须确保所有必要的子资源都已加载，并且必须正确修补所有交叉引用。

• 管理已加载资源的内存使用情况，并确保资源存储在内存中的适当位置。

• 允许在资源加载后根据资源类型执行自定义处理。此过程有时称为登录或加载初始化资源。

• 通常（但并非总是）提供单一统一的接口，用于管理各种类型的资源。理想情况下，资源管理器也应易于扩展，以便能够根据游戏开发团队的需求处理新类型的资源。

• 如果引擎支持此功能，则处理流（即异步资源加载）。

7.2.2.2 资源文件和目录组织

在某些游戏引擎（通常是 PC 引擎）中，每个资源都存放在磁盘上一个单独的“松散”文件中进行管理。这些文件通常包含在一个目录树中，其内部组织结构的设计主要是为了方便资源创建者；引擎通常不关心资源文件在资源树中的位置。以下是假设一款名为《太空逃亡者》的游戏的典型资源目录树：

- Ensures that only *one copy* of each unique resource exists in memory at any given time.
- Manages the *lifetime* of each resource.
- *Loads* needed resources and *unloads* resources that are no longer needed.
- Handles loading of *composite resources*. A composite resource is a resource comprised of other resources. For example, a *3D model* is a composite resource that consists of a mesh, one or more materials, one or more textures and optionally a skeleton and multiple skeletal animations.
- Maintains *referential integrity*. This includes *internal* referential integrity (cross-references within a single resource) and *external* referential integrity (cross-references between resources). For example, a model refers to its mesh and skeleton; a mesh refers to its materials, which in turn refer to texture resources; animations refer to a skeleton, which ultimately ties them to one or more models. When loading a composite resource, the resource manager must ensure that all necessary subresources are loaded, and it must patch in all of the cross-references properly.
- Manages the *memory usage* of loaded resources and ensures that resources are stored in the appropriate place(s) in memory.
- Permits *custom processing* to be performed on a resource after it has been loaded, on a per-resource-type basis. This process is sometimes known as *logging in* or *load-initializing* the resource.
- Usually (but not always) provides a single *unified interface* through which a wide variety of resource types can be managed. Ideally a resource manager is also easily extensible, so that it can handle new types of resources as they are needed by the game development team.
- Handles *streaming* (i.e., asynchronous resource loading), if the engine supports this feature.

7.2.2.2 Resource File and Directory Organization

In some game engines (typically PC engines), each individual resource is managed in a separate “loose” file on-disk. These files are typically contained within a tree of directories whose internal organization is designed primarily for the convenience of the people creating the assets; the engine typically doesn’t care where resource files are located within the resource tree. Here’s a typical resource directory tree for a hypothetical game called *Space Evaders:*

```
SpaceEvaders    整个游戏的根目录。
  Resources     所有资源的根源。
    NPC         非玩家角色模型和动画。
      海盗模型和海盗动画。海军陆战队员模型和海军
      陆战队员动画。
      ...

    Player      玩家角色模型和动画。

    Weapons     武器的模型和动画。
      手枪模型和手枪动画。
      Rifle     步枪的模型和动画。
      BFG       大...呃...枪的模型和动画。
      ...

    Levels      背景几何和级别布局。
      Level1 第一级的资源。
      Level2 第二级的资源。
      ...

    Objects     各种 3D 对象。
      Crate     随处可见的易碎板条箱。
      桶 随处可见的爆炸桶。
```

其他引擎将多个资源打包到一个文件中，例如 ZIP 压缩包或其他复合文件（可能是专有格式）。这种方法的主要好处是缩短了加载时间。从文件加载数据时，最大的三个成本是寻道时间（即将读取头移动到物理介质上的正确位置）、打开每个文件所需的时间以及将数据从文件读入内存的时间。其中，寻道时间和文件打开时间在许多操作系统上可能非常关键。使用单个大文件时，所有这些成本都会最小化。单个文件可以在磁盘上按顺序组织，从而将寻道时间降至最低。而且由于只需打开一个文件，因此无需再打开单个资源文件。

固态硬盘 (SSD) 不会像 DVD、蓝光光盘和硬盘 (HDD) 等旋转介质那样存在寻道时间问题。然而，迄今为止，还没有哪款游戏机将固态硬盘作为主要固定存储设备（即使是 PS4 和 Xbox One 也不例外）。因此，在未来一段时间内，设计游戏的 I/O 模式以最大限度地缩短寻道时间可能会成为一种必需。

OGRE 渲染引擎的资源管理器允许资源存在


```
SpaceEvaders    Root directory for entire game.
  Resources     Root of all resources.
    NPC         Non-player character models and animations.
      Pirate    Models and animations for pirates.
      Marine    Models and animations for marines.
      ...

    Player      Player character models and animations.

    Weapons     Models and animations for weapons.
      Pistol    Models and animations for the pistol.
      Rifle     Models and animations for the rifle.
      BFG       Models and animations for the big…uh…gun.
      ...

    Levels      Background geometry and level layouts.
      Level1    First level's resources.
      Level2    Second level's resources.
      ...

    Objects     Miscellaneous 3D objects.
      Crate     The ubiquitous breakable crate.
      Barrel    The ubiquitous exploding barrel.
```

Other engines package multiple resources together in a single file, such as a ZIP archive, or some other composite file (perhaps of a proprietary format). The primary benefit of this approach is improved load times. When loading data from files, the three biggest costs are *seek times* (i.e., moving the read head to the correct place on the physical media), the time required to open each individual file, and the time to read the data from the file into memory. Of these, the seek times and file-open times can be nontrivial on many operating systems. When a single large file is used, all of these costs are minimized. A single file can be organized sequentially on the disk, reducing seek times to a minimum. And with only one file to open, the cost of opening individual resource files is eliminated.

Solid-state drives (SSD) do not suffer from the seek time problems that plague spinning media like DVDs, Blu-ray discs and hard disc drives (HDD). However, no game console to date includes a solid-state drive as the primary fixed storage device (not even the PS4 and Xbox One). So designing your game's I/O patterns in order to minimize seek times is likely to be a necessity for some time to come.

The OGRE rendering engine's resource manager permits resources to exist

可以将其作为磁盘上的松散文件，或作为大型 ZIP 压缩包中的虚拟文件。ZIP 格式的主要优点如下：

1. 它是一种开放格式。用于读写 ZIP 压缩包的 zlib 和 zziplib 库均可免费使用。zlib SDK 完全免费（参见 http://www.zlib.net），而 zziplib SDK 遵循宽 Gnu 公共许可证 (LGPL)（参见 http://zziplib.sourceforge.net）。

2. ZIP 压缩包中的虚拟文件会“记住”它们的相对路径。这意味着 ZIP 压缩包在大多数情况下“看起来像”一个原始文件系统。OGRE 资源管理器通过看似文件系统路径的字符串唯一地标识所有资源。然而，这些路径有时标识的是 ZIP 压缩包中的虚拟文件，而不是磁盘上的松散文件，游戏程序员在大多数情况下无需注意这种差异。

3. ZIP 压缩包可以进行压缩。这可以减少资源占用的磁盘空间。更重要的是，它还能加快加载速度，因为需要从硬盘加载到内存的数据更少。这在从 DVD-ROM 或蓝光光盘读取数据时尤其有用，因为这些设备的数据传输速率比硬盘驱动器慢得多。因此，数据加载到内存后解压的成本通常会被从设备加载较少数据所节省的时间所抵消。

4. ZIP 压缩包是模块化的。资源可以组合成一个 ZIP 文件并作为一个单元进行管理。这一理念的一个特别巧妙的应用是产品本地化。所有需要本地化的资源（例如包含对话的音频片段以及包含特定区域文字或符号的纹理）都可以放在一个 ZIP 文件中，然后可以生成该 ZIP 文件的不同版本，每个语言或区域对应一个版本。要让游戏在特定区域运行，引擎只需加载相应版本的 ZIP 压缩包即可。

虚幻引擎采用了类似的方法，但也存在一些重要区别。在虚幻引擎中，所有资源都必须包含在称为“包”（又称“pak文件”）的大型复合文件中。不允许使用松散的磁盘文件。包文件的格式是专有的。虚幻引擎的游戏编辑器UnrealEd允许开发者创建和管理包及其包含的资源。

as loose files on disk, or as virtual files within a large ZIP archive. The primary benefits of the ZIP format are the following:

1. *It is an open format*. The `zlib` and `zziplib` libraries used to read and write ZIP archives are freely available. The zlib SDK is totally free (see http://www.zlib.net), while the zziplib SDK falls under the Lesser Gnu Public License (LGPL) (see http://zziplib.sourceforge.net).

2. *The virtual files within a ZIP archive "remember" their relative paths*. This means that a ZIP archive "looks like" a raw file system for most intents and purposes. The OGRE resource manager identifies all resources uniquely via strings that appear to be file system paths. However, these paths sometimes identify virtual files within a ZIP archive instead of loose files on disk, and a game programmer needn't be aware of the difference in most situations.

3. *ZIP archives may be compressed*. This reduces the amount of disk space occupied by resources. But, more importantly, it again speeds up load times, as less data need be loaded into memory from the fixed disk. This is especially helpful when reading data from a DVD-ROM or Blu-ray disk, as the data transfer rates of these devices are much slower than a hard disk drive. Hence the cost of decompressing the data after it has been loaded into memory is often more than offset by the time saved in loading less data from the device.

4. *ZIP archives are modular*. Resources can be grouped together into a ZIP file and managed as a unit. One particularly elegant application of this idea is in product localization. All of the assets that need to be localized (such as audio clips containing dialogue and textures that contain words or region-specific symbols) can be placed in a single ZIP file, and then different versions of this ZIP file can be generated, one for each language or region. To run the game for a particular region, the engine simply loads the corresponding version of the ZIP archive.

The Unreal Engine takes a similar approach, with a few important differences. In Unreal, all resources must be contained within large composite files known as *packages* (a.k.a. "pak files"). No loose disk files are permitted. The format of a package file is proprietary. The Unreal Engine's game editor, UnrealEd, allows developers to create and manage packages and the resources they contain.

7.2.2.3 资源文件格式

每种类型的资源文件都可能具有不同的格式。例如，网格文件的存储格式始终与纹理位图的存储格式不同。某些类型的资源则以标准化的开放格式存储。例如，纹理通常存储为 Targa 文件 (TGA)、便携式网络图形文件 (PNG)、标记图像文件格式 (TIFF)、联合图像专家组文件 (JPEG) 或 Windows 位图文件 (BMP)，或者以标准化压缩格式（例如 DirectX 的 S3 纹理压缩系列格式（S3TC，也称为 DXT n 或 DXTC））存储。同样，3D 网格数据通常从 Maya 或 Lightwave 等建模工具导出为标准化格式（例如 OBJ 或 COLLADA），以供游戏引擎使用。

有时，一种文件格式可以存储多种不同类型的资源。例如，Rad Game Tools 的 Granny SDK（http://www.radgametools.com）实现了一种灵活的开放文件格式，可用于存储 3D 网格数据、骨骼层次结构和骨骼动画数据。（事实上，Granny 文件格式可以轻松重新用于存储几乎任何类型的数据。）

许多游戏引擎程序员出于各种原因会自行设计文件格式。如果没有标准化格式能够提供引擎所需的所有信息，那么这可能是必要的。此外，许多游戏引擎会尽可能多地进行离线处理，以最大程度地减少运行时加载和处理资源数据所需的时间。例如，如果数据需要符合内存中的特定布局，则可能会选择原始二进制格式，以便离线工具可以对数据进行布局（而不是在资源加载后尝试在运行时对其进行格式化）。

7.2.2.4 资源 GUID

游戏中的每个资源都必须具有某种全局唯一标识符 (GUID)。最常见的 GUID 是资源的文件系统路径（以字符串或 32 位哈希值存储）。这种 GUID 非常直观，因为它将每个资源清晰地映射到磁盘上的物理文件。而且，由于操作系统已经保证了不会有两个文件具有相同的路径，因此它在整个游戏中保证是唯一的。

然而，文件系统路径绝不是资源 GUID 的唯一选择。有些引擎使用不太直观的 GUID 类型，例如 128 位哈希码，可能由保证唯一性的工具分配。在其他引擎中，使用文件系统路径作为资源标识符是不可行的。例如，虚幻引擎将许多资源存储在一个称为包的大型文件中，因此包文件的路径无法唯一地标识任何一个资源。为了解决这个问题，虚幻包文件被组织到一个文件夹中

7.2.2.3 Resource File Formats

Each type of resource file potentially has a different format. For example, a mesh file is always stored in a different format than that of a texture bitmap. Some kinds of assets are stored in standardized, open formats. For example, textures are typically stored as Targa files (TGA), Portable Network Graphics files (PNG), Tagged Image File Format files (TIFF), Joint Photographic Experts Group files (JPEG) or Windows Bitmap files (BMP)—or in a standardized compressed format such as DirectX's S3 Texture Compression family of formats (S3TC, also known as DXT*n* or DXTC). Likewise, 3D mesh data is often exported out of a modeling tool like Maya or Lightwave into a standardized format such as OBJ or COLLADA for consumption by the game engine.

Sometimes a single file format can be used to house many different types of assets. For example, the *Granny* SDK by Rad Game Tools (http://www.radgametools.com) implements a flexible open file format that can be used to store 3D mesh data, skeletal hierarchies and skeletal animation data. (In fact the Granny file format can be easily repurposed to store virtually any kind of data imaginable.)

Many game engine programmers roll their own file formats for various reasons. This might be necessary if no standardized format provides all of the information needed by the engine. Also, many game engines endeavor to do as much offline processing as possible in order to minimize the amount of time needed to load and process resource data at runtime. If the data needs to conform to a particular layout in memory, for example, a raw binary format might be chosen so that the data can be laid out by an offline tool (rather than attempting to format it at runtime after the resource has been loaded).

7.2.2.4 Resource GUIDs

Every resource in a game must have some kind of *globally unique identifier* (GUID). The most common choice of GUID is the resource's file system path (stored either as a string or a 32-bit hash). This kind of GUID is intuitive, because it clearly maps each resource to a physical file on-disk. And it's guaranteed to be unique across the entire game, because the operating system already guarantees that no two files will have the same path.

However, a file system path is by no means the only choice for a resource GUID. Some engines use a less-intuitive type of GUID, such as a 128-bit hash code, perhaps assigned by a tool that guarantees uniqueness. In other engines, using a file system path as a resource identifier is infeasible. For example, Unreal Engine stores many resources in a single large file known as a *package*, so the path to the package file does not uniquely identify any one resource. To overcome this problem, an Unreal package file is organized into a folder

包含单个资源的层次结构。虚幻引擎会为包中的每个单个资源赋予一个唯一的名称，该名称看起来很像文件系统路径。因此，在虚幻引擎中，资源 GUID 是由包文件的（唯一）名称与相应资源的包内路径连接而成。例如，《战争机器》的资源 GUID Locust_Boomer.Physical Materials.LocustBoomerLeather 标识了 Locust_Boomer 包文件的 PhysicalMaterials 文件夹中名为 Locust BoomerLeather 的材质。

7.2.2.5 资源注册表

为了确保每个唯一资源在任意时刻都只有一个副本加载到内存中，大多数资源管理器都会维护某种已加载资源的注册表。最简单的实现是一个字典——即键值对的集合。键包含资源的唯一 ID，而值通常是指向内存中资源的指针。

每当资源加载到内存中时，都会将其对应的条目添加到资源注册表字典中，并使用其 GUID 作为键。每当资源卸载时，其注册表条目都会被删除。当游戏请求资源时，资源管理器会通过资源注册表中的 GUID 查找该资源。如果可以找到该资源，则直接返回指向该资源的指针。如果找不到该资源，则可以自动加载或返回失败代码。

乍一看，如果在资源注册表中找不到请求的资源，自动加载该资源似乎是最直观的做法。事实上，一些游戏引擎也这样做了。然而，这种方法存在一些严重的问题。加载资源是一项缓慢的操作，因为它涉及在磁盘上定位和打开文件，将可能大量的数据读入内存（从 DVD-ROM 驱动器等可能速度较慢的设备），并且还可能在资源数据加载后执行加载后初始化。如果请求是在游戏过程中发出的，加载资源所需的时间可能会导致游戏帧率出现非常明显的卡顿，甚至出现数秒的卡顿。因此，引擎倾向于采用以下两种替代方法之一：

1. 在游戏过程中，资源加载可能被完全禁止。在这种模式下，游戏关卡的所有资源都会在游戏开始前一次性加载，通常是在玩家观看加载画面或进度条时。

2. 资源加载可能是异步的（即数据可能是流式的）。在这个模型中，当玩家处于 X 关卡时，重新

hierarchy containing individual resources. Unreal gives each individual resource within a package a unique name, which looks much like a file system path. So in Unreal, a resource GUID is formed by concatenating the (unique) name of the package file with the in-package path of the resource in question. For example, the *Gears of War* resource GUID `Locust_Boomer.PhysicalMaterials.LocustBoomerLeather` identifies a material called `LocustBoomerLeather` within the `PhysicalMaterials` folder of the `Locust_Boomer` package file.

7.2.2.5 The Resource Registry

In order to ensure that only one copy of each unique resource is loaded into memory at any given time, most resource managers maintain some kind of *registry* of loaded resources. The simplest implementation is a *dictionary*—i.e., a collection of *key-value pairs*. The keys contain the unique ids of the resources, while the values are typically pointers to the resources in memory.

Whenever a resource is loaded into memory, an entry for it is added to the resource registry dictionary, using its GUID as the key. Whenever a resource is unloaded, its registry entry is removed. When a resource is requested by the game, the resource manager looks up the resource by its GUID within the resource registry. If the resource can be found, a pointer to it is simply returned. If the resource cannot be found, it can either be loaded automatically or a failure code can be returned.

At first blush, it might seem most intuitive to automatically load a requested resource if it cannot be found in the resource registry. And in fact, some game engines do this. However, there are some serious problems with this approach. Loading a resource is a slow operation, because it involves locating and opening a file on disk, reading a potentially large amount of data into memory (from a potentially slow device like a DVD-ROM drive), and also possibly performing post-load initialization of the resource data once it has been loaded. If the request comes during active gameplay, the time it takes to load the resource might cause a very noticeable hitch in the game's frame rate, or even a multi-second freeze. For this reason, engines tend to take one of two alternative approaches:

1. Resource loading might be disallowed completely during active gameplay. In this model, all of the resources for a game level are loaded *en masse* just prior to gameplay, usually while the player watches a loading screen or progress bar of some kind.

2. Resource loading might be done *asynchronously* (i.e., the data might be *streamed*). In this model, while the player is engaged in level X, the re-

关卡 Y 的资源正在后台加载。这种方法更可取，因为它为玩家提供了无加载画面的游戏体验。然而，实现起来相当困难。

7.2.2.6 资源生命周期

资源的生命周期定义为从首次加载到内存到其内存被回收用于其他用途之间的时间间隔。资源管理器的职责之一是管理资源生命周期——可以自动管理，也可以通过向游戏提供必要的 API 函数来手动管理。

每种资源都有其自己的生命周期要求：

• 某些资源必须在游戏首次启动时加载，并且必须在整个游戏过程中驻留在内存中。也就是说，它们的生命周期实际上是无限的。这些资源有时被称为全局资源或全局资产。典型示例包括玩家角色的网格、材质、纹理和核心动画、平视显示器上使用的纹理和字体，以及游戏中使用的所有标准武器的资源。任何在整个游戏过程中玩家可见或可听到的资源（并且无法在需要时动态加载）都应被视为全局资源。

• 其他资源的生命周期与特定游戏关卡的生命周期相匹配。这些资源必须在玩家首次进入该关卡时存在于内存中，并在玩家永久离开该关卡后被转储。

• 某些资源的生命周期可能短于其所在关卡的持续时间。例如，构成游戏内过场动画（推进故事情节或向玩家提供重要信息的迷你电影）的动画和音频片段可能会在玩家观看过场动画之前加载，并在过场动画播放结束后被丢弃。

• 某些资源（例如背景音乐、环境音效或全屏电影）会在播放时“实时”流式传输。此类资源的生命周期很难定义，因为每个字节在内存中仅保留极短的时间，但整首音乐听起来却会持续很长时间。此类资源通常以符合底层硬件要求的大小块加载。例如，一个音乐曲目可能以 4 KiB 的块读取，因为这可能是低级音响系统使用的缓冲区大小。只有两个

sources for level Y are being loaded in the background. This approach is preferable because it provides the player with a load-screen-free play experience. However, it is considerably more difficult to implement.

7.2.2.6 Resource Lifetime

The *lifetime* of a resource is defined as the time period between when it is first loaded into memory and when its memory is reclaimed for other purposes. One of the resource manager's jobs is to manage resource lifetimes—either automatically or by providing the necessary API functions to the game, so it can manage resource lifetimes manually.

Each resource has its own lifetime requirements:

- Some resources must be loaded when the game first starts up and must stay resident in memory for the entire duration of the game. That is, their lifetimes are effectively infinite. These are sometimes called *global resources* or *global assets*. Typical examples include the player character's mesh, materials, textures and core animations, textures and fonts used on the heads-up display, and the resources for all of the standard-issue weapons used throughout the game. Any resource that is visible or audible to the player throughout the entire game (and cannot be loaded on the fly when needed) should be treated as a global resource.
- Other resources have a lifetime that matches that of a particular game level. These resources must be in memory by the time the level is first seen by the player and can be dumped once the player has permanently left the level.
- Some resources might have a lifetime that is shorter than the duration of the level in which they are found. For example, the animations and audio clips that make up an *in-game cinematic* (a mini-movie that advances the story or provides the player with important information) might be loaded in advance of the player seeing the cinematic and then dumped once the cinematic has played.
- Some resources like background music, ambient sound effects or full-screen movies are streamed "live" as they play. The lifetime of this kind of resource is difficult to define, because each byte only persists in memory for a tiny fraction of a second, but the entire piece of music sounds like it lasts for a long period of time. Such assets are typically loaded in chunks of a size that matches the underlying hardware's requirements. For example, a music track might be read in 4 KiB chunks, because that might be the buffer size used by the low-level sound system. Only two

事件	A	B	C	D	E
初始状态	0	0	0	0	0
X 级计数增加	1	1	1	0	0
X级负荷	(1)	(1)	(1)	0	0
X级游戏	1	1	1	0	0
Y 级计数增加	1	2	2	1	1
X 级计数减少	0	1	1	1	1
X 级卸载，Y 级加载	(0)	1	1	(1)	(1)
Y 级游戏	0	1	1	1	1

表 7.2. 两级加载和卸载时的资源使用情况。

块在任何给定时刻都存在于内存中 - 当前正在播放的块和紧随其后正在加载到内存中的块。

何时加载资源的问题通常很容易回答，只要知道玩家何时首次看到该资源即可。然而，何时卸载资源并回收其内存的问题却不那么容易回答。问题在于，许多资源在多个关卡之间共享。我们不希望在关卡 X 完成后卸载资源，然后因为关卡 Y 需要相同的资源而立即重新加载。

解决这个问题的一个方法是对资源进行引用计数。每当需要加载新的游戏关卡时，都会遍历该关卡使用的所有资源列表，并将每个资源的引用计数加一（但它们尚未加载）。接下来，我们遍历所有不需要的关卡的资源，并将其引用计数减一；任何引用计数降至零的资源都将被卸载。最后，我们遍历所有引用计数刚刚从零变为一的资源列表，并将这些资源加载到内存中。

例如，假设关卡 X 使用资源 A、B 和 C，关卡 Y 使用资源 B、C、D 和 E。（B 和 C 在两个关卡之间共享。）表 7.2 显示了玩家在关卡 X 和 Y 中游戏时这五种资源的引用计数。表中，引用计数以粗体显示，表示相应资源实际存在于内存中，灰色背景表示资源不在内存中。括号中的引用计数表示相应资源数据正在加载或卸载。

Event	A	B	C	D	E
Initial state	0	0	0	0	0
Level X counts incremented	1	1	1	0	0
Level X loads	(1)	(1)	(1)	0	0
Level X plays	1	1	1	0	0
Level Y counts incremented	1	2	2	1	1
Level X counts decremented	0	1	1	1	1
Level X unloads, level Y loads	(0)	1	1	(1)	(1)
Level Y plays	0	1	1	1	1

Table 7.2. Resource usage as two levels load and unload.

chunks are ever present in memory at any given moment—the chunk that is currently playing and the chunk immediately following it that is being loaded into memory.

The question of when to load a resource is usually answered quite easily, based on knowledge of when the asset is first seen by the player. However, the question of when to unload a resource and reclaim its memory is not so easily answered. The problem is that many resources are shared across multiple levels. We don't want to unload a resource when level X is done, only to immediately reload it because level Y needs the same resource.

One solution to this problem is to reference-count the resources. Whenever a new game level needs to be loaded, the list of all resources used by that level is traversed, and the reference count for each resource is incremented by one (but they are not loaded yet). Next, we traverse the resources of any unneeded levels and decrement their reference counts by one; any resource whose reference count drops to zero is unloaded. Finally, we run through the list of all resources whose reference count just went from zero to one and load those assets into memory.

For example, imagine that level X uses resources A, B and C, and that level Y uses resources B, C, D and E. (B and C are shared between both levels.) Table 7.2 shows the reference counts of these five resources as the player plays through levels X and Y. In this table, reference counts are shown in boldface type to indicate that the corresponding resource actually exists in memory, while a grey background indicates that the resource is not in memory. A reference count in parentheses indicates that the corresponding resource data is being loaded or unloaded.

7.2.2.7 资源的内存管理

资源管理与内存管理密切相关，因为我们必须决定资源加载后应该存放在内存中的哪个位置。每个资源的目的地并不总是相同的。首先，某些类型的资源必须驻留在视频 RAM 中（或者，在 PlayStation 4 上，驻留在已映射以便通过高速“garlic”总线访问的内存块中）。典型的例子包括纹理、顶点缓冲区、索引缓冲区和着色器代码。大多数其他资源可以驻留在主 RAM 中，但不同类型的资源可能需要驻留在不同的地址范围内。例如，已加载并在整个游戏中驻留的资源（全局资源）可能会被加载到一个内存区域，而频繁加载和卸载的资源可能会驻留在其他地方。

游戏引擎内存分配子系统的设计通常与其资源管理器的设计紧密相关。有时，我们会设计资源管理器以充分利用现有的内存分配器类型，反之亦然——我们可能会设计内存分配器来满足资源管理器的需求。

正如我们在 6.2.1.4 节中看到的，任何资源管理系统面临的主要问题之一是需要避免在资源加载和卸载时产生内存碎片。下面我们将讨论一些更常见的解决方案。

基于堆的资源分配

一种方法是直接忽略内存碎片问题，使用通用堆分配器来分配资源（例如 C 语言中 malloc() 实现的分配器，或 C++ 中的全局 new 运算符）。如果您的游戏仅计划在支持虚拟内存分配的个人电脑上运行，这种方法最有效。在这样的系统上，物理内存会变得碎片化，但操作系统能够将不连续的物理 RAM 页面映射到连续的虚拟内存空间，这有助于减轻碎片化带来的部分影响。

如果你的游戏运行在物理内存有限且只有一个基础的虚拟内存管理器（或者根本没有）的主机上，那么碎片化就会成为一个问题。在这种情况下，一个替代方案是定期对内存进行碎片整理。我们在 6.2.2.2 节中介绍了如何进行碎片整理。

7.2.2.7 Memory Management for Resources

Resource management is closely related to memory management, because we must inevitably decide *where* the resources should end up in memory once they have been loaded. The destination of every resource is not always the same. For one thing, certain types of resources must reside in video RAM (or, on the PlayStation 4, in a memory block that has been mapped for access via the high-speed “garlic” bus). Typical examples include textures, vertex buffers, index buffers and shader code. Most other resources can reside in main RAM, but different kinds of resources might need to reside within different address ranges. For example, a resource that is loaded and stays resident for the entire game (global resources) might be loaded into one region of memory, while resources that are loaded and unloaded frequently might go somewhere else.

The design of a game engine’s memory allocation subsystem is usually closely tied to that of its resource manager. Sometimes we will design the resource manager to take best advantage of the types of memory allocators we have available, or vice versa—we may design our memory allocators to suit the needs of the resource manager.

As we saw in Section 6.2.1.4, one of the primary problems facing any resource management system is the need to avoid fragmenting memory as resources are loaded and unloaded. We’ll discuss a few of the more-common solutions to this problem below.

Heap-Based Resource Allocation

One approach is to simply ignore memory fragmentation issues and use a general-purpose heap allocator to allocate your resources (like the one implemented by `malloc()` in C, or the global `new` operator in C++). This works best if your game is only intended to run on personal computers, on operating systems that support virtual memory allocation. On such a system, physical memory will become fragmented, but the operating system’s ability to map noncontiguous pages of physical RAM into a contiguous virtual memory space helps to mitigate some of the effects of fragmentation.

If your game is running on a console with limited physical RAM and only a rudimentary virtual memory manager (or none whatsoever), then fragmentation will become a problem. In this case, one alternative is to defragment your memory periodically. We saw how to do this in Section 6.2.2.2.

基于堆栈的资源分配

堆栈分配器不会出现碎片问题，因为内存是连续分配的，并且释放顺序与分配的顺序相反。如果满足以下两个条件，则可以使用堆栈分配器加载资源：

- 游戏是线性的且以关卡为中心（即，玩家观看加载屏幕，然后玩一个关卡，然后观看另一个加载屏幕，然后玩另一个关卡）。
- 每个级别都完整地融入记忆中。

假设满足这些要求，我们可以使用堆栈分配器按如下方式加载资源：游戏首次启动时，首先分配全局资源。然后标记堆栈顶部，以便稍后释放回该位置。要加载关卡，我们只需将其资源分配到堆栈顶部即可。关卡完成后，我们只需将堆栈顶部设置回先前标记的位置，从而一次性释放该关卡的所有资源，而不会干扰全局资源。此过程可以重复用于任意数量的关卡，而不会造成内存碎片。图 7.3 演示了如何实现此过程。

可以使用双端堆栈分配器来增强此方法。在单个大型内存块中定义两个堆栈。一个从内存区域的底部向上增长，另一个从顶部向下增长。只要两个堆栈永不重叠，它们就可以自然地来回交换内存资源——如果每个堆栈都位于其自己的固定大小的块中，这是不可能的。

在《Hydro Thunder》中，Midway 使用了双端堆栈分配器。下层堆栈用于持久数据加载，而上层堆栈用于每帧释放的临时分配。双端堆栈分配器的另一种使用方式是乒乓关卡加载。Bionic Games, Inc. 的一个项目就采用了这种方法。基本思路是将关卡 B 的压缩版本加载到上层堆栈中，而当前活动的关卡 A（以未压缩的形式）驻留在下层堆栈中。要从关卡 A 切换到关卡 B，我们只需释放关卡 A 的资源（通过清除下层堆栈），然后将关卡 B 从上层堆栈解压缩到下层堆栈。解压缩通常比从磁盘加载数据快得多，因此这种方法有效地消除了玩家在关卡之间可能遇到的加载时间。

Stack-Based Resource Allocation

A stack allocator does not suffer from fragmentation problems, because memory is allocated contiguously and freed in an order opposite to that in which it was allocated. A stack allocator can be used to load resources if the following two conditions are met:

- The game is linear and level-centric (i.e., the player watches a loading screen, then plays a level, then watches another loading screen, then plays another level).
- Each level fits into memory in its entirety.

Presuming that these requirements are satisfied, we can use a stack allocator to load resources as follows: When the game first starts up, the global resources are allocated first. The top of the stack is then marked, so that we can free back to this position later. To load a level, we simply allocate its resources on the top of the stack. When the level is complete, we can simply set the stack top back to the marker we took earlier, thereby freeing all of the level's resources in one fell swoop without disturbing the global resources. This process can be repeated for any number of levels, without ever fragmenting memory. Figure 7.3 illustrates how this is accomplished.

A double-ended stack allocator can be used to augment this approach. Two stacks are defined within a single large memory block. One grows up from the bottom of the memory area, while the other grows down from the top. As long as the two stacks never overlap, the stacks can trade memory resources back and forth naturally—something that wouldn't be possible if each stack resided in its own fixed size block.

On *Hydro Thunder*, Midway used a double-ended stack allocator. The lower stack was used for persistent data loads, while the upper was used for temporary allocations that were freed every frame. Another way a double-ended stack allocator can be used is to ping-pong level loads. Such an approach was used at Bionic Games, Inc. for one of their projects. The basic idea is to load a compressed version of level B into the upper stack, while the currently active level A resides (in uncompressed form) in the lower stack. To switch from level A to level B, we simply free level A's resources (by clearing the lower stack) and then decompress level B from the upper stack into the lower stack. Decompression is generally much faster than loading data from disk, so this approach effectively eliminates the load time that would otherwise be experienced by the player between levels.

图 7.3. 使用堆栈分配器加载资源。

基于池的资源分配

在支持流式传输的游戏引擎中，另一种常见的资源分配技术是将资源数据加载到大小相同的块中。由于这些块的大小相同，因此可以使用池分配器进行分配（参见第 6.2.1.2 节）。稍后卸载资源时，可以释放这些块而不会造成碎片。

当然，基于块的分配方法要求所有资源数据的布局方式都允许划分成大小相等的块。我们不能简单地将任意资源文件分块加载，因为该文件可能包含连续的数据结构，例如数组，或者比单个块更大的结构体。例如，如果包含数组的块在 RAM 中没有按顺序排列，数组的连续性就会丧失，数组索引将无法正常工作。这意味着所有资源数据的设计都必须考虑“块”的概念。必须避免使用大型连续数据结构，而应选择足够小以容纳单个块或不需要连续 RAM 的数据结构。

Figure 7.3. Loading resources using a stack allocator.

Pool-Based Resource Allocation

Another resource allocation technique that is common in game engines that support streaming is to load resource data in equally sized chunks. Because the chunks are all the same size, they can be allocated using a *pool allocator* (see Section 6.2.1.2). When resources are later unloaded, the chunks can be freed without causing fragmentation.

Of course, a chunk-based allocation approach requires that all resource data be laid out in a manner that permits division into equally sized chunks. We cannot simply load an arbitrary resource file in chunks, because the file might contain a contiguous data structure like an array or a very large `struct` that is larger than a single chunk. For example, if the chunks that contain an array are not arranged sequentially in RAM, the continuity of the array will be lost, and array indexing will cease to function properly. This means that all resource data must be designed with “chunkiness” in mind. Large contiguous data structures must be avoided in favor of data structures that are either small enough to fit within a single chunk or do not require contiguous RAM

图 7.4. X 和 Y 级别的资源大块分配。

才能正常运行（例如，链接列表）。

池中的每个块通常与特定的游戏级别相关联。（一种简单的方法是给每个级别一个其块的链接列表。）这使得引擎可以适当地管理每个块的生命周期，即使具有不同生命周期的多个级别同时在内存中。例如，当加载级别 X 时，它可能会分配并使用 N 个块。稍后，级别 Y 可能会分配另外的 M 个块。当级别 X 最终被卸载时，它的 N 个块将返回到空闲池。如果级别 Y 仍然处于活动状态，则它的 M 个块需要保留在内存中。通过将每个块与特定级别相关联，可以轻松有效地管理块的生命周期。如图 7.4 所示。

“块状”资源分配方案固有的一大弊端是空间浪费。除非资源文件的大小是块大小的整数倍，否则文件中的最后一个块将无法得到充分利用（参见图 7.5）。选择较小的块大小有助于缓解此问题，但块越小，对资源数据布局的限制就越严格。（举一个极端的例子，如果选择的块大小为 1 字节，则任何数据结构都不能大于单个字节——这显然是站不住脚的。）典型的块大小约为几 KB。例如，在顽皮狗，我们使用块状资源分配器作为资源流系统的一部分，我们的块在 PS3 上的大小为 512 KB，在 PS4 上的大小为 1 MB。您可能还需要考虑选择操作系统 I/O 缓冲区大小的倍数，以最大限度地提高加载单个块时的效率。

资源块分配器

限制块内存浪费影响的一种方法是设置一个特殊的内存分配器，该分配器可以利用块中未使用的部分。据我所知

Figure 7.4. Chunky allocation of resources for levels X and Y.

to function properly (e.g., linked lists).

Each chunk in the pool is typically associated with a particular game level. (One simple way to do this is to give each level a linked list of its chunks.) This allows the engine to manage the lifetimes of each chunk appropriately, even when multiple levels with different life spans are in memory concurrently. For example, when level X is loaded, it might allocate and make use of *N* chunks. Later, level Y might allocate an additional *M* chunks. When level X is eventually unloaded, its *N* chunks are returned to the free pool. If level Y is still active, its *M* chunks need to remain in memory. By associating each chunk with a specific level, the lifetimes of the chunks can be managed easily and efficiently. This is illustrated in Figure 7.4.

One big trade-off inherent in a “chunky” resource allocation scheme is wasted space. Unless a resource file’s size is an exact multiple of the chunk size, the last chunk in a file will not be fully utilized (see Figure 7.5). Choosing a smaller chunk size can help to mitigate this problem, but the smaller the chunks, the more onerous the restrictions on the layout of the resource data. (As an extreme example, if a chunk size of one byte were selected, then no data structure could be larger than a single byte—clearly an untenable situation.) A typical chunk size is on the order of a few kibibytes. For example, at Naughty Dog, we use a chunky resource allocator as part of our resource streaming system, and our chunks are 512 KiB in size on the PS3 and 1 MiB on the PS4. You may also want to consider selecting a chunk size that is a multiple of the operating system’s I/O buffer size to maximize efficiency when loading individual chunks.

Resource Chunk Allocators

One way to limit the effects of wasted chunk memory is to set up a special memory allocator that can utilize the unused portions of chunks. As far as I’m

图 7.5。资源文件的最后一块通常没有得到充分利用。

请注意，这种分配器没有标准化的名称，但由于缺乏更好的名称，我们将其称为资源块分配器。

资源块分配器的实现并不特别困难。我们只需要维护一个包含所有未使用内存块的链表，以及每个空闲块的位置和大小。然后，我们可以按照任何我们认为合适的方式从这些空闲块中进行分配。例如，我们可以使用通用堆分配器来管理空闲块的链表。或者，我们可以将一个小型堆栈分配器映射到每个空闲块上；每当有内存请求时，我们就可以扫描空闲块，找到一个堆栈中具有足够可用 RAM 的块，然后使用该堆栈来满足请求。

不幸的是，这里有一个看起来相当怪诞的缺陷。
如果我们在资源块的未使用区域分配内存，那么当这些块被释放时会发生什么？我们无法释放部分块——这是一个“全有或全无”的命题。因此，我们在资源块未使用部分分配的任何内存，在资源卸载时都会神奇地消失。

解决这个问题的一个简单方法是，仅将我们的空闲块分配器用于生命周期与特定块所关联层级的生命周期相匹配的内存请求。换句话说，我们应该只从层级 A 的块中分配与层级 A 独占关联的数据内存，而只从层级 B 的块中分配由层级 B 独占使用的内存。这要求我们的资源块分配器分别管理每个层级的块。并且，它要求块分配器的用户指定他们要为哪个层级分配内存，以便可以使用正确的空闲块链表来满足请求。

值得庆幸的是，大多数游戏引擎在加载资源时需要动态分配内存，这些内存超出了资源文件本身所需的内存。因此，资源块分配器可以有效地回收原本可能被浪费的块内存。

Figure 7.5. The last chunk of a resource file is often not fully utilized.

aware, there is no standardized name for this kind of allocator, but we will call it a *resource chunk allocator* for lack of a better name.

A resource chunk allocator is not particularly difficult to implement. We need only maintain a linked list of all chunks that contain unused memory, along with the locations and sizes of each free block. We can then allocate from these free blocks in any way we see fit. For example, we might manage the linked list of free blocks using a general-purpose heap allocator. Or we might map a small stack allocator onto each free block; whenever a request for memory comes in, we could then scan the free blocks for one whose stack has enough free RAM and then use that stack to satisfy the request.

Unfortunately, there's a rather grotesque-looking fly in our ointment here. If we allocate memory in the unused regions of our resource chunks, what happens when those chunks are freed? We cannot free part of a chunk—it's an all or nothing proposition. So any memory we allocate within an unused portion of a resource chunk will magically disappear when that resource is unloaded.

A simple solution to this problem is to only use our free-chunk allocator for memory requests whose lifetimes match the lifetime of the level with which a particular chunk is associated. In other words, we should only allocate memory out of level A's chunks for data that is associated exclusively with level A and only allocate from B's chunks memory that is used exclusively by level B. This requires our resource chunk allocator to manage each level's chunks separately. And it requires the users of the chunk allocator to specify which level they are allocating for, so that the correct linked list of free blocks can be used to satisfy the request.

Thankfully, most game engines need to allocate memory dynamically when loading resources, over and above the memory required for the resource files themselves. So a resource chunk allocator can be a fruitful way to reclaim chunk memory that would otherwise have been wasted.

分段资源文件

与“块状”资源文件相关的另一个有用想法是文件节的概念。典型的资源文件可能包含一到四个节，每个节又分为一个或多个块，以便如上所述进行池分配。一个节可能包含要发送到主 RAM 的数据，而另一个节可能包含视频 RAM 数据。另一个节可能包含在加载过程中需要的临时数据，但在资源完全加载后会被丢弃。还有一个节可能包含调试信息。在调试模式下运行游戏时可以加载此调试数据，但在游戏的最终生产版本中根本不会加载。Granny SDK 的文件系统（http://www.radgametools.com）是如何以简单灵活的方式实现文件分段的绝佳示例。

7.2.2.8 复合资源和引用完整性

通常，游戏的资源数据库由多个资源文件组成，每个文件包含一个或多个数据对象。这些数据对象可以以任意方式相互引用和依赖。例如，网格数据结构可能包含对其材质的引用，而材质又包含对纹理的引用列表。通常，交叉引用意味着依赖关系（例如，如果资源 A 引用资源 B，则 A 和 B 都必须在内存中，资源才能在游戏中正常运行。）通常，游戏的资源数据库可以用相互依赖的数据对象的有向图来表示。

数据对象之间的交叉引用可以是内部的（即同一文件中两个对象之间的引用），也可以是外部的（即对不同文件中对象的引用）。这种区别非常重要，因为内部和外部交叉引用的实现方式通常不同。在可视化游戏资源数据库时，我们可以在各个资源文件周围画虚线，以清晰地区分内部/外部——图中任何跨越虚线文件边界的边都是外部引用，而不跨越文件边界的边则是内部引用。如图 7.6 所示。

我们有时会使用术语“复合资源”来描述一个自给自足的、相互依赖的资源集群。例如，一个模型就是一个复合资源，由一个或多个三角形网格、一个可选的骨架和一个可选的动画集合组成。每个网格都映射一种材质，每种材质又对应一个或多个纹理。要将像 3D 模型这样的复合资源完全加载到内存中，还必须加载其所有依赖的资源。

Sectioned Resource Files

Another useful idea that is related to “chunky” resource files is the concept of *file sections*. A typical resource file might contain between one and four sections, each of which is divided into one or more chunks for the purposes of pool allocation as described above. One section might contain data that is destined for main RAM, while another section might contain video RAM data. Another section could contain temporary data that is needed during the loading process but is discarded once the resource has been completely loaded. Yet another section might contain debugging information. This debug data could be loaded when running the game in debug mode, but not loaded at all in the final production build of the game. The Granny SDK’s file system (http://www.radgametools.com) is an excellent example of how to implement file sectioning in a simple and flexible manner.

7.2.2.8 Composite Resources and Referential Integrity

Usually a game’s resource database consists of multiple *resource files*, each file containing one or more *data objects*. These data objects can refer to and depend upon one another in arbitrary ways. For example, a mesh data structure might contain a reference to its material, which in turn contains a list of references to textures. Usually cross-references imply dependency (i.e., if resource A refers to resource B, then both A and B must be in memory in order for the resources to be functional in the game.) In general, a game’s resource database can be represented by a *directed graph* of interdependent data objects.

Cross-references between data objects can be *internal* (a reference between two objects within a single file) or *external* (a reference to an object in a different file). This distinction is important because internal and external cross-references are often implemented differently. When visualizing a game’s resource database, we can draw dotted lines surrounding individual resource files to make the internal/external distinction clear—any edge of the graph that crosses a dotted line file boundary is an external reference, while edges that do not cross file boundaries are internal. This is illustrated in Figure 7.6.

We sometimes use the term *composite resource* to describe a self-sufficient cluster of interdependent resources. For example, a *model* is a composite resource consisting of one or more *triangle meshes*, an optional *skeleton* and an optional collection of *animations*. Each mesh is mapped with a *material*, and each material refers to one or more *textures*. To fully load a composite resource like a 3D model into memory, all of its dependent resources must be loaded as well.

图 7.6. 资源数据库依赖关系图的示例。

7.2.2.9 处理资源之间的交叉引用

实现资源管理器最具挑战性的方面之一是管理资源对象之间的交叉引用并确保维护引用完整性。为了理解资源管理器如何实现这一点，让我们看看交叉引用在内存中是如何表示的，以及它们在磁盘上是如何表示的。

在 C++ 中，两个数据对象之间的交叉引用通常通过指针或引用来实现。例如，一个网格可能包含数据成员 Material* m_pMaterial（一个指针）或 Material & m_material（一个引用），以便引用其材质。然而，指针只是内存地址——当脱离正在运行的应用程序上下文时，它们就失去了意义。事实上，即使在同一应用程序的后续运行之间，内存地址也可能会发生变化。显然，在将数据存储到磁盘文件时，我们不能使用指针来描述对象间的依赖关系。

GUID 作为交叉引用

一个好方法是将每个交叉引用存储为包含被引用对象唯一 ID 的字符串或哈希码。这意味着每个可能被交叉引用的资源对象都必须具有全局唯一标识符或 GUID。

Figure 7.6. Example of a resource database dependency graph.

7.2.2.9 Handling Cross-References between Resources

One of the more-challenging aspects of implementing a resource manager is managing the cross-references between resource objects and guaranteeing that referential integrity is maintained. To understand how a resource manager accomplishes this, let's look at how cross-references are represented in memory, and how they are represented on-disk.

In C++, a cross-reference between two data objects is usually implemented via a *pointer* or a *reference*. For example, a mesh might contain the data member `Material* m_pMaterial` (a pointer) or `Material& m_material` (a reference) in order to refer to its material. However, pointers are just memory addresses—they lose their meaning when taken out of the context of the running application. In fact, memory addresses can and do change even between subsequent runs of the same application. Clearly when storing data to a disk file, we cannot use pointers to describe inter-object dependencies.

GUIDs as Cross-References

One good approach is to store each cross-reference as a string or hash code containing the unique id of the referenced object. This implies that every resource object that might be cross-referenced must have a *globally unique identifier* or *GUID*.

图 7.7。保存到二进制文件时，内存中的对象图像变得连续。

为了使这种交叉引用能够正常工作，运行时资源管理器维护了一个全局资源查找表。每当一个资源对象加载到内存中时，指向该对象的指针就会存储在表中，并以该对象的 GUID 作为查找键。所有资源对象都加载到内存中并将其条目添加到表中后，我们可以遍历所有对象，并通过该对象的 GUID 在全局资源查找表中查找每个交叉引用对象的地址，从而将它们的所有交叉引用转换为指针。

指针修复表

将数据对象存储到二进制文件中时，另一种常用的方法是将指针转换为文件偏移量。假设有一组 C 结构体或 C++ 对象，它们通过指针相互交叉引用。要将这组对象存储到二进制文件中，我们需要以任意顺序访问每个对象一次（且仅一次），并将每个对象的内存映像按顺序写入文件。这样做可以将对象序列化为文件中连续的映像，即使它们的内存映像在 RAM 中并不连续。

如图 7.7 所示。

由于对象的内存映像现在在文件中是连续的，我们可以确定每个对象映像相对于文件开头的偏移量。在写入二进制文件映像的过程中，我们定位每个

Figure 7.7. In-memory object images become contiguous when saved into a binary file.

To make this kind of cross-reference work, the runtime resource manager maintains a global resource look-up table. Whenever a resource object is loaded into memory, a pointer to that object is stored in the table with its GUID as the look-up key. After all resource objects have been loaded into memory and their entries added to the table, we can make a pass over all of the objects and convert all of their cross-references into pointers, by looking up the address of each cross-referenced object in the global resource look-up table via that object's GUID.

Pointer Fix-Up Tables

Another approach that is often used when storing data objects into a binary file is to convert the *pointers* into *file offsets*. Consider a group of C structs or C++ objects that cross-reference each other via pointers. To store this group of objects into a binary file, we need to visit each object once (and only once) in an arbitrary order and write each object's memory image into the file sequentially. This has the effect of serializing the objects into a *contiguous* image within the file, even when their memory images are not contiguous in RAM. This is shown in Figure 7.7.

Because the objects' memory images are now contiguous within the file, we can determine the *offset* of each object's image relative to the beginning of the file. During the process of writing the binary file image, we locate every

在每个数据对象中存储指针，将每个指针转换为偏移量，并将这些偏移量存储到文件中，代替指针本身。我们可以简单地用它们的偏移量覆盖指针，因为偏移量所需的存储位数永远不会比原始指针更多。实际上，偏移量相当于内存中指针的二进制文件。（请注意开发平台和目标平台之间的差异。如果您在 64 位 Windows 计算机上写入内存映像，则其指针将全部为 64 位宽，并且生成的文件将与 32 位控制台不兼容。）

当然，当文件稍后加载到内存中时，我们需要将偏移量转换回指针。这种转换称为指针修复。加载文件的二进制映像时，映像中包含的对象将保持其连续的布局，因此将偏移量转换为指针非常简单。我们只需将偏移量添加到整个文件映像的地址即可。下面的代码片段演示了这一点，并在图 7.8 中进行了说明。

```
U8* ConvertOffsetToPointer(U32 objectOffset,
                           U8* pAddressOfFileImage)
{
    U8* pObject = pAddressOfFileImage + objectOffset;
    return pObject;
}
```

图 7.8. 连续的资源文件图像，加载到 RAM 之后。

图 7.9。指针修复表。

pointer within every data object, convert each pointer into an offset and store those offsets into the file in place of the pointers. We can simply overwrite the pointers with their offsets, because the offsets never require more bits to store than the original pointers. In effect, an *offset* is the binary file equivalent of a *pointer* in memory. (Do be aware of the differences between your development platform and your target platform. If you write out a memory image on a 64-bit Windows machine, its pointers will all be 64 bits wide and the resulting file won't be compatible with a 32-bit console.)

Of course, we'll need to convert the offsets back into pointers when the file is loaded into memory some time later. Such conversions are known as *pointer fix-ups*. When the file's binary image is loaded, the objects contained in the image retain their contiguous layout, so it is trivial to convert an offset into a pointer. We merely add the offset to the address of the file image as a whole. This is demonstrated by the code snippet below and illustrated in Figure 7.8.

```
U8* ConvertOffsetToPointer(U32 objectOffset,
                           U8* pAddressOfFileImage)
{
    U8* pObject = pAddressOfFileImage + objectOffset;
    return pObject;
}
```

Figure 7.8. Contiguous resource file image, after it has been loaded into RAM.

Figure 7.9. A pointer fix-up table.

在尝试将指针转换为偏移量（反之亦然）时遇到的问题是如何找到所有需要转换的指针。这个问题通常在写入二进制文件时解决。写出数据对象映像的代码了解正在写入的数据类型和类，因此它了解每个对象内所有指针的位置。指针的位置存储在一个简单的表中，称为指针修复表。此表与所有对象的二进制映像一起写入二进制文件。稍后，当文件再次加载到 RAM 中时，可以查阅该表以查找和修复每个指针。表本身只是文件内的偏移量列表 - 每个偏移量代表一个需要修复的指针。如图 7.9 所示。

将 C++ 对象存储为二进制图像：构造函数

从二进制文件加载 C++ 对象时，一个容易被忽视的重要步骤是确保调用对象的构造函数。例如，如果我们加载一个包含三个对象的二进制图像——类 A 的实例、类 B 的实例和类 C 的实例——那么我们必须确保在这三个对象上分别调用了正确的构造函数。

解决这个问题有两种常见的方法。首先，你可以直接决定在二进制文件中完全不支持 C++ 对象。换句话说，限制使用普通的旧式数据结构（缩写为 PODS 或 POD），即不包含虚函数和不做任何事的构造函数的 C 结构体、C++ 结构体和类。（有关 PODS 的更完整讨论，请参阅 http://en.wikipedia.org/wiki/Plain_Old_Data_ Structures。）

其次，你可以保存一个表，其中包含二进制映像中所有非 PODS 对象的偏移量，以及每个对象所属类的实例信息。然后，二进制映像加载完成后，你可以遍历该表，访问每个对象，并使用 Placement New 语法（即在预分配的内存块上调用构造函数）调用相应的构造函数。例如，给定二进制映像中某个对象的偏移量，我们可以这样写：

```
void* pObject = ConvertOffsetToPointer(objectOffset,
                                       pAddressOfFileImage);
::new(pObject) ClassName; // placement new syntax
```

其中 ClassName 是对象所属的类的实例。

处理外部引用

上述两种方法在应用于所有交叉引用都是内部的资源时非常有效——即，它们只引用

The problem we encounter when trying to convert pointers into offsets, and vice versa, is how to *find* all of the pointers that require conversion. This problem is usually solved at the time the binary file is written. The code that writes out the images of the data objects has knowledge of the data types and classes being written, so it has knowledge of the locations of all the pointers within each object. The locations of the pointers are stored into a simple table known as a *pointer fix-up table*. This table is written into the binary file along with the binary images of all the objects. Later, when the file is loaded into RAM again, the table can be consulted in order to find and fix up every pointer. The table itself is just a list of offsets within the file—each offset represents a single pointer that requires fixing up. This is illustrated in Figure 7.9.

Storing C++ Objects as Binary Images: Constructors

One important step that is easy to overlook when loading C++ objects from a binary file is to ensure that the objects' constructors are called. For example, if we load a binary image containing three objects—an instance of class A, an instance of class B, and an instance of class C—then we must make sure that the correct constructor is called on each of these three objects.

There are two common solutions to this problem. First, you can simply decide not to support C++ objects in your binary files at all. In other words, restrict yourself to plain old data structures (abbreviated PODS or POD)—i.e., C structs and C++ structs and classes that contain *no virtual functions* and *trivial do-nothing constructors*. (See http://en.wikipedia.org/wiki/Plain_Old_Data_ Structures for a more complete discussion of PODS.)

Second, you can save off a table containing the offsets of all non-PODS objects in your binary image along with some indication of which class each object is an instance of. Then, once the binary image has been loaded, you can iterate through this table, visit each object and call the appropriate constructor using *placement new* syntax (i.e., calling the constructor on a preallocated block of memory). For example, given the offset to an object within the binary image, we might write:

```
void* pObject = ConvertOffsetToPointer(objectOffset,
                                       pAddressOfFileImage);
::new(pObject) ClassName; // placement new syntax
```

where *ClassName* is the class of which the object is an instance.

Handling External References

The two approaches described above work very well when applied to resources in which all of the cross-references are *internal*—i.e., they only refer-

ence 对象在单个资源文件中。在这个简单的例子中，您可以将二进制映像加载到内存中，然后应用指针修复来解析所有交叉引用。但是，当交叉引用延伸到其他资源文件时，就需要稍微增强的方法。

为了成功表示外部交叉引用，我们不仅必须指定相关数据对象的偏移量或 GUID，还必须指定引用对象所在的资源文件的路径。

加载多文件复合资源的关键在于首先加载所有相互依赖的文件。具体做法是：加载一个资源文件，然后扫描其交叉引用表，并加载所有尚未加载的外部引用文件。将每个数据对象加载到 RAM 中时，我们可以将对象的地址添加到主查找表中。所有相互依赖的文件加载完毕，并且所有对象都已加载到 RAM 中后，我们就可以进行最后一次操作，使用主查找表修复所有指针，将 GUID 或文件偏移量转换为实际地址。

7.2.2.10 加载后初始化

理想情况下，我们的离线工具会为每一种资源做好充分的准备，使其在加载到内存后即可立即使用。但实际上，这并非总是可行。许多类型的资源在加载后至少需要进行一些“处理”，才能为引擎做好准备。本书中，我将使用“加载后初始化”一词来指代加载后对资源数据进行的任何处理。其他引擎可能会使用不同的术语。（例如，在顽皮狗，我们称之为“记录资源”。）大多数资源管理器还支持在释放资源内存之前执行某种类型的拆卸步骤。（在顽皮狗，我们称之为“记录资源”。）

加载后初始化通常有两种类型：

• 在某些情况下，加载后初始化是不可避免的步骤。例如，在 PC 上，描述 3D 网格的顶点和索引会加载到主 RAM 中，但必须先传输到视频 RAM 中才能渲染。这只能在运行时完成，方法是创建 Direct X 顶点缓冲区或索引缓冲区，锁定它，将数据复制或读取到缓冲区中，然后解锁。

• 在其他情况下，加载后初始化期间进行的处理是可以避免的（即可以移到工具中），但这样做是为了方便或快捷。例如，程序员可能希望将精确弧长的计算添加到我们引擎的样条函数库中。与其花时间修改工具来生成弧长数据，不如

ence objects within a single resource file. In this simple case, you can load the binary image into memory and then apply the pointer fix-ups to resolve all the cross-references. But when cross-references reach out into other resource files, a slightly augmented approach is required.

To successfully represent an *external* cross-reference, we must specify not only the offset or GUID of the data object in question, but also the path to the resource file in which the referenced object resides.

The key to loading a multi-file composite resource is to load *all* of the interdependent files first. This can be done by loading one resource file and then scanning through its table of cross-references and loading any externally referenced files that have not already been loaded. As we load each data object into RAM, we can add the object's address to the master look-up table. Once all of the interdependent files have been loaded and all of the objects are present in RAM, we can make a final pass to fix up all of the pointers using the master look-up table to convert GUIDs or file offsets into real addresses.

7.2.2.10 Post-Load Initialization

Ideally, each and every resource would be completely prepared by our offline tools, so that it is ready for use the moment it has been loaded into memory. Practically speaking, this is not always possible. Many types of resources require at least some "massaging" after having been loaded in order to prepare them for use by the engine. In this book, I will use the term *post-load initialization* to refer to any processing of resource data after it has been loaded. Other engines may use different terminology. (For example, at Naughty Dog we call this *logging in* a resource.) Most resource managers also support some kind of tear-down step prior to a resource's memory being freed. (At Naughty Dog, we call this *logging out* a resource.)

Post-load initialization generally comes in one of two varieties:

- In some cases, post-load initialization is an unavoidable step. For example, on a PC, the vertices and indices that describe a 3D mesh are loaded into main RAM, but they must be transferred into video RAM before they can be rendered. This can only be accomplished at runtime, by creating a Direct X vertex buffer or index buffer, locking it, copying or reading the data into the buffer and then unlocking it.
- In other cases, the processing done during post-load initialization is avoidable (i.e., could be moved into the tools), but is done for convenience or expedience. For example, a programmer might want to add the calculation of accurate arc lengths to our engine's spline library. Rather than spend the time to modify the tools to generate the arc length data,

程序员可能只是在运行时加载后初始化时计算一下。之后，当计算完善后，可以将这段代码移到工具中，从而避免运行时计算的成本。

显然，每种类型的资源对于加载后初始化和卸载都有其独特的要求。因此，资源管理器通常允许根据每种资源类型配置这两个步骤。在像 C 这样的非面向对象语言中，我们可以设想一个查找表，将每种类型的资源映射到一对函数指针，一个用于加载后初始化，一个用于卸载。在像 C++ 这样的面向对象语言中，情况就更简单了——我们可以利用多态性，允许每个类以独特的方式处理加载后初始化和卸载。

在 C++ 中，加载后初始化可以实现为一个特殊的构造函数，而卸载可以在类的析构函数中完成。然而，使用构造函数和析构函数来实现这一点存在一些问题。例如，通常需要先构造所有已加载的对象，然后应用指针修复，最后将加载后初始化作为一个单独的步骤执行。因此，大多数开发人员将加载后初始化和卸载推迟到普通的虚函数中。例如，我们可能会选择使用一对命名合理的虚函数，例如 Init() 和 Destroy() 。

加载后初始化与资源的内存分配策略密切相关，因为新数据通常由初始化例程生成。在某些情况下，加载后初始化步骤生成的数据会扩充从文件加载的数据。（例如，如果我们在加载 Catmull-Rom 样条曲线后计算其各段的弧长，我们可能需要分配一些额外的内存来存储结果。）在其他情况下，加载后初始化期间生成的数据会替换已加载的数据。（例如，出于向后兼容的原因，我们可能允许加载较旧且过时的网格数据，然后自动转换为最新格式。）在这种情况下，在加载后步骤生成新数据后，可能需要部分或全部丢弃已加载的数据。

Hydro Thunder 引擎有一个简单但强大的方法来处理这个问题。它允许以两种方式之一加载资源：(a) 直接加载到内存中的最终位置；(b) 加载到内存的临时区域。在后一种情况下，加载后初始化例程负责将最终数据复制到其最终目的地；加载后初始化完成后，资源的临时副本将被丢弃。这对于加载包含相关和不相关资源的资源文件非常有用

the programmer might simply calculate it at runtime during post-load initialization. Later, when the calculations are perfected, this code can be moved into the tools, thereby avoiding the cost of doing the calculations at runtime.

Clearly, each type of resource has its own unique requirements for post-load initialization and tear-down. So, resource managers typically permit these two steps to be configurable on a per-resource-type basis. In a non-object-oriented language like C, we can envision a look-up table that maps each type of resource to a pair of function pointers, one for post-load initialization and one for tear-down. In an object-oriented language like C++, life is even easier—we can make use of polymorphism to permit each class to handle post-load initialization and tear-down in a unique way.

In C++, post-load initialization could be implemented as a special constructor, and tear-down could be done in the class' destructor. However, there are some problems with using constructors and destructors for this purpose. For example, one typically needs to construct all loaded objects first, then apply pointer fix-ups, and finally perform post-load initialization as a separate step. As such, most developers defer post-load initialization and tear-down to plain old virtual functions. For example, we might choose to use a pair of virtual functions named something sensible like `Init()` and `Destroy()`.

Post-load initialization is closely related to a resource's memory allocation strategy, because new data is often generated by the initialization routine. In some cases, the data generated by the post-load initialization step *augments* the data loaded from the file. (For example, if we are calculating the arc lengths of the segments of a Catmull-Rom spline curve after it has been loaded, we would probably want to allocate some additional memory in which to store the results.) In other cases, the data generated during post-load initialization *replaces* the loaded data. (For example, we might allow mesh data in an older out-of-date format to be loaded and then automatically converted into the latest format for backwards compatibility reasons.) In this case, the loaded data may need to be discarded, either partially or in its entirety, after the post-load step has generated the new data.

The *Hydro Thunder* engine had a simple but powerful way of handling this. It would permit resources to be loaded in one of two ways: (a) directly into its final resting place in memory or (b) into a temporary area of memory. In the latter case, the post-load initialization routine was responsible for copying the finalized data into its ultimate destination; the temporary copy of the resource would be discarded after post-load initialization was complete. This was very useful for loading resource files that contained both relevant and irrelevant

数据。相关数据将被复制到内存中的最终目标位置，而不相关的数据将被丢弃。例如，过时格式的网格数据可以加载到临时内存中，然后通过加载后初始化例程转换为最新格式，而无需浪费任何内存来保留旧格式的数据。

data. The relevant data would be copied into its final destination in memory, while the irrelevant data would be discarded. For example, mesh data in an out-of-date format could be loaded into temporary memory and then converted into the latest format by the post-load initialization routine, without having to waste any memory keeping the old-format data kicking around.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

8
游戏循环和实时模拟

游戏是实时、动态、交互式的计算机模拟。因此，时间在任何电子游戏中都扮演着极其重要的角色。游戏引擎中需要处理许多不同类型的时间——实时、游戏时间、动画的本地时间轴、特定函数中实际消耗的 CPU 周期等等。每个引擎系统对时间的定义和操作方式可能有所不同。我们必须对游戏中所有时间的使用方式有深入的理解。在本章中，我们将了解实时动态模拟软件的工作原理，并探讨时间在此类模拟中常见的作用方式。

8.1 渲染循环

在 Windows PC 或 Macintosh 等操作系统的图形用户界面 (GUI) 中，屏幕上的大部分内容都是静态的。任何窗口都只有一小部分在特定时刻主动改变外观。因此，图形用户界面传统上是通过一种称为“矩形无效”的技术在屏幕上绘制的，该技术只重绘屏幕上内容实际发生变化的一小部分。较早的 2D 电子游戏也使用类似的技术来最大限度地减少需要绘制的像素数量。

8
The Game Loop and Real-Time Simulation

Games are real-time, dynamic, interactive computer simulations. As such, *time* plays an incredibly important role in any electronic game. There are many different kinds of time to deal with in a game engine—real time, game time, the local timeline of an animation, the actual CPU cycles spent within a particular function, and the list goes on. Every engine system might define and manipulate time differently. We must have a solid understanding of all the ways time can be used in a game. In this chapter, we'll take a look at how real-time, dynamic simulation software works and explore the common ways in which time plays a role in such a simulation.

8.1 The Rendering Loop

In a graphical user interface (GUI), of the sort found on a Windows PC or a Macintosh, the majority of the screen's contents are static. Only a small part of any one window is actively changing appearance at any given moment. Because of this, graphical user interfaces have traditionally been drawn on-screen via a technique known as *rectangle invalidation*, in which only the small portions of the screen whose contents have actually changed are redrawn. Older 2D video games used similar techniques to minimize the number of pixels that needed to be drawn.

实时 3D 计算机图形的实现方式截然不同。随着摄像机在 3D 场景中移动，屏幕或窗口的所有内容都会不断变化，因此无效矩形的概念不再适用。取而代之的是，它以与电影类似的方式产生运动和交互的幻觉——通过快速连续地向观看者呈现一系列静止图像。

显然，在屏幕上快速连续地生成静态图像需要一个循环。在实时渲染应用程序中，这有时被称为渲染循环。最简单的渲染循环结构如下：

```
while (!quit)
{
    // Update the camera transform based on interactive
    // inputs or by following a predefined path.
    updateCamera();

    // Update positions, orientations and any other
    // relevant visual state of any dynamic elements
    // in the scene.
    updateSceneElements();

    // Render a still frame into an off-screen frame
    // buffer known as the "back buffer".
    renderScene();

    // Swap the back buffer with the front buffer, making
    // the most recently rendered image visible
    // on-screen. (Or, in windowed mode, copy (blit) the
    // back buffer's contents to the front buffer.
    swapBuffers();
}
```

8.2 游戏循环

游戏由许多相互作用的子系统组成，包括设备 I/O、渲染、动画、碰撞检测和解析、可选的刚体动力学模拟、多人网络、音频等等。大多数游戏引擎子系统在游戏运行时都需要定期维护。然而，这些子系统的维护频率因子系统而异。动画通常需要以 30 或 60 Hz 的频率更新，与渲染子系统同步。然而，动力学（物理）模拟实际上可能需要更频繁的更新（例如 120 Hz）。像 AI 这样的更高级系统可能只需要

Real-time 3D computer graphics are implemented in an entirely different way. As the camera moves about in a 3D scene, the *entire contents* of the screen or window change continually, so the concept of invalid rectangles no longer applies. Instead, an illusion of motion and interactivity is produced in much the same way that a movie produces it—by presenting the viewer with a series of still images in rapid succession.

Obviously, producing a rapid succession of still images on-screen requires a loop. In a real-time rendering application, this is sometimes known as the *render loop*. At its simplest, a rendering loop is structured as follows:

```
while (!quit)
{
    // Update the camera transform based on interactive
    // inputs or by following a predefined path.
    updateCamera();

    // Update positions, orientations and any other
    // relevant visual state of any dynamic elements
    // in the scene.
    updateSceneElements();

    // Render a still frame into an off-screen frame
    // buffer known as the "back buffer".
    renderScene();

    // Swap the back buffer with the front buffer, making
    // the most recently rendered image visible
    // on-screen. (Or, in windowed mode, copy (blit) the
    // back buffer's contents to the front buffer.
    swapBuffers();
}
```

8.2 The Game Loop

A game is composed of many interacting subsystems, including device I/O, rendering, animation, collision detection and resolution, optional rigid body dynamics simulation, multiplayer networking, audio, and the list goes on. Most game engine subsystems require periodic *servicing* while the game is running. However, the *rate* at which these subsystems need to be serviced varies from subsystem to subsystem. Animation typically needs to be updated at a rate of 30 or 60 Hz, in synchronization with the rendering subsystem. However, a dynamics (physics) simulation may actually require more frequent updates (e.g., 120 Hz). Higher-level systems, like AI, might only need to be

每秒服务一次或两次，并且它们根本不必与渲染循环同步。

实现游戏引擎子系统的定期更新有很多方法。我们稍后会探讨一些可能的架构。但目前，我们先来介绍最简单的引擎子系统更新方法——使用单个循环来更新所有内容。这样的循环通常被称为游戏循环，因为它是服务于引擎中所有子系统的主循环。

8.2.1 一个简单的例子：Pong

乒乓球（Pong）是一种著名的乒乓球电子游戏类型，起源于1958年，最初是一款名为《双人网球》（Tennis for Two）的模拟电脑游戏，由威廉·A·希金博坦（William A. Higinbotham）在布鲁克海文国家实验室开发，并在示波器上显示。该游戏类型以其后来在数字计算机上的版本而闻名——Magnavox Oddysey 游戏《乒乓球》（Table Tennis）和雅达利街机游戏《乒乓》（Pong）。

在乒乓球比赛中，球在两个可移动的垂直挡板和两面固定的水平墙之间来回弹动。人类玩家通过控制轮控制挡板的位置。（现代的乒乓球游戏可以通过操纵杆、键盘或其他人机界面设备进行控制。）如果球绕过挡板而没有击中它，则另一方获胜，球将被重置，开始新一轮比赛。

以下伪代码演示了乒乓游戏的游戏循环的核心：

```
void main() // Pong
{
    initGame();

    while (true) // game loop
    {
        readHumanInterfaceDevices();

        if (quitButtonPressed())
        {
            break; // exit the game loop
        }

        movePaddles();

        moveBall();

        collideAndBounceBall();
```


serviced once or twice per second, and they needn't necessarily be synchronized with the rendering loop at all.

There are a number of ways to implement the periodic updating of our game engine subsystems. We'll explore some of the possible architectures in a moment. But for the time being, let's stick with the simplest way to update our engine's subsystems—using a single loop to update everything. Such a loop is often called the *game loop*, because it is the master loop that services every subsystem in the engine.

8.2.1 A Simple Example: Pong

Pong is a well-known genre of table tennis video games that got its start in 1958, in the form of an analog computer game called *Tennis for Two*, created by William A. Higinbotham at the Brookhaven National Laboratory and displayed on an oscilloscope. The genre is best known by its later incarnations on digital computers—the Magnavox Oddysey game *Table Tennis* and the Atari arcade game *Pong*.

In pong, a ball bounces back and forth between two movable vertical paddles and two fixed horizontal walls. The human players control the positions of the paddles via control wheels. (Modern re-implementations allow control via a joystick, the keyboard or some other human interface device.) If the ball passes by a paddle without striking it, the other team wins the point and the ball is reset for a new round of play.

The following pseudocode demonstrates what the game loop of a pong game might look like at its core:

```
void main() // Pong
{
    initGame();

    while (true) // game loop
    {
        readHumanInterfaceDevices();

        if (quitButtonPressed())
        {
            break; // exit the game loop
        }

        movePaddles();

        moveBall();

        collideAndBounceBall();
```

```
            if (ballImpactedSide(LEFT_PLAYER))
            {
                incremenentScore(RIGHT_PLAYER);
                resetBall();
            }
            else if (ballImpactedSide(RIGHT_PLAYER))
            {
                incrementScore(LEFT_PLAYER);
                resetBall();
            }

            renderPlayfield();
        }
    }
```

显然，这个例子有些牵强。最初的乒乓球游戏显然不是通过以每秒 30 帧的速度重绘整个屏幕来实现的。当时，CPU 速度非常慢，几乎无法实时绘制两条球拍线和一个球框。专门的 2D 精灵硬件通常用于在屏幕上绘制移动物体。然而，我们只对这里的概念感兴趣，而不是原始乒乓球的实现细节。

如您所见，游戏首次运行时，它会调用 initGame() 来执行图形系统、人机 I/O 设备、音频系统等可能需要的所有设置。然后进入主游戏循环。while (true) 语句告诉我们循环将永远持续下去，除非内部中断。我们在循环中做的第一件事是读取人机接口设备。我们检查是否有人类玩家按下了“退出”按钮——如果是，我们通过 break 语句退出游戏。接下来，根据控制轮、操纵杆或其他 I/O 设备的当前偏转，在 movePaddles() 中略微向上或向下调整挡板的位置。函数 moveBall() 将球的当前速度矢量添加到其位置，以便在下一帧找到它的新位置。然后在 collideAndBounceBall() 中，检查此位置是否与固定的水平墙壁和挡板发生碰撞。如果检测到碰撞，则重新计算球的位置以考虑任何反弹。我们还会记录球是撞击到屏幕的左边缘还是右边缘。如果撞击到，则表示球没有击中任何挡板，在这种情况下，我们会增加对方玩家的得分，并重置球，以进行下一轮游戏。最后， renderPlay 字段（） 会绘制整个屏幕内容。


```
            if (ballImpactedSide(LEFT_PLAYER))
            {
                incremenentScore(RIGHT_PLAYER);
                resetBall();
            }
            else if (ballImpactedSide(RIGHT_PLAYER))
            {
                incrementScore(LEFT_PLAYER);
                resetBall();
            }

            renderPlayfield();
        }
    }
```

Clearly this example is somewhat contrived. The original pong games were certainly not implemented by redrawing the entire screen at a rate of 30 frames per second. Back then, CPUs were so slow that they could barely muster the power to draw two lines for the paddles and a box for the ball in real time. Specialized 2D sprite hardware was often used to draw moving objects on-screen. However, we're only interested in the concepts here, not the implementation details of the original Pong.

As you can see, when the game first runs, it calls `initGame()` to do whatever set-up might be required by the graphics system, human I/O devices, audio system, etc. Then the main game loop is entered. The statement `while (true)` tells us that the loop will continue forever, unless interrupted internally. The first thing we do inside the loop is to read the human interface device(s). We check to see whether either human player pressed the "quit" button—if so, we exit the game via a `break` statement. Next, the positions of the paddles are adjusted slightly upward or downward in `movePaddles()`, based on the current deflection of the control wheels, joysticks or other I/O devices. The function `moveBall()` adds the ball's current velocity vector to its position in order to find its new position next frame. In `collideAndBounceBall()`, this position is then checked for collisions against both the fixed horizontal walls and the paddles. If collisions are detected, the ball's position is recalculated to account for any bounce. We also note whether the ball impacted either the left or right edge of the screen. This means that it missed one of the paddles, in which case we increment the other player's score and reset the ball for the next round. Finally, `renderPlayfield()` draws the entire contents of the screen.

8.3 游戏循环架构风格

游戏循环可以通过多种不同的方式实现——但其核心通常归结为一个或多个简单的循环，并带有各种修饰。我们将在下文探讨一些较为常见的架构。

8.3.1 Windows 消息泵

在 Windows 平台上，游戏除了服务于游戏引擎本身的各个子系统之外，还需要服务来自 Windows 操作系统的消息。因此，Windows 游戏包含一段称为 消息泵 的代码。其基本思想是，每当 Windows 消息到达时，立即响应，并且仅在没有待处理的 Windows 消息时才响应游戏引擎。消息泵通常如下所示：

```
while (true)
{
    // Service any and all pending Windows messages.
    MSG msg;

    while (PeekMessage(&msg, nullptr, 0, 0) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // No more Windows messages to process -- run one
    // iteration of our "real" game loop.
    RunOneIterationOfGameLoop();
}
```

像这样实现游戏循环的副作用之一是，Windows 消息优先于游戏的渲染和模拟。因此，每当你在桌面上调整游戏窗口大小或拖动游戏窗口时，游戏就会暂时冻结。

8.3.2 回调驱动框架

大多数游戏引擎子系统和第三方游戏中间件包都以库的形式构建。库是一套函数和/或类，应用程序程序员可以以任何合适的方式调用它们。库为程序员提供了最大的灵活性。但是，库有时很难使用，因为程序员必须了解如何正确使用它们提供的函数和类。

8.3 Game Loop Architectural Styles

Game loops can be implemented in a number of different ways—but at their core, they usually boil down to one or more simple loops, with various embellishments. We'll explore a few of the more common architectures below.

8.3.1 Windows Message Pumps

On a Windows platform, games need to service messages from the Windows operating system in addition to servicing the various subsystems in the game engine itself. Windows games therefore contain a chunk of code known as a *message pump*. The basic idea is to service Windows messages whenever they arrive and to service the game engine only when no Windows messages are pending. A message pump typically looks something like this:

```
while (true)
{
    // Service any and all pending Windows messages.
    MSG msg;

    while (PeekMessage(&msg, nullptr, 0, 0) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // No more Windows messages to process -- run one
    // iteration of our "real" game loop.
    RunOneIterationOfGameLoop();
}
```

One of the side-effects of implementing the game loop like this is that Windows messages take precedence over rendering and simulating the game. As a result, the game will temporarily freeze whenever you resize or drag the game's window around on the desktop.

8.3.2 Callback-Driven Frameworks

Most game engine subsystems and third-party game middleware packages are structured as *libraries*. A library is a suite of functions and/or classes that can be called in any way the application programmer sees fit. Libraries provide maximum flexibility to the programmer. But, libraries are sometimes difficult to use, because the programmer must understand how to properly use the functions and classes they provide.

相比之下，一些游戏引擎和游戏中间件包则采用框架结构。框架是一个部分构建的应用程序——程序员通过提供框架中缺失功能的自定义实现（或覆盖其默认行为）来完成应用程序。但是，程序员对应用程序内部的整体控制流程几乎没有控制权，因为它由框架控制。

在基于框架的渲染引擎或游戏引擎中，主游戏循环已经为我们编写完毕，但它基本上是空的。游戏程序员可以编写回调函数来“填补”缺失的细节。OGRE 渲染引擎就是一个被框架包装的库的例子。在最底层，OGRE 提供了可供游戏引擎程序员直接调用的函数。然而，OGRE 也提供了一个框架，该框架封装了如何有效使用底层 OGRE 库的知识。如果程序员选择使用 OGRE 框架，他/她需要从 Ogre::FrameListener 派生一个类，并重写两个虚函数：frameStarted() 和 frameEnded()。正如你可能猜到的那样，这两个函数分别在 OGRE 渲染主 3D 场景之前和之后被调用。OGRE 框架的内部游戏循环实现类似于以下伪代码。（请参阅 OgreRoot.cpp 中的 Ogre::Root::renderOneFrame() 获取实际源代码。）

```
while (true)
{
    for (each frameListener)
    {
        frameListener.frameStarted();
    }

    renderCurrentScene();

    for (each frameListener)
    {
        frameListener.frameEnded();
    }

    finalizeSceneAndSwapBuffers();
}
```

特定游戏的帧监听器实现可能看起来像这样。

In contrast, some game engines and game middleware packages are structured as *frameworks*. A framework is a partially constructed application—the programmer completes the application by providing custom implementations of missing functionality within the framework (or overriding its default behavior). But he or she has little or no control over the overall flow of control within the application, because it is controlled by the framework.

In a framework-based rendering engine or game engine, the main game loop has been written for us, but it is largely empty. The game programmer can write callback functions in order to “fill in” the missing details. The OGRE rendering engine is an example of a library that has been wrapped in a framework. At the lowest level, OGRE provides functions that can be called directly by a game engine programmer. However, OGRE also provides a framework that encapsulates knowledge of how to use the low-level OGRE library effectively. If the programmer chooses to use the OGRE framework, he or she derives a class from `Ogre::FrameListener` and overrides two virtual functions: `frameStarted()` and `frameEnded()`. As you might guess, these functions are called before and after the main 3D scene has been rendered by OGRE, respectively. The OGRE framework’s implementation of its internal game loop looks something like the following pseudocode. (See `Ogre::Root::renderOneFrame()` in *OgreRoot.cpp* for the actual source code.)

```
while (true)
{
    for (each frameListener)
    {
        frameListener.frameStarted();
    }

    renderCurrentScene();

    for (each frameListener)
    {
        frameListener.frameEnded();
    }

    finalizeSceneAndSwapBuffers();
}
```

A particular game’s frame listener implementation might look something like this.

```
class GameFrameListener : public Ogre::FrameListener
{
public:
    virtual void frameStarted(const FrameEvent& event)
    {
        // Do things that must happen before the 3D scene
        // is rendered (i.e., service all game engine
        // subsystems).
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);

        // etc.
    }

    virtual void frameEnded(const FrameEvent& event)
    {
        // Do things that must happen after the 3D scene
        // has been rendered.
        drawHud(event);

        // etc.
    }
};
```

8.3.3 基于事件的更新

在游戏中，事件是指游戏状态或其环境发生的任何有趣变化。例如：人类玩家按下游戏手柄上的按钮、发生爆炸、敌方角色发现玩家等等。大多数游戏引擎都拥有事件系统，它允许各种引擎子系统注册对特定类型事件的关注，并在这些事件发生时做出响应（详情请参阅第 16.8 节）。游戏的事件系统通常与几乎所有图形用户界面的底层事件/消息系统非常相似（例如，Microsoft Windows 的窗口消息、Java AWT 中的事件处理系统，以及 C# 的委托和事件关键字提供的服务）。

一些游戏引擎利用其事件系统来实现对部分或全部子系统的定期服务。为此，事件系统必须允许将事件提交到未来，即排队等待后续交付。这样，游戏引擎只需提交事件即可实现定期更新。在事件处理程序中，代码可以执行


```
class GameFrameListener : public Ogre::FrameListener
{
public:
    virtual void frameStarted(const FrameEvent& event)
    {
        // Do things that must happen before the 3D scene
        // is rendered (i.e., service all game engine
        // subsystems).
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);

        // etc.
    }

    virtual void frameEnded(const FrameEvent& event)
    {
        // Do things that must happen after the 3D scene
        // has been rendered.
        drawHud(event);

        // etc.
    }
};
```

8.3.3 Event-Based Updating

In games, an *event* is any interesting change in the state of the game or its environment. Some examples include: the human player pressing a button on the joypad, an explosion going off, an enemy character spotting the player, and the list goes on. Most game engines have an *event system*, which permits various engine subsystems to register interest in particular kinds of events and to respond to those events when they occur (see Section 16.8 for details). A game's event system is usually very similar to the event/messaging system underlying virtually all graphical user interfaces (for example, Microsoft Windows' window messages, the event handling system in Java's AWT or the services provided by C#'s `delegate` and `event` keywords).

Some game engines leverage their event system in order to implement the periodic servicing of some or all of their subsystems. For this to work, the event system must permit events to be posted into the future—that is, to be queued for later delivery. A game engine can then implement periodic updating by simply posting an event. In the event handler, the code can perform

无论需要什么周期性服务。然后，它可以在未来 1/30 秒或 1/60 秒后发布新事件，从而在需要时持续提供周期性服务。

8.4 抽象时间线

在游戏编程中，用抽象的时间轴来思考是非常有用的。时间轴是一个连续的一维轴，其原点（t = 0）可以位于系统中相对于其他时间轴的任意位置。时间轴可以通过一个简单的时钟变量来实现，该变量以整数或浮点格式存储绝对时间值。

8.4.1 实时

我们可以将直接通过 CPU 的高精度定时器寄存器（参见第 8.5.3 节）测量的时间视为位于我们称之为真实时间线 (real timeline) 的坐标上。该时间线的原点定义为 CPU 上次开机或复位的时刻。它以 CPU 周期（或其倍数）为单位测量时间，尽管这些时间值可以通过乘以当前 CPU 上高精度定时器的频率轻松转换为秒为单位。

8.4.2 游戏时间

我们不必局限于只使用真实时间轴。为了解决手头的问题，我们可以根据需要定义任意数量的其他时间轴。例如，我们可以定义一条技术上独立于真实时间的游戏时间轴。在正常情况下，游戏时间与真实时间一致。如果我们想暂停游戏，只需暂时停止更新游戏时间轴即可。如果我们想让游戏进入慢动作，我们可以将游戏时钟的更新速度设置为比真实时钟更慢。通过缩放和扭曲一条时间轴相对于另一条时间轴，可以实现各种效果。

暂停或减慢游戏时钟也是□□一种非常有用的调试工具。为了追踪视觉异常，开发者可以暂停游戏时间以冻结动作。与此同时，渲染引擎和调试飞越摄像机可以继续运行，只要它们由不同的时钟（实时时钟或单独的摄像机时钟）控制。这使得开发者可以操控摄像机在游戏世界中飞行，从任何角度进行观察。我们甚至可以支持单步调整游戏时钟，每次将游戏时钟推进一个目标帧间隔（例如，1/30 秒）。

whatever periodic servicing is required. It can then post a new event 1/30 or 1/60 of a second into the future, thus continuing the periodic servicing for as long as it is required.

8.4 Abstract Timelines

In game programming, it can be extremely useful to think in terms of *abstract timelines*. A timeline is a continuous, one-dimensional axis whose origin ($t = 0$) can lie at any arbitrary location relative to other timelines in the system. A timeline can be implemented via a simple clock variable that stores absolute time values in either integer or floating-point format.

8.4.1 Real Time

We can think of times measured directly via the CPU's high-resolution timer register (see Section 8.5.3) as lying on what we'll call the *real timeline*. The origin of this timeline is defined to coincide with the moment the CPU was last powered on or reset. It measures times in units of CPU cycles (or some multiple thereof), although these time values can be easily converted into units of seconds by multiplying them by the frequency of the high-resolution timer on the current CPU.

8.4.2 Game Time

We needn't limit ourselves to working with the real timeline exclusively. We can define as many other timeline(s) as we need in order to solve the problems at hand. For example, we can define a *game timeline* that is technically independent of real time. Under normal circumstances, game time coincides with real time. If we wish to pause the game, we can simply stop updating the game timeline temporarily. If we want our game to go into slow motion, we can update the game clock more slowly than the real-time clock. All sorts of effects can be achieved by scaling and warping one timeline relative to another.

Pausing or slowing down the game clock is also a highly useful debugging tool. To track down a visual anomaly, a developer can pause game time in order to freeze the action. Meanwhile, the rendering engine and debug fly-through camera can continue to run, as long as they are governed by a different clock (either the *real-time clock* or a separate *camera clock*). This allows the developer to fly the camera around the game world to inspect it from any angle desired. We can even support single-stepping the game clock, by advancing the game clock by one target frame interval (e.g., 1/30 of a second) each time

当游戏处于暂停状态时，按下游戏手柄或键盘上的“单步”按钮。

使用上述方法时，务必意识到游戏暂停时游戏循环仍在运行——只是游戏时钟停止了。通过在暂停的游戏时钟上增加 1/30 秒来单步执行游戏，与在主循环中设置断点，然后反复按 F5 键一次运行一次循环不同。这两种单步执行方法都有助于追踪不同类型的问题。我们只需记住这两种方法之间的区别即可。

8.4.3 本地和全局时间线

我们可以设想各种其他的时间轴。例如，动画剪辑或音频剪辑可能有一个本地时间轴，其原点（t = 0）定义为与剪辑的起始点重合。本地时间轴测量剪辑最初创作或录制时的时间进展。在游戏中播放剪辑时，我们不必以原始速率播放。我们可能想要加快动画速度，或减慢音频采样速度。我们甚至可以通过反向运行本地时钟来倒放动画。

任何一种效果都可以可视化为局部时间轴与全局时间轴（例如实时或游戏时间）之间的映射。为了以原始速度播放动画剪辑，我们只需将动画局部时间轴的起点（t = 0）映射到全局时间轴上的所需起始时间（τ = τ start）。如图 8.1 所示。

为了以一半的速度播放动画剪辑，我们可以设想将局部时间轴缩放至其原始大小的两倍，然后再将其映射到全局时间轴上。为此，除了剪辑的全局起始时间τstart之外，我们只需跟踪时间缩放因子或播放速率R。如图8.2所示。甚至可以通过使用负时间缩放（R<0）来反向播放剪辑，如图8.3所示。

图 8.1。播放动画剪辑可以被视为将其本地时间轴映射到全局游戏时间轴上。

a “single-step” button is pressed on the joypad or keyboard while the game is in a paused state.

When using the approach described above, it’s important to realize that the game loop is still running when the game is paused—only the game clock has stopped. Single-stepping the game by adding 1/30 of a second to a paused game clock is not the same thing as setting a breakpoint in your main loop, and then hitting the F5 key repeatedly to run one iteration of the loop at a time. Both kinds of single-stepping can be useful for tracking down different kinds of problems. We just need to keep the differences between these approaches in mind.

8.4.3 Local and Global Timelines

We can envision all sorts of other timelines. For example, an animation clip or audio clip might have a *local timeline*, with its origin ($t = 0$) defined to coincide with the start of the clip. The local timeline measures how time progressed when the clip was originally authored or recorded. When the clip is played back in-game, we needn’t play it at the original rate. We might want to speed up an animation, or slow down an audio sample. We can even play an animation backwards by running its local clock in reverse.

Any one of these effects can be visualized as a *mapping* between the local timeline and a global timeline, such as real time or game time. To play an animation clip back at its originally authored speed, we simply map the start of the animation’s local timeline ($t = 0$) onto the desired start time ($\tau = \tau_{\text{start}}$) along the global timeline. This is shown in Figure 8.1.

To play an animation clip back at half speed, we can imagine scaling the local timeline to twice its original size prior to mapping it onto the global timeline. To accomplish this, we simply keep track of a time scale factor or playback rate R, in addition to the clip’s global start time τ_{start}. This is illustrated in Figure 8.2. A clip can even be played in reverse, by using a negative time scale ($R < 0$) as shown in Figure 8.3.

Figure 8.1. Playing an animation clip can be visualized as mapping its local timeline onto the global game timeline.

图 8.2. 动画播放速度可以通过在将本地时间线映射到全局时间线之前对其进行缩放来控制。

图 8.3 反向播放动画就像将剪辑映射到时间尺度为 R = −1 的全局时间轴。

8.5 测量和处理时间

在本节中，我们将研究不同类型的时间线和时钟之间的一些细微和不太细微的区别，并了解它们在实际游戏引擎中的实现方式。

8.5.1 帧速率和时间增量

实时游戏的帧速率描述了静态 3D 帧序列呈现给观看者的速度。赫兹 (Hz) 的单位定义为每秒的周期数，可用于描述任何周期性过程的速率。在游戏和电影中，帧速率通常以每秒帧数 (FPS) 来衡量，这实际上与赫兹是同一个概念。电影通常以 24 FPS 的速度播放。北美和日本的游戏通常以 30 或 60 FPS 渲染，因为这是这些地区使用的 NTSC 彩色电视标准的自然刷新率。在欧洲和其他大多数国家/地区，

Figure 8.2. Animation playback speed can be controlled by simply scaling the local timeline prior to mapping it onto the global timeline.

Figure 8.3. Playing an animation in reverse is like mapping the clip to the global timeline with a time scale of $R = -1$.

8.5 Measuring and Dealing with Time

In this section, we'll investigate some of the subtle and not-so-subtle distinctions between different kinds of timelines and clocks and see how they are implemented in real game engines.

8.5.1 Frame Rate and Time Deltas

The *frame rate* of a real-time game describes how rapidly the sequence of still 3D frames is presented to the viewer. The unit of *Hertz* (Hz), defined as the number of cycles per second, can be used to describe the rate of any periodic process. In games and film, frame rate is typically measured in *frames per second* (FPS), which is the same thing as Hertz for all intents and purposes. Films traditionally run at 24 FPS. Games in North America and Japan are typically rendered at 30 or 60 FPS, because this is the natural refresh rate of the NTSC color television standard used in these regions. In Europe and most of the rest

世界上的游戏更新速度为 50 FPS，因为这是 PAL 或 SECAM 彩色电视信号的自然刷新率。

帧与帧之间的时间间隔称为帧时间、时间增量或时间增量。最后一个术语很常见，因为帧与帧之间的持续时间在数学上通常用符号 Δ t 表示。（从技术上讲，Δ t 实际上应该称为帧周期，因为它是帧频率的倒数：T = 1/f。但是，游戏程序员几乎从不在这种情况下使用"周期"一词。）如果游戏以 30 FPS 的速度渲染，则其时间增量为 1/30 秒，即 33.3 毫秒（毫秒）。在 60 FPS 下，时间增量是其一半，为 1/60 秒或 16.6 毫秒。要真正了解游戏循环一次迭代所用的时间，我们需要对其进行测量。下面我们将看到如何做到这一点。

这里需要注意的是，毫秒是游戏中常用的时间度量单位。例如，我们可以说动画系统运行耗时 4 毫秒，这意味着它占据了整帧的 12% 左右（4/33.3 ≈ 0.12）。其他常用单位包括秒和机器周期。

我们将在下面更深入地讨论时间单位和时钟变量。

8.5.2 从帧速率到速度

假设我们想要让一艘宇宙飞船以每秒 40 米的恒定速度（在 2D 游戏中，我们可以将其指定为每秒 40 像素）飞过我们的游戏世界。实现此目标的一个简单方法是将飞船的速度 v（以米/秒为单位）乘以一帧的持续时间 Δ t（以秒为单位），得出位置变化 Δ x = v Δ t（以米/帧为单位）。然后可以将这个位置变化量添加到飞船的当前位置 x 1 中，以便找到下一帧的位置：x 2 = x 1 + Δ x = x 1 + v Δ t。

这实际上是一种简单的数值积分形式，称为显式欧拉方法（参见 13.4.4 节）。只要物体的速度大致恒定，这种方法就能很好地工作。为了处理变化的速度，我们需要采用稍微复杂一些的积分方法。但所有数值积分技术都会以某种方式利用已用帧时间 Δ t。因此，可以肯定地说，游戏中物体的感知速度取决于帧持续时间 Δ t。因此，游戏编程中的一个核心问题是确定 Δ t 的合适值。在接下来的章节中，我们将讨论实现此目的的各种方法。

8.5.2.1 依赖 CPU 的老派游戏

在许多早期的电子游戏中，并没有尝试去测量游戏循环中实际经过了多少时间。程序员基本上会

of the world, games update at 50 FPS, because this is the natural refresh rate of a PAL or SECAM color television signal.

The amount of time that elapses between frames is known as the *frame time*, *time delta* or *delta time*. This last term is commonplace because the duration between frames is often represented mathematically by the symbol Δt. (Technically speaking, Δt should really be called the *frame period*, since it is the inverse of the *frame frequency*: $T = 1/f$. But, game programmers hardly ever use the term "period" in this context.) If a game is being rendered at exactly 30 FPS, then its delta time is 1/30 of a second, or 33.3 ms (milliseconds). At 60 FPS, the delta time is half as big, 1/60 of a second or 16.6 ms. To really know how much time has elapsed during one iteration of the game loop, we need to measure it. We'll see how this is done below.

We should note here that milliseconds are a common unit of time measurement in games. For example, we might say that the animation system is taking 4 ms to run, which implies that it occupies about 12% of the entire frame ($4/33.3 \approx 0.12$). Other common units include seconds and machine cycles. We'll discuss time units and clock variables in more depth below.

8.5.2 From Frame Rate to Speed

Let's imagine that we want to make a spaceship fly through our game world at a constant speed of 40 meters per second (or in a 2D game, we might specify this as 40 *pixels* per second). One simple way to accomplish this is to multiply the ship's speed v (measured in meters per second) by the duration of one frame Δt (measured in seconds), yielding a change in position $\Delta x = v\Delta t$ (which is measured in *meters per frame*). This position delta can then be added to the ship's current position x_1, in order to find its position next frame: $x_2 = x_1 + \Delta x = x_1 + v\Delta t$.

This is actually a simple form of *numerical integration* known as the *explicit Euler* method (see Section 13.4.4). It works well as long as the speeds of our objects are roughly constant. To handle variable speeds, we need to resort to somewhat more complex integration methods. But all numerical integration techniques make use of the elapsed frame time Δt in one way or another. So it is safe to say that the *perceived speeds* of the objects in a game are dependent upon the frame duration, Δt. Hence a central problem in game programming is to determine a suitable value for Δt. In the sections that follow, we'll discuss various ways of doing this.

8.5.2.1 Old-School CPU-Dependent Games

In many early video games, no attempt was made to measure how much real time had elapsed during the game loop. The programmers would essentially

完全忽略 Δ t ，而是直接用米（或像素，或其他距离单位）每帧来表示物体的速度。换句话说，他们可能在不知不觉中用 Δ x = v Δ t 来表示物体的速度，而不是用 v 来表示。

这种过于简化的方法最终导致这些游戏中物体的感知速度完全取决于游戏在特定硬件上实际达到的帧速率。如果这类游戏在 CPU 速度比其最初编写的机器更快的计算机上运行，□□游戏就会看起来像是在快进。因此，我将这些游戏称为“CPU 依赖型游戏”。

一些老款电脑配备了“Turbo”按钮来支持这类游戏。按下“Turbo”按钮时，电脑会以最快速度运行，但依赖CPU的游戏则会快进运行。不按下“Turbo”按钮时，电脑会模拟老款电脑的处理器速度，从而允许为这些电脑编写的依赖CPU的游戏正常运行。

8.5.2.2 根据经过的时间进行更新

为了使我们的游戏独立于 CPU，我们必须以某种方式测量 Δ t，而不是简单地忽略它。这样做非常简单。我们只需读取两次 CPU 高分辨率计时器的值——一次在帧开始时，一次在帧结束时。然后我们将其减去，得到刚刚过去的帧的 Δ t 的精确测量值。然后，所有需要它的引擎子系统都可以使用此增量，方法是将其传递给我们在游戏循环中调用的每个函数，或者将其存储在全局变量中，或者将其封装在某种单例类中。（我们将在 8.5.3 节中更详细地介绍 CPU 的高分辨率计时器。）许多游戏引擎都使用了上述方法。事实上，我敢大胆地说，大多数游戏引擎都在使用这种方法。然而，这项技术存在一个大问题：我们使用在第 k 帧期间测量的 Δ t 值来估计下一帧（k + 1）的时长。这不一定非常准确。（正如投资界所说，“过往业绩并不能保证未来的结果。”）下一帧可能发生某些事情，导致其耗时比当前帧长得多（或短得多）。我们将此类事件称为帧率峰值。

使用上一帧的增量来估计下一帧可能会产生一些非常严重的负面影响。例如，如果我们不小心，它可能会把游戏推入糟糕的帧时间的“恶性循环”。假设我们的物理模拟在每 33.3 毫秒（即 30 Hz）更新一次时最稳定。如果我们遇到一个糟糕的帧，比如说 57 毫秒，那么我们可能会让

ignore Δt altogether and instead specify the speeds of objects directly in terms of meters (or pixels, or some other distance unit) per *frame*. In other words, they were, perhaps unwittingly, specifying object speeds in terms of $\Delta x = v\Delta t$, instead of in terms of v.

The net effect of this simplistic approach was that the perceived speeds of the objects in these games were entirely dependent upon the frame rate that the game was actually achieving on a particular piece of hardware. If this kind of game were to be run on a computer with a faster CPU than the machine for which it was originally written, the game would appear to be running in fast forward. For this reason, I'll call these games *CPU-dependent games*.

Some older PCs provided a "Turbo" button to support these kinds of games. When the Turbo button was pressed, the PC would run at its fastest speed, but CPU-dependent games would run in fast forward. When the Turbo button was not pressed, the PC would mimic the processor speed of an older generation of PCs, allowing CPU-dependent games written for those PCs to run properly.

8.5.2.2 Updating Based on Elapsed Time

To make our games CPU-independent, we must measure Δt in some way, rather than simply ignoring it. Doing this is quite straightforward. We simply read the value of the CPU's high-resolution timer twice—once at the beginning of the frame and once at the end. Then we subtract, producing an accurate measure of Δt for the frame that has just passed. This delta is then made available to all engine subsystems that need it, either by passing it to every function that we call from within the game loop or by storing it in a global variable or encapsulating it within a singleton class of some kind. (We'll describe the CPU's high-resolution timer in more detail in Section 8.5.3.)

The approach outlined above is used by many game engines. In fact, I am tempted to go out on a limb and say that *most* game engines use it. However, there is one big problem with this technique: We are using the measured value of Δt taken during frame k as an estimate of the duration of the *upcoming* frame $(k + 1)$. This isn't necessarily very accurate. (As they say in investing, "past performance is not a guarantee of future results.") Something might happen next frame that causes it to take much more time (or much less) than the current frame. We call such an event a *frame-rate spike*.

Using last frame's delta as an estimate of the upcoming frame can have some very real detrimental effects. For example, if we're not careful it can put the game into a "viscious cycle" of poor frame times. Let's assume that our physics simulation is most stable when updated once every 33.3 ms (i.e., at 30 Hz). If we get one bad frame, taking say 57 ms, then we might make the

错误地在下一帧中两次启动物理系统，大概是为了“弥补”已经过去的57毫秒。这两个步骤的完成时间大约是常规步骤的两倍，导致下一帧至少和这一帧一样糟糕，甚至可能更糟。这只会加剧和延长问题。

8.5.2.3 使用移动平均值

诚然，游戏循环往往至少具有一定的帧间连贯性。如果在某一帧中，摄像机指向一条包含大量绘制成本高昂的物体的走廊，那么很有可能在下一帧中它仍然指向这条走廊。因此，一种合理的方法是对少数帧的帧时间测量值取平均值，并将其用作下一帧的Δ t 估计值。这使得游戏能够适应变化的帧速率，同时减轻瞬时性能峰值的影响。平均间隔越长，游戏对变化的帧速率的响应越慢，但峰值的影响也会减小。

8.5.2.4 控制帧速率

通过颠倒问题，我们可以完全避免使用上一帧的 Δ t 作为此帧持续时间估计值的不准确性。我们不必猜测下一帧的持续时间，而是尝试保证每一帧的持续时间恰好为 33.3 毫秒（如果以 60 FPS 运行，则为 16.6 毫秒）。为此，我们像以前一样测量当前帧的持续时间。如果测得的持续时间小于理想帧时间，我们只需让主线程进入睡眠状态，直到目标帧时间过去。如果测得的持续时间大于理想帧时间，我们必须“承受损失”并等待另一个完整的帧时间过去。这称为帧速率控制。

显然，这种方法只有当游戏的平均帧率相当接近目标帧率时才有效。如果游戏由于频繁出现“慢帧”而导致帧率在 30 FPS 和 15 FPS 之间来回波动，游戏质量可能会显著下降。因此，设计所有引擎系统使其能够处理任意帧时长仍然是一个好主意。在开发过程中，您可以将引擎保持在“可变帧率”模式，一切将按预期运行。之后，当游戏越来越接近稳定地达到目标帧率时，我们就可以启用帧率控制，开始享受它带来的好处。

出于多种原因，保持帧速率一致非常重要。
某些引擎系统（例如物理模拟中使用的数值积分器）在以恒定速率更新时运行最佳。一致的帧速率

mistake of stepping the physics system *twice* on the next frame, presumably to “cover” the 57 ms that has passed. Those two steps take roughly twice as long to complete as a regular step, causing the *next* frame to be at least as bad as this one was, and possibly worse. This only serves to exacerbate and prolong the problem.

8.5.2.3 Using a Running Average

It is true that game loops tend to have at least some frame-to-frame coherency. If the camera is pointed down a hallway containing lots of expensive-to-draw objects on one frame, there’s a good chance it will still be pointed down that hallway on the next. Therefore, one reasonable approach is to average the frame-time measurements over a small number of frames and use that as the next frame’s estimate of Δt. This allows the game to adapt to a varying frame rate, while softening the effects of momentary performance spikes. The longer the averaging interval, the less responsive the game will be to a varying frame rate, but spikes will have less of an impact as well.

8.5.2.4 Governing the Frame Rate

We can avoid the inaccuracy of using last frame’s Δt as an estimate of this frame’s duration altogether, by flipping the problem on its head. Rather than trying to *guess* at what next frame’s duration will be, we can instead attempt to *guarantee* that every frame’s duration will be exactly 33.3 ms (or 16.6 ms if we’re running at 60 FPS). To do this, we measure the duration of the current frame as before. If the measured duration is less than the ideal frame time, we simply put the main thread to sleep until the target frame time has elapsed. If the measured duration is more than the ideal frame time, we must “take our lumps” and wait for one more whole frame time to elapse. This is called *frame-rate governing*.

Clearly this approach only works when your game’s frame rate is reasonably close to your target frame rate on average. If your game is ping-ponging between 30 FPS and 15 FPS due to frequent “slow” frames, then the game’s quality can degrade significantly. As such, it’s still a good idea to design all engine systems so that they are capable of dealing with arbitrary frame durations. During development, you can leave the engine in “variable frame rate” mode, and everything will work as expected. Later on, when the game is getting closer to achieving its target frame rate consistently, we can switch on frame-rate governing and start to reap its benefits.

Keeping the frame rate consistent can be important for a number of reasons. Some engine systems, such as the numerical integrators used in a physics simulation, operate best when updated at a constant rate. A consistent frame rate

看起来也更好，正如我们将在下一节中看到的，它可用于避免当视频缓冲区以与显示器刷新率不匹配的速率更新时可能发生的撕裂（参见第 8.5.2.5 节）。

此外，当帧速率一致时，录制和回放等功能会变得更加可靠。顾名思义，录制回放功能可以记录玩家的游戏体验，并在之后以完全相同的方式回放。这不仅是一项有趣的游戏功能，也是一个宝贵的测试和调试工具。例如，只需回放一段演示该漏洞的录制游戏，就可以重现一些难以发现的漏洞。

为了实现记录和回放，我们会记录游戏过程中发生的每个相关事件，并将每个事件连同准确的时间戳一起保存在一个列表中。然后，可以使用相同的初始条件和相同的初始随机种子，以完全相同的时间重播事件列表。理论上，这样做应该能带来与原始游戏体验无异的游戏体验。然而，如果帧速率不一致，事件发生的顺序可能不会完全相同。这可能会导致“漂移”，很快你的AI角色就会在本应撤退时进行侧翼攻击。

8.5.2.5 屏幕撕裂和垂直同步

当屏幕仅由视频硬件部分“绘制”时，后缓冲区与前缓冲区发生交换时，就会出现一种称为屏幕撕裂的视觉异常现象。发生撕裂时，屏幕的一部分会显示新图像，而其余部分则会显示旧图像。为了避免撕裂，许多渲染引擎会等待显示器的垂直消隐间隔后再交换缓冲区。

老式 CRT 显示器和电视通过从左到右、从上到下扫描的电子束激发屏幕上的荧光粉，从而“绘制”内存帧缓冲区的内容。在这种显示器上，垂直消隐间隔是指电子枪重置到屏幕左上角时“消隐”（关闭）的时间。现代 LCD、等离子和 LED 显示器不再使用电子束，它们在完成一帧的最后一条扫描线和下一帧的第一条扫描线的绘制之间不需要任何时间。但垂直消隐间隔仍然存在，部分原因是视频标准是在 CRT 成为主流时建立的，部分原因是需要支持老式显示器。

等待垂直消隐间隔的过程称为垂直同步。它实际上只是帧率控制的另一种形式，因为它有效地将主游戏循环的帧率限制为屏幕刷新率的倍数。例如，在刷新率为 60 Hz 的 NTSC 显示器上，游戏的实际更新率

also looks better, and as we'll see in the next section, it can be used to avoid the *tearing* that can occur when the video buffer is updated at a rate that doesn't match the refresh rate of the monitor (see Section 8.5.2.5).

In addition, when elapsed frame times are consistent, features like *record and playback* become a lot more reliable. As its name implies, the record-and-playback feature allows a player's gameplay experience to be recorded and later played back in exactly the same way. This can be a fun game feature, and it's also a valuable testing and debugging tool. For example, difficult-to-find bugs can be reproduced by simply playing back a recorded game that demonstrates the bug.

To implement record and playback, we make note of every relevant event that occurs during gameplay, saving each one in a list along with an accurate time stamp. The list of events can then be replayed with exactly the same timing, using the same initial conditions and an identical initial random seed. In theory, doing this should produce a gameplay experience that is indistinguishable from the original playthrough. However, if the frame rate isn't consistent, things may not happen in exactly the same order. This can cause "drift," and pretty soon your AI characters are flanking when they should have fallen back.

8.5.2.5 Screen Tearing and V-Sync

A visual anomaly known as *screen tearing* occurs when the back buffer is swapped with the front buffer while the screen has only been partially "drawn" by the video hardware. When tearing occurs, a portion of the screen shows the new image, while the remainder shows the old one. To avoid tearing, many rendering engines wait for the *vertical blanking interval* of the monitor before swapping buffers.

Older CRT monitors and TVs "draw" the contents of the in-memory frame buffer by exciting phosphors on the screen via a beam of electrons that scans from left-to-right and top-to-bottom. On such displays, the v-blank interval is the time during which the electron gun is "blanked" (turned off) while it is being reset to the top-left corner of the screen. Modern LCD, plasma and LED displays no longer use an electron beam, and they don't require any time between finishing the draw of the last scan line of one frame and the first scan line of the next. But the v-blank interval still exists, in part because video standards were established when CRTs were the norm, and in part because of the need to support older displays.

Waiting for the v-blank interval is called *v-sync*. It is really just another form of frame-rate governing, because it effectively clamps the frame rate of the main game loop to a multiple of the screen's refresh rate. For example, on an NTSC monitor that refreshes at a rate of 60 Hz, the game's real update rate

有效量化为 1/60 秒的倍数。如果帧之间间隔超过 1/60 秒，则必须等到下一个垂直消隐间隔，这意味着要等待 2/60 秒（30 FPS）。如果错过两个垂直消隐，则必须等待总共 3/60 秒（20 FPS），依此类推。此外，即使游戏的帧速率与垂直消隐间隔同步，也不要对其做出任何假设；如果您的游戏支持这些标准，请务必记住，PAL 和 SECAM 标准基于 50 Hz 的更新速率，而不是 60 Hz。

8.5.3 使用高精度定时器测量实时

我们已经讨论了很多关于测量每帧实际经过的“挂钟”时间的方法。在本节中，我们将详细探讨如何进行此类时间测量。

大多数操作系统都提供了查询系统时间的函数，例如 C 标准库函数 time()。然而，这类函数并不适合测量实时游戏中的耗时，因为它们的精度不够。例如，time() 返回一个整数，表示自 1970 年 1 月 1 日午夜以来的秒数，因此它的精度为一秒——考虑到一帧的执行时间仅为几十毫秒，这个精度实在太过粗糙。

所有现代 CPU 都包含一个高分辨率计时器，它通常以硬件寄存器的形式实现，用于计算自上次启动或重置处理器以来经过的 CPU 周期数（或其倍数）。这是我们在测量游戏中经过的时间时应该使用的计时器，因为它的分辨率通常与几个 CPU 周期的持续时间相当。例如，在 3 GHz 奔腾处理器上，高分辨率计时器每个 CPU 周期增加一次，或每秒 30 亿次。因此，高分辨率计时器的分辨率为 1/3 亿 = 3.33 × 10 − 10 秒 = 0.333 纳秒（三分之一纳秒）。对于我们在游戏中所有的时间测量需求来说，这个分□□辨率已经足够了。

不同的微处理器和操作系统提供了不同的查询高精度计时器的方法。在奔腾处理器上，可以使用名为 rdtsc（读取时间戳计数器）的特殊指令，尽管 Win32 API 将此功能封装在两个函数中：QueryPerformanceCounter() 读取 64 位计数器寄存器，QueryPerformanceFrequency() 返回当前 CPU 每秒计数器增量的次数。在 PowerPC 架构上，例如 Xbox 360 和 PlayStation 3 中的芯片，可以使用指令 mftb（从时基寄存器移动）读取两个 32 位时基寄存器，而在其他 PowerPC 架构上，则使用指令 mfspr（从专用寄存器移动）。

is effectively quantized to a multiple of 1/60 of a second. If more than 1/60 of a second elapses between frames, we must wait until the next v-blank interval, which means waiting 2/60 of a second (30 FPS). If we miss two v-blanks, then we must wait a total of 3/60 of a second (20 FPS) and so on. Also, be careful not to make assumptions about the frame rate of your game, even when it is synchronized to the v-blank interval; if your game supports them, you must keep in mind that the PAL and SECAM standards are based around an update rate of 50 Hz, not 60 Hz.

8.5.3 Measuring Real Time with a High-Resolution Timer

We've talked a lot about measuring the amount of real "wall clock" time that elapses during each frame. In this section, we'll investigate how such timing measurements are made in detail.

Most operating systems provide a function for querying the system time, such as the C standard library function `time()`. However, such functions are not suitable for measuring elapsed times in a real-time game because they do not provide sufficient resolution. For example, `time()` returns an integer representing the number of *seconds* since midnight, January 1, 1970, so its resolution is one second—far too coarse, considering that a frame takes only tens of milliseconds to execute.

All modern CPUs contain a *high-resolution timer*, which is usually implemented as a hardware register that counts the number of CPU cycles (or some multiple thereof) that have elapsed since the last time the processor was powered on or reset. This is the timer that we should use when measuring elapsed time in a game, because its resolution is usually on the order of the duration of a few CPU cycles. For example, on a 3 GHz Pentium processor, the high-resolution timer increments once per CPU cycle, or 3 billion times per second. Hence the resolution of the high-res timer is $1/3\,\text{billion} = 3.33 \times 10^{-10}$ seconds $= 0.333$ ns (one-third of a nanosecond). This is more than enough resolution for all of our time-measurement needs in a game.

Different microprocessors and different operating systems provide different ways to query the high-resolution timer. On a Pentium, a special instruction called `rdtsc` (read time-stamp counter) can be used, although the Win32 API wraps this facility in a pair of functions: `QueryPerformanceCounter()` reads the 64-bit counter register and `QueryPerformanceFrequency()` returns the number of counter increments per second for the current CPU. On a PowerPC architecture, such as the chips found in the Xbox 360 and PlayStation 3, the instruction `mftb` (move from time base register) can be used to read the two 32-bit time base registers, while on other PowerPC architectures, the instruction `mfspr` (move from special-purpose register) is used instead.

大多数处理器的 CPU 高分辨率定时器寄存器都是 64 位宽，以确保不会频繁回绕。64 位无符号整数的最大可能值为 0xFFFFFFFFFFFFFFFF ≈ 1.8 × 10^19 个时钟周期。因此，对于一个 3 GHz 奔腾处理器，如果每个 CPU 周期更新一次高分辨率定时器，则该寄存器的值大约每 195 年就会回绕一次——这绝对不值得我们为此担心。相比之下，32 位整数时钟在 3 GHz 频率下仅需大约 1.4 秒就会回绕一次。

8.5.3.1 高分辨率时钟漂移

请注意，即使通过高精度计时器进行计时测量，在某些情况下也可能不准确。例如，在某些多核处理器上，每个核心上的高精度计时器是独立的，它们可能会（并且确实）出现偏差。如果您尝试比较在不同核心上获取的绝对计时器读数，可能会得到一些奇怪的结果，甚至是负的时间增量。请务必留意此类问题。

8.5.4 时间单位和时钟变量

每当我们在游戏中测量或指定时间长度时，我们有两个选择：

1. 应该使用什么时间单位？我们想用秒、毫秒、机器周期……还是其他单位来存储时间？

2. 应该使用什么数据类型来存储时间测量值？我们应该使用 64 位整数、32 位整数还是 32 位浮点变量？

这些问题的答案取决于特定测量的预期目的。这又引出了两个问题：我们需要多高的精度？我们期望能够表示什么范围的量级？

8.5.4.1 64位整数时钟

我们已经看到，以机器周期为单位的 64 位无符号整数时钟不仅支持极高的精度（在 3 GHz CPU 上，单个周期的持续时间为 0.333 纳秒），而且幅度范围也很广（在 3 GHz CPU 上，64 位时钟大约每 195 年回绕一次）。因此，假设您能够负担得起 64 位的存储空间，那么这是最灵活的时间表示方式。

A CPU's high-resolution timer register is 64 bits wide on most processors, to ensure that it won't wrap too often. The largest possible value of a 64-bit unsigned integer is 0xFFFFFFFFFFFFFFFF $\approx 1.8 \times 10^{19}$ clock ticks. So, on a 3 GHz Pentium processor that updates its high-res timer once per CPU cycle, the register's value will wrap back to zero once every 195 years or so—definitely not a situation we need to lose too much sleep over. In contrast, a 32-bit integer clock will wrap after only about 1.4 seconds at 3 GHz.

8.5.3.1 High-Resolution Clock Drift

Be aware that even timing measurements taken via a high-resolution timer can be inaccurate in certain circumstances. For example, on some multi-core processors, the high-resolution timers are independent on each core, and they can (and do) drift apart. If you try to compare absolute timer readings taken on different cores to one another, you might end up with some strange results—even negative time deltas. Be sure to keep an eye out for these kinds of problems.

8.5.4 Time Units and Clock Variables

Whenever we measure or specify time durations in a game, we have two choices to make:

1. What *time units* should be used? Do we want to store our times in seconds, or milliseconds, or machine cycles…or in some other unit?

2. What *data type* should be used to store time measurements? Should we employ a 64-bit integer, or a 32-bit integer, or a 32-bit floating point variable?

The answers to these questions depend on the intended purpose of a given measurement. This gives rise to two more questions: How much precision do we need? And what range of magnitudes do we expect to be able to represent?

8.5.4.1 64-Bit Integer Clocks

We've already seen that a 64-bit unsigned integer clock, measured in machine cycles, supports both an extremely high precision (a single cycle is 0.333 ns in duration on a 3 GHz CPU) and a broad range of magnitudes (a 64-bit clock wraps once roughly every 195 years at 3 GHz). So this is the most flexible time representation, presuming you can afford 64 bits worth of storage.

8.5.4.2 32位整数时钟

当需要高精度测量相对较短的持续时间时，我们可以使用 32 位整数时钟，以机器周期为单位。例如，为了分析一段代码的性能，我们可以这样做：

```
// Grab a time snapshot.
U64 begin_ticks = readHiResTimer();

// This is the block of code whose performance we wish
// to measure.
doSomething();
doSomethingElse();
nowReallyDoSomething();

// Measure the duration.
U64 end_ticks = readHiResTimer();
U32 dt_ticks = static_cast<U32>(end_ticks - begin_ticks);

// Now use or cache the value of dt_ticks...
```

请注意，我们仍然将原始时间测量值存储在 64 位整数变量中。只有时间增量 dt 存储在 32 位变量中。这可以避免在 32 位边界处回绕的潜在问题。例如，如果 begin_ticks = 0x12345678FFFFFFB7 且 end_ticks = 0x1234567 9 00000039 ，那么如果我们在减法之前将各个时间测量值截断为 32 位，则会测得负的时间增量。

8.5.4.3 32 位浮点时钟

另一种常见的方法是将相对较小的时间增量以浮点格式存储，以秒为单位。为此，我们只需将以 CPU 周期为单位的持续时间乘以 CPU 的时钟频率（以每秒周期数为单位）。例如：

```
// Start off assuming an ideal frame time (30 FPS).
F32 dt_seconds = 1.0f / 30.0f;

// Prime the pump by reading the current time.
U64 begin_ticks = readHiResTimer();

while (true) // main game loop
{
    runOneIterationOfGameLoop(dt_seconds);

    // Read the current time again, and calculate the delta.
```


8.5.4.2 32-Bit Integer Clocks

When measuring relatively short durations with high precision, we can turn to a 32-bit integer clock, measured in machine cycles. For eample, to profile the performance of a block of code, we might do something like this:

```
// Grab a time snapshot.
U64 begin_ticks = readHiResTimer();

// This is the block of code whose performance we wish
// to measure.
doSomething();
doSomethingElse();
nowReallyDoSomething();

// Measure the duration.
U64 end_ticks = readHiResTimer();
U32 dt_ticks = static_cast<U32>(end_ticks - begin_ticks);

// Now use or cache the value of dt_ticks...
```

Notice that we still store the raw time measurements in 64-bit integer variables. Only the time delta `dt` is stored in a 32-bit variable. This circumvents potential problems with wrapping at the 32-bit boundary. For example, if `begin_ticks = 0x12345678FFFFFFB7` and `end_ticks = 0x12345679 00000039`, then we would measure a negative time delta if we were to truncate the individual time measurements to 32 bits each prior to subtracting them.

8.5.4.3 32-Bit Floating-Point Clocks

Another common approach is to store relatively small time deltas in floating-point format, measured in units of seconds. To do this, we simply multiply a duration measured in CPU cycles by the CPU's clock frequency, which is in cycles per second. For example:

```
// Start off assuming an ideal frame time (30 FPS).
F32 dt_seconds = 1.0f / 30.0f;

// Prime the pump by reading the current time.
U64 begin_ticks = readHiResTimer();

while (true) // main game loop
{
    runOneIterationOfGameLoop(dt_seconds);

    // Read the current time again, and calculate the delta.
```

```
        U64 end_ticks = readHiResTimer();

        // Check our units: seconds = ticks / (ticks/second)
        dt_seconds = (F32)(end_ticks - begin_ticks)
                   / (F32)getHiResTimerFrequency();

        // Use end_ticks as the new begin_ticks for next frame.
        begin_ticks = end_ticks;
    }
```

再次注意，在将两个 64 位时间测量值转换为浮点格式之前，我们必须小心地将它们相减。这确保我们不会将过大的数值存储到 32 位浮点变量中。

8.5.4.4 浮点时钟的局限性

回想一下，在 32 位 IEEE 浮点数中，尾数的 23 位通过指数动态地分布在整数部分和小数部分之间（参见第 3.3.1.4 节）。较小的幅度只需要几位，为小数部分留下了足够的精度位。但是，一旦时钟的幅度变得过大，其整数部分就会占用更多位，为小数部分留下更少的位。最终，即使是整数部分的最低有效位也会隐式变为零。这意味着在浮点时钟变量中存储长持续时间时必须谨慎。如果我们跟踪自游戏开始以来经过的时间量，浮点时钟最终将变得不准确，以至于无法使用。

浮点时钟通常仅用于存储相对较短的时间增量，最多测量几分钟，通常仅测量一帧或更短。如果在游戏中使用绝对值浮点时钟，则需要定期将时钟重置为零，以避免累积较大的幅度。

8.5.4.5 其他时间单位

某些游戏引擎允许以游戏定义的单位指定时间值，该单位足够精细，允许使用整数格式（而不是浮点格式），足够精确，可以用于引擎内的广泛应用，同时又足够大，以至于 32 位时钟不会过于频繁地回绕。一个常见的选择是 1/300 秒的时间单位。这种做法效果很好，因为 (a) 它足够精细，可以用于许多用途，(b) 它每 165.7 天才回绕一次，并且 (c) 它是 NTSC 和 PAL 刷新率的偶数倍。60 FPS 帧的持续时间为 5 个这样的单位，而 50 FPS 帧的持续时间为 6 个单位。


```
        U64 end_ticks = readHiResTimer();

        // Check our units: seconds = ticks / (ticks/second)
        dt_seconds = (F32)(end_ticks - begin_ticks)
                   / (F32)getHiResTimerFrequency();

        // Use end_ticks as the new begin_ticks for next frame.
        begin_ticks = end_ticks;
    }
```

Notice once again that we must be careful to subtract the two 64-bit time measurements *before* converting them into floating-point format. This ensures that we don't store too large a magnitude into a 32-bit floating-point variable.

8.5.4.4 Limitations of Floating-Point Clocks

Recall that in a 32-bit IEEE float, the 23 bits of the mantissa are dynamically distributed between the whole and fractional parts of the value, by way of the exponent (see Section 3.3.1.4). Small magnitudes require only a few bits, leaving plenty of bits of precision for the fraction. But once the magnitude of our clock grows too large, its whole part eats up more bits, leaving fewer bits for the fraction. Eventually, even the least-significant bits of the whole part become implicit zeros. This means that we must be cautious when storing *long durations* in a floating-point clock variable. If we keep track of the amount of time that has elapsed since the game was started, a floating-point clock will eventually become inaccurate to the point of being unusable.

Floating-point clocks are usually only used to store relatively short time deltas, measuring at most a few minutes and more often just a single frame or less. If an absolute-valued floating-point clock is used in a game, you will need to reset the clock to zero periodically to avoid accumulation of large magnitudes.

8.5.4.5 Other Time Units

Some game engines allow timing values to be specified in a game-defined unit that is fine-grained enough to permit an integer format to be used (as opposed to requiring a floating-point format), precise enough to be useful for a wide range of applications within the engine, and yet large enough that a 32-bit clock won't wrap too often. One common choice is a 1/300 second time unit. This works well because (a) it is fine-grained enough for many purposes, (b) it only wraps once every 165.7 days and (c) it is an even multiple of both NTSC and PAL refresh rates. A 60 FPS frame would be 5 such units in duration, while a 50 FPS frame would be 6 units in duration.

显然，1/300 秒的时间单位不够精确，无法处理诸如动画时间缩放之类的细微效果。（如果我们试图将 30 FPS 的动画速度减慢到其常规速度的十分之一以下，那就超出精度了！）因此，在许多情况下，最好仍然使用浮点时间单位或机器周期。但 1/300 秒的时间单位可以有效地用于诸如指定自动武器两次射击之间的时间间隔、AI 控制角色在开始巡逻前应等待多长时间，或者玩家站在酸液池中时可以存活的时间等。

8.5.5 处理断点

当游戏遇到断点时，其循环会停止运行，调试器会接管。但是，如果您的游戏和调试器在同一台计算机上运行，□□则 CPU 会继续运行，实时时钟也会继续累积周期。当您在断点处检查代码时，可能会经过大量的挂钟时间。当您允许程序继续运行时，这可能会导致测量到的帧时间长达数秒，甚至数分钟或数小时！

显然，如果我们允许如此巨大的时间差传递给引擎中的子系统，就会发生糟糕的事情。如果幸运的话，游戏在一帧中向前移动数秒后，可能仍能正常运行。更糟糕的是，游戏可能会崩溃。

一个简单的方法可以解决这个问题。在游戏主循环中，如果我们测量到的帧时间超过了某个预定义的上限（例如 1 秒），我们可以假设游戏刚从断点处恢复执行，并将时间增量人为地设置为 1/30 或 1/60 秒（或任何目标帧率）。实际上，游戏会锁定一帧，以避免测量到的帧时长出现大幅飙升。

```
// Start off assuming the ideal dt (30 FPS).
F32 dt = 1.0f / 30.0f;

// Prime the pump by reading the current time.
U64 begin_ticks = readHiResTimer();

while (true) // main game loop
{
    updateSubsystemA(dt);
    updateSubsystemB(dt);
    // ...
    renderScene();
    swapBuffers();
```


Obviously a 1/300 second time unit is not precise enough to handle subtle effects, like time-scaling an animation. (If we tried to slow a 30 FPS animation down to less than 1/10 of its regular speed, we'd be out of precision!) So for many purposes, it's still best to use floating-point time units or machine cycles. But a 1/300 second time unit can be used effectively for things like specifying how much time should elapse between the shots of an automatic weapon, or how long an AI-controlled character should wait before starting his patrol, or the amount of time the player can survive when standing in a pool of acid.

8.5.5 Dealing with Breakpoints

When your game hits a breakpoint, its loop stops running and the debugger takes over. However, if your game is running on the same computer that your debugger is running on, then the CPU continues to run, and the real-time clock continues to accrue cycles. A large amount of wall clock time can pass while you are inspecting your code at a breakpoint. When you allow the program to continue, this can lead to a measured frame time many seconds, or even minutes or hours in duration!

Clearly if we allow such a huge delta time to be passed to the subsystems in our engine, bad things will happen. If we are lucky, the game might continue to function properly after lurching forward many seconds in a single frame. Worse, the game might just crash.

A simple approach can be used to get around this problem. In the main game loop, if we ever measure a frame time in excess of some predefined upper limit (e.g., 1 second), we can assume that we have just resumed execution after a breakpoint, and we set the delta-time artificially to 1/30 or 1/60 of a second (or whatever the target frame rate is). In effect, the game becomes frame-locked for one frame, in order to avoid a massive spike in the measured frame duration.

```
// Start off assuming the ideal dt (30 FPS).
F32 dt = 1.0f / 30.0f;

// Prime the pump by reading the current time.
U64 begin_ticks = readHiResTimer();

while (true) // main game loop
{
    updateSubsystemA(dt);
    updateSubsystemB(dt);
    // ...
    renderScene();
    swapBuffers();
```

```
    // Read the current time again, and calculate an
    // estimate of next frame's delta time.
    U64 end_ticks = readHiResTimer();

    dt = (F32)(end_ticks - begin_ticks)
       / (F32)getHiResTimerFrequency();

    // If dt is too large, we must have resumed from a
    // breakpoint -- frame-lock to the target rate this
    // frame.
    if (dt > 1.0f)
    {
        dt = 1.0f/30.0f;
    }

    // Use end_ticks as the new begin_ticks for next frame.
    begin_ticks = end_ticks;
}
```

8.6 多处理器游戏循环

在第四章中，我们探索了如今在消费级计算机、移动设备和游戏机中无处不在的并行计算硬件，并学习了如何编写能够利用这些并行计算资源的并发软件。在本节中，我们将讨论将这些知识应用于游戏引擎的游戏循环的各种方法。

8.6.1 任务分解

为了充分利用并行计算硬件，我们需要将游戏循环每次迭代中执行的各种任务分解为多个子任务，每个子任务都可以并行执行。这种分解操作将我们的软件从顺序程序转换为并发程序。

分解软件系统以实现并发的方法有很多种，但正如我们在第 4.1.3 节中讨论的那样，我们可以将它们大致分为两类：任务并行和数据并行。

任务并行非常适合需要完成多项不同任务的情况，我们选择在多个核心上并行执行这些任务。例如，我们可能会尝试在游戏循环的每次迭代中并行执行动画混合和碰撞检测，或者提交


```
    // Read the current time again, and calculate an
    // estimate of next frame's delta time.
    U64 end_ticks = readHiResTimer();

    dt = (F32)(end_ticks - begin_ticks)
       / (F32)getHiResTimerFrequency();

    // If dt is too large, we must have resumed from a
    // breakpoint -- frame-lock to the target rate this
    // frame.
    if (dt > 1.0f)
    {
        dt = 1.0f/30.0f;
    }

    // Use end_ticks as the new begin_ticks for next frame.
    begin_ticks = end_ticks;
}
```

8.6 Multiprocessor Game Loops

In Chapter 4, we explored the parallel computing hardware that is now ubiquitous in consumer-grade computers, mobile devices and game consoles, and we learned the mechanics of how to write *concurrent* software that takes advantage of these parallel computing resources. In this section, we'll discuss various ways in which this knowledge can be applied to a game engine's *game loop*.

8.6.1 Task Decomposition

To take advantage of parallel computing hardware, we need to *decompose* the various tasks that are performed during each iteration of our game loop into multiple subtasks, each of which can be executed in parallel. This act of decomposition transforms our software from a *sequential* program into a *concurrent* one.

There are many ways to decompose a software system for concurrency, but as we discussed in Section 4.1.3, we can place all of them roughly into one of two categories: *task parallelism* and *data parallelism*.

Task parallelism naturally fits situations in which a number of *different* things need to be done, and we opt to do them in parallel across multiple cores. For example, we might attempt to perform animation blending in parallel with collision detection during each iteration of our game loop, or we might submit

将原语发送到 GPU 以渲染第 N 帧，同时开始更新第 N + 1 帧的游戏世界状态。

数据并行最适合需要对大量数据元素重复执行单次计算的情况。GPU 可能是数据并行的最佳示例——GPU 通过将工作分配到大量并行运行的处理核心上，每帧执行数百万次逐像素和逐顶点计算。然而，正如我们将在以下章节中看到的，数据并行不仅仅存在于 GPU 上——在游戏循环中，CPU 执行的大量任务也可以从数据并行中受益。

在接下来的章节中，我们将探讨几种细分游戏循环工作的不同方法，其中一些方法采用任务并行，另一些则依赖于数据并行。我们将探讨每种方法的优缺点，然后了解通用作业系统如何成为一种实用工具，将几乎任何工作负载转换为可利用硬件并行性的并发操作。

8.6.2 每个子系统一个线程

为了实现并发，分解游戏循环的一个简单方法是将特定的引擎子系统分配到单独的线程中运行。例如，渲染引擎、碰撞和物理模拟、动画管道以及音频引擎可以分别分配到各自的线程中。主线程将控制和同步这些次级子系统线程的操作，并继续处理游戏高级逻辑（主游戏循环）的大部分内容。在具有多个物理 CPU 的硬件平台上，这种设计将允许线程化的引擎子系统彼此并行执行，并与主游戏循环并行执行。这是一个任务并行的简单示例，如图 8.4 所示。

为每个引擎子系统分配单独的线程这种简单的方法存在许多问题。首先，引擎子系统的数量可能与我们游戏平台上的核心数量不匹配。结果，我们最终的线程数可能会超过核心数，一些子系统线程将需要通过时间分片共享核心。

另一个问题是，每个引擎子系统每帧所需的处理量不同。这意味着，虽然某些线程（及其对应的 CPU 核心）每帧都利用率很高，但其他线程可能会在帧的大部分时间内处于空闲状态。

另一个问题是，一些引擎子系统依赖于其他子系统生成的数据。例如，渲染和音频子系统无法

primitives to the GPU for rendering of frame N in parallel with starting to update the state of the game world for frame $N + 1$.

Data parallelism is best suited to situations in which a *single* computation needs to be performed repeatedly on a large number of data elements. The GPU is probably the best example of data parallelism in action—a GPU performs millions of per-pixel and per-vertex calculations every frame by distributing the work across a large number of processing cores that operate in parallel. However, as we'll see in the following sections, data parallelism isn't only found on the GPU—there are plenty of tasks performed by the CPU during the game loop that can benefit from data parallelism as well.

In the following sections, we'll take a look at a number of different ways to subdivide the work that's done by the game loop, some of which employ a task-parallel approach, others of which rely on data parallelism. We'll explore the pros and cons of each approach, and then see how a general-purpose *job system* can be a useful tool for converting literally any workload into a concurrent operation that can take advantage of hardware parallelism.

8.6.2 One Thread per Subsystem

One simple way to decompose our game loop for concurrency would be to assign particular engine subsystems to run in separate threads. For example, the rendering engine, the collision and physics simulation, the animation pipeline, and the audio engine could each be assigned to its own thread. A master thread would control and synchronize the operations of these secondary subsystem threads, and would also continue to handle the lion's share of the game's high-level logic (the main game loop). On a hardware platform with multiple physical CPUs, this design would allow the threaded engine subsystems to execute in parallel with one another and with the main game loop. This is a simple example of *task parallelism;* it is depicted in Figure 8.4.

There are a number of problems with the simple approach of assigning each engine subsystem to its own thread. For one thing, the number of engine subsystems probably won't match the number of cores on our gaming platform. As a result, we'll probably end up with more threads than we have cores, and some subsystem threads will need to share a core via time-slicing.

Another problem is that each engine subsystem requires a different amount of processing each frame. This means that while some threads (and their corresponding CPU cores) are highly utilized each frame, others may sit idle for large portions of the frame.

Yet another issue is that some engine subsystems *depend* on the data produced by others. For example, the rendering and audio subsystems cannot

图 8.4. 每个主引擎子系统一个线程。

开始执行第 N 帧的工作，直到动画、碰撞和物理系统完成第 N 帧的工作。如果两个子系统像这样相互依赖，我们就无法并行运行它们。

由于所有这些问题，尝试为每个引擎子系统分配一个单独的线程实际上并不实用。我们可以做得更好。

8.6.3 分散/聚集

在游戏循环的单次迭代中执行的许多任务都是数据密集型的。例如，我们可能需要处理大量的射线投射请求，混合大量的动画姿势，或者计算大型交互式场景中每个对象的世界空间矩阵。利用并行计算硬件执行此类任务的一种方法是采用分治法。与其尝试在单个 CPU 核心上一次处理 9000 条射线投射，不如将工作分成 6 个批次，每个批次 1500 条射线投射，然后在每个批次上执行一个批次。

Figure 8.4. One thread per major engine subsystem.

start doing their work for frame N until the animation, collision and physics systems have completed their work for frame N. We cannot run two subsystems in parallel if they depend on one another like this.

As a result of all of these problems, attempting to assign each engine subsystem to its own thread is really not a practical design. We can do better.

8.6.3 Scatter/Gather

Many of the tasks performed during a single iteration of the game loop are data-intensive. For example, we may need to process a large number of ray cast requests, blend together a large batch of animation poses, or calculate world-space matrices for every object in a massive interactive scene. One way to take advantage of parallel computing hardware to perform these kinds of tasks is to use a divide-and-conquer approach. Instead of attempting to process 9000 ray casts one at a time on a single CPU core, we can divide the work into, say, six batches of 1500 ray casts each, and then execute one batch on each

图 8.5.分散/聚集用于并行化游戏循环中选定的 CPU 密集型部分。

PS4 或 Xbox One 上的六个 CPU 核心之一。这种方法是一种数据并行形式。

在分布式系统术语中，这被称为分散/聚集方法，因为一个工作单元被分成更小的子单元，分布到多个处理核心上执行（分散），然后在所有工作负载完成后以适当的方式组合或最终确定结果（聚集）。

8.6.3.1 游戏循环中的分散/聚集

在我们的游戏循环中，“主”游戏循环线程可能会在游戏循环的一次迭代中的不同时间执行一个或多个分散/聚集操作。该架构如图 8.5 所示。

给定一个包含 N 个需要处理的数据项的数据集，主线程会将工作分成 m 个批次，每个批次大约包含 N / m 个元素。（m 的值可能取决于系统中可用核心的数量，但如果我们希望留出一些核心用于其他工作，则情况可能并非如此。）然后，主线程会创建 m 个工作线程，并为每个线程提供起始索引和数量。

1 PS4 和 Xbox One 都允许开发者使用第七个核心的部分处理能力。由于操作系统也占用了该核心，因此开发者无法充分利用其全部带宽。这些 8 核机器中的第八个核心是完全禁止使用的。这样做是为了应对 CPU 制造过程中不可避免地会出现一些故障核心的情况。

Figure 8.5. Scatter/gather used to parallelize selected CPU-intensive parts of the game loop.

of the six[1] CPU cores on our PS4 or Xbox One. This approach is a form of *data parallelism*.

In distributed systems terminology, this is known as a *scatter/gather* approach, because a unit of work is divided into smaller subunits, distributed onto multiple processing cores for execution (scatter), and then the results are combined or finalized in an appropriate manner once all workloads have been completed (gather).

8.6.3.1 Scatter/Gather in the Game Loop

In the context of our game loop, one or more scatter/gather operations might be performed, at various times during one iteration of the game loop, by the "master" game loop thread. This architecture is illustrated in Figure 8.5.

Given a dataset containing N data items that require processing, the master thread would divide the work into m batches, each containing roughly N/m elements. (The value of m would probably be determined based on the number of available cores in the system, although this may not be the case if, for example, we wish to leave some cores free for other work.) The master thread would then spawn m worker threads and provide each with a start index and count,

[1] Both the PS4 and the Xbox One allow developers to access some of the processing power of a seventh core. Not all of its bandwidth is available to developers because this core is also utilized by the operating system. The eighth core in these 8-core machines is entirely off-limits. This is done to allow for the fact that some faulty cores are inevitably produced during any CPU's fabrication process.

允许它处理分配给它的数据子集。每个工作线程可能会就地更新数据项，或者（通常更好）它可能将输出数据生成到单独的预分配缓冲区（每个工作线程一个）。

成功分散工作负载后，主线程就可以自由地执行一些其他有用的工作，同时等待工作线程完成其任务。

在帧的稍后某个时间点，主线程会等待所有工作线程终止，从而收集结果，这可能需要使用诸如 pthread_join() 之类的函数。如果所有工作线程都已退出，此函数将立即返回；但是，如果仍有任何工作线程在运行，此调用将使主线程进入睡眠状态。

收集步骤完成后，主线程可能会以任何合适的方式合并结果。例如，将动画混合在一起后，下一步可能是计算蒙皮矩阵——只有在所有动画混合线程都完成工作后才会启动此步骤。我们在 4.4.6 节中讨论线程创建和连接时提供了一个非常类似的示例。

8.6.3.2 用于分散/聚集的 SIMD

在 4.10 节中，我们探讨了循环矢量化，将其作为利用 SIMD 并行性来提升数据密集型工作性能的一种手段。这实际上只是分散/聚集方法的另一种形式，以非常精细的粒度执行。SIMD 可以代替基于线程的分散/聚集，但它很可能与其结合使用（每个工作线程在内部利用矢量化来执行其工作）。

8.6.3.3 提高分散/聚集效率

分散/聚集方法是一种直观的方法，可以将数据密集型工作分布到多个核心。然而，正如我们上面所述，这种并行化方法确实存在一个大问题——创建线程的开销很大。创建线程需要内核调用，将主线程与其工作线程连接起来也需要内核调用。每当线程启动和关闭时，内核本身都会进行大量的设置和拆卸工作。因此，每次执行分散/聚集操作时都创建一堆线程是不切实际的。

我们可以通过使用预先生成的线程池来降低创建线程的成本。某些操作系统（例如 Windows）提供了用于创建和管理线程池的 API。（例如，请参阅 https://bit.ly/2H8ChIp。）如果没有这样的 API，您可以自行实现一个简单的线程池，使用条件变量、信号量和原子布尔值。

allowing it to process its assigned subset of the data. Each worker thread might update the data items in-place, or (usually better) it might produce output data into a separate preallocated buffer (one per worker thread).

Having successfully scattered the workload, the master thread would then be free to perform some other useful work while it waits for the worker threads to complete their tasks.

At some point later in the frame, the master thread would gather the results by waiting until all of the worker threads have terminated, perhaps using a function such as `pthread_join()`. If the worker threads have all exited, this function will return immediately; however, if any worker threads are still running, this call would have the effect of putting the master thread to sleep.

Once the gather step has been completed, the master thread could potentially combine the results in whatever manner is appropriate. For example, after blending animations together, the next step might be to calculate skinning matrices—this step would be kicked off only after all animation blending threads have completed their work. We presented an example very much like this in Section 4.4.6 when we covered thread creation and joining.

8.6.3.2 SIMD for Scatter/Gather

In Section 4.10, we explored loop *vectorization* as a means of leveraging SIMD parallelism to improve the performance of data-intensive work. This is really just another form of the scatter/gather approach, performed at a very fine level of granularity. SIMD could be used in lieu of thread-based scatter/gather, but it would likely be used in conjunction with it (with each worker thread utilizing vectorization internally to perform its work).

8.6.3.3 Making Scatter/Gather More Efficient

The scatter/gather approach is an intuitive way to distribute data-intensive work across multiple cores. However, as we've described it above, this parallelization method does suffer from one big problem—spawning threads is expensive. Spawning a thread involves a kernel call, as does joining the master thread with its workers. The kernel itself does quite a lot of set-up and tear-down work whenever threads come and go. So spawning a bunch of threads every time we want to perform a scatter/gather is impractical.

We could mitigate the costs of spawning threads by using of a pool of pre-spawned threads. Some operating systems like Windows provide an API for creating and managing a pool of threads. (See for example https://bit.ly/2H8ChIp.) In the absence of such an API, you can always implement a simple thread pool yourself, using *condition variables*, *semaphores*, atomic Boolean

变量或其他机制来同步线程的活动。

我们希望线程池能够在每一帧的运行过程中执行各种各样的分散/聚集操作。这意味着我们不能再简单地为每个分散/聚集操作创建一堆线程，而这些线程的入口函数只执行一项特定的计算。相反，线程池中的每个线程都必须能够执行我们在游戏循环的任何迭代过程中可能想要执行的任何分散/聚集操作。我们可以设想使用一个庞大的 switch 语句来实现这一点，但这个想法听起来笨重、丑陋且难以维护。实际上，我们想要的是一个通用的系统，用于在目标硬件的可用核心上并发执行工作单元。

8.6.4 工作系统

作业系统是一种通用系统，用于在多个核心上执行任意工作单元（通常称为作业）。借助作业系统，游戏程序员可以将游戏循环的每次迭代细分为相对大量的独立作业，并将它们提交给作业系统执行。作业系统维护一个已提交作业的队列，并在可用核心之间调度这些作业，方法是将它们提交给线程池中的工作线程执行，或者通过其他方式。从某种意义上说，作业系统就像一个自定义的轻量级操作系统内核，只不过它不是调度线程在可用核心上运行，而是调度作业。

作业可以任意细粒度地划分，并且在实际的游戏引擎中，许多作业彼此独立。如图 8.6 所示，这些特性有助于最大限度地提高处理器利用率。该架构还可以自然地扩展或缩减，以适应具有任意数量 CPU 核心的硬件。

8.6.4.1 典型的作业系统界面

典型的作业系统提供简单易用的 API，其 API 与线程库的 API 非常相似。它包含一个用于创建作业的函数（相当于 pthread_create()，通常称为“踢出作业”），一个允许一个作业等待一个或多个其他作业终止的函数（相当于 pthread_join()），以及一个允许作业“提前”自行终止的方法（在从其入口点函数返回之前）。作业系统还必须提供某种类型的自旋锁或互斥锁，以便以原子方式执行关键的并发操作。它还可以提供通过条件变量或类似机制使作业进入睡眠状态并唤醒的功能。

variables, or some other mechanism in order to synchronize the activities of the threads.

We'd like our thread pool to be capable of performing a wide variety of scatter/gather operations over the course of each frame. This means that we can no longer simply spawn a bunch of threads for each scatter/gather, whose entry point function does one particular computation. Instead, each thread in the pool would have to be capable of performing any of the scatter/gather operations we might want to perform during any iteration of our game loop. We could imagine using a giant switch statement to implement this, but that idea sounds clunky, ugly and hard to maintain. Really, what we want is a general-purpose system for executing units of work concurrently, across the available cores on our target hardware.

8.6.4 Job Systems

A *job system* is a general-purpose system for executing arbitrary units of work, typically called *jobs*, across multiple cores. With a job system, a game programmer can subdivide each iteration of the game loop into a relatively large number of independent jobs, and submit them to the job system for execution. The job system maintains a queue of submitted jobs, and it schedules those jobs across the available cores, either by submitting them to be executed by worker threads in a thread pool, or by some other means. In a sense, a job system is like a custom, light-weight operating system kernel, except that instead of scheduling threads to run on the available cores, it schedules jobs.

Jobs can be arbitrarily fine-grained, and in a real game engine many are independent of one another. As shown in Figure 8.6, these facts help to maximize processor utilization. This architecture also scales up or down naturally to hardware with any number of CPU cores.

8.6.4.1 Typical Job System Interface

A typical job system provides a simple, easy-to-use API that looks very similar to the API for a threading library. There's a function for spawning a job (the equivalent of `pthread_create()`, often called *kicking* a job), a function that allows one job to wait for one or more other jobs to terminate (the equivalent of `pthread_join()`), and perhaps a way for a job to self-terminate "early" (before returning from its entry point function). A job system must also provide spin locks of mutexes of some kind for performing critical concurrent operations in an *atomic* manner. It may also provide facilities for putting jobs to sleep and waking them back up, via condition variables or some similar mechanism.

图 8.6。在作业模型中，工作被分解成细粒度的块，可以由任何可用的处理器处理。这有助于最大限度地提高处理器利用率，同时提高主游戏循环的灵活性。

要启动一个作业，我们需要告诉作业系统要执行什么作业以及如何执行。通常，这些信息会通过一个小型数据结构传递给 KickJob() 函数，我们在此将其称为作业声明。

作业声明至少必须包含指向其入口函数的指针。能够将任意输入参数传递给作业的入口函数也很重要。这可以通过多种方式实现，但最简单的方法是提供一个 uintptr_t 类型的参数，该参数将在作业实际运行时传递给入口函数。这使我们能够轻松地将简单信息（例如单个布尔值或整数）传递给作业；但是由于指针可以安全地转换为 uintptr_t 类型，因此我们也可以使用此作业参数将指针传递给任意数据结构，该数据结构本身包含作业可能需要的任何输入参数。

作业系统也可能提供优先级机制，就像大多数线程库一样。在这种情况下，优先级也可能包含在作业声明中。

下面是一个简单的作业声明的示例，以及一个简单的作业系统 API：

Figure 8.6. In a job model, work is broken down into fine-grained chunks that can be picked up by any available processor. This can help maximize processor utilization while providing the main game loop with improved flexibility.

To kick a job, we need to tell the job system *what* job to perform, and *how* to perform it. Normally this information is passed to the `KickJob()` function via a small data structure which we'll call a *job declaration* here.

At minimum, a job declaration must contain a pointer to the job's entry point function. It's also important to be able to pass arbitrary input parameters to the job's entry point function. This could be done in various ways, but the simplest is to provide a single parameter of type `uintptr_t`, which will be passed to the entry point function when the job actually runs. This allows us to pass simple information like a single Boolean or integer to the job with little fuss; but because a pointer can be safely cast to a `uintptr_t`, we can also use this job parameter to pass a pointer to an arbitrary data structure, that itself contains whatever input parameters the job might require.

A job system may also provide a priority mechanism, much as most thread libraries do. In this case, a priority might also be included within the job declaration.

Here's an example of a simple job declaration, along with a simple job system API:

```
namespace job
{
   // signature of all job entry points
   typedef void EntryPoint(uintptr_t param);

   // allowable priorities
   enum class Priority
   {
       LOW, NORMAL, HIGH, CRITICAL
   };

   // counter (implementation not shown)
   struct Counter  ... ;
   Counter* AllocCounter();
   void FreeCounter(Counter* pCounter);

   // simple job declaration
   struct Declaration
   {
       EntryPoint*     m_pEntryPoint;
       uintptr_t       m_param;
       Priority        m_priority;
       Counter*        m_pCounter;
   };

   // kick a job
   void KickJob(const Declaration& decl);
   void KickJobs(int count, const Declaration aDecl[]);

   // wait for job to terminate (for its Counter to become zero)
   void WaitForCounter(Counter* pCounter);

   // kick jobs and wait for completion
   void KickJobAndWait(const Declaration& decl);
   void KickJobsAndWait(int count, const Declaration aDecl[]);
}
```

你会注意到这里有一个不透明的类型，叫做 job::Counter 。计数器允许一个作业进入睡眠状态，并等待一个或多个其他作业完成执行。我们将在 8.6.4.6 节讨论作业计数器。

8.6.4.2 基于线程池的简单作业系统

正如我们之前提到的，可以围绕工作线程池构建一个作业系统。一个好主意是为目标机器上的每个 CPU 生成一个线程，并使用关联性将每个线程锁定到一个核心。每个工作线程都处于无限循环中，处理以下作业请求：


```
namespace job
{
   // signature of all job entry points
   typedef void EntryPoint(uintptr_t param);

   // allowable priorities
   enum class Priority
   {
       LOW, NORMAL, HIGH, CRITICAL
   };

   // counter (implementation not shown)
   struct Counter  ... ;
   Counter* AllocCounter();
   void FreeCounter(Counter* pCounter);

   // simple job declaration
   struct Declaration
   {
       EntryPoint*     m_pEntryPoint;
       uintptr_t       m_param;
       Priority        m_priority;
       Counter*        m_pCounter;
   };

   // kick a job
   void KickJob(const Declaration& decl);
   void KickJobs(int count, const Declaration aDecl[]);

   // wait for job to terminate (for its Counter to become zero)
   void WaitForCounter(Counter* pCounter);

   // kick jobs and wait for completion
   void KickJobAndWait(const Declaration& decl);
   void KickJobsAndWait(int count, const Declaration aDecl[]);
}
```

You'll notice there's an opaque type here called `job::Counter`. Counters allow one job to go to sleep and wait until one or more other jobs have completed executing. We'll discuss job counters in Section 8.6.4.6.

8.6.4.2 A Simple Job System Based on a Thread Pool

As we hinted at earlier, it's possible to build a job system around a pool of worker threads. It's a good idea to spawn one thread for each CPU that's present in the target machine, and use affinity to lock each thread to one core. Each worker thread sits in an infinite loop processing job requests that are

其他线程（和/或其他作业）为其提供任务。在这个无限循环的顶部，工作线程进入睡眠状态（可能通过条件变量），等待作业请求可用。收到请求后，工作线程被唤醒，调用作业的入口函数，并将作业声明中的输入参数传递给它。入口函数返回时，表示工作已完成，因此工作线程将回到无限循环的顶部执行更多作业。如果没有可用的作业，它将再次进入睡眠状态，等待下一个作业请求。

作业工作线程的底层实现如下。请注意，这并非完整的实现，仅供参考。

```
namespace job
{
    void* JobWorkerThread(void*)
    {
        // keep on running jobs forever...
        while (true)
        {
            Declaration declCopy;

            // wait for a job to become available
            pthread_mutex_lock(&g_mutex);
            while (!g_ready)
            {
                pthread_cond_wait(&g_jobCv, &g_mutex);
            }

            // copy the JobDeclaration locally and
            // release our mutex lock
            declCopy = GetNextJobFromQueue();
            pthread_mutex_unlock(&g_mutex);

            // run the job
            declCopy.m_pEntryPoint(declCopy.m_param);

            // job is done! rinse and repeat...
        }
    }
}
```

8.6.4.3 基于线程的作业的局限性

假设我们编写一个作业，用于更新 AI 驱动的非玩家角色 (NPC) 的状态。该作业的入口函数可能如下所示：

provided to it by other threads (and/or other jobs). At the top of this infinite loop, the worker thread goes to sleep (possibly via a condition variable), waiting for a job request to become available. When a request is received, the worker thread wakes up, calls the job's entry point function and passes the input parameter from the job declaration to it. When the entry point function returns, that indicates that the work is complete, so the worker thread goes back to the top of its infinite loop to execute more jobs. If none are available, it goes back to sleep, waiting for another job request to come in.

Here's what a job worker thread might look like under the hood. Please bear in mind that this is not a complete implementation; it's just for illustrative purposes.

```
namespace job
{
    void* JobWorkerThread(void*)
    {
        // keep on running jobs forever...
        while (true)
        {
            Declaration declCopy;

            // wait for a job to become available
            pthread_mutex_lock(&g_mutex);
            while (!g_ready)
            {
                pthread_cond_wait(&g_jobCv, &g_mutex);
            }

            // copy the JobDeclaration locally and
            // release our mutex lock
            declCopy = GetNextJobFromQueue();
            pthread_mutex_unlock(&g_mutex);

            // run the job
            declCopy.m_pEntryPoint(declCopy.m_param);

            // job is done! rinse and repeat...
        }
    }
}
```

8.6.4.3 A Limitation of Thread-Based Jobs

Let's imagine writing a job that updates the state of an AI-driven non-player character (NPC). The job's entry point function might look something like this:

```
void NpcThinkJob(uintparam_t param)
{
    Npc* pNpc = reinterpret_cast<Npc*>(param);

    pNpc->StartThinking();
    pNpc->DoSomeMoreUpdating();
    // ...

    // now let's cast a ray to see if we're aiming
    // at anything interesting -- this involves
    // kicking off another job that will run on
    // a different code (worker thread)
    RayCastHandle hRayCast = CastGunAimRay(pNpc);

    // the results of the ray cast aren't going to
    // be ready until later this frame, so let's
    // go to sleep until it's ready
    WaitForRayCast(hRayCast);

    // zzz...

    // wake up!

    // now fire my weapon, but only if the ray
    // cast indicates that we are aiming at an
    // enemy
    pNpc->TryFireWeaponAtTarget(hRayCast);

    // ...
}
```

这项工作看起来很简单：它执行一些更新，启动一个射线投射任务（在另一个工作线程/核心上），以确定 NPC 的枪瞄准了哪个物体。然后，NPC 会开枪，但前提是射线投射报告敌人进入了他的视线。

不幸的是，如果我们尝试在上一节描述的简单作业系统中运行此类作业，它将无法正常工作。问题在于对 WaitForRayCast() 的调用。在我们基于线程池的简单作业系统中，每个作业一旦开始运行就必须运行完成。它无法“进入睡眠状态”等待光线投射的结果，从而允许其他作业在工作线程上运行，然后在光线投射结果准备好后再“唤醒”。

出现这种限制是因为在我们的简单系统中，每个正在运行的作业都与调用它的工作线程共享相同的调用堆栈。要使一个作业进入睡眠状态，我们需要有效地上下文切换到另一个作业。这将涉及保存即将退出的作业的调用堆栈和寄存器，然后覆盖


```
void NpcThinkJob(uintparam_t param)
{
    Npc* pNpc = reinterpret_cast<Npc*>(param);

    pNpc->StartThinking();
    pNpc->DoSomeMoreUpdating();
    // ...

    // now let's cast a ray to see if we're aiming
    // at anything interesting -- this involves
    // kicking off another job that will run on
    // a different code (worker thread)
    RayCastHandle hRayCast = CastGunAimRay(pNpc);

    // the results of the ray cast aren't going to
    // be ready until later this frame, so let's
    // go to sleep until it's ready
    WaitForRayCast(hRayCast);

    // zzz...

    // wake up!

    // now fire my weapon, but only if the ray
    // cast indicates that we are aiming at an
    // enemy
    pNpc->TryFireWeaponAtTarget(hRayCast);

    // ...
}
```

This job seems simple enough: It performs some updates, kicks off a ray cast job (on another worker thread/core) to determine at what object the NPC is aiming his gun. The NPC then fires his weapon, but only if the ray cast reports that an enemy is in his sights.

Unfortunately this kind of job wouldn't work if we tried to run it within the simple job system that we described in the previous section. The problem is the call to `WaitForRayCast()`. In our simple thread-pool-based job system, every job must *run to completion* once it starts running. It cannot "go to sleep" waiting for the results of the ray cast, allowing other jobs to run on the worker thread, and then "wake up" later, when the ray cast results are ready.

This limitation arises because in our simple system, each running job shares the same call stack as the worker thread that invoked it. To put a job to sleep, we would need to effectively *context switch* to another job. That would involve saving off the outgoing job's call stack and registers, and then overwriting the

工作线程的调用堆栈与传入作业的调用堆栈。使用这种简单的作业执行方法时，没有简单的方法可以做到这一点。

8.6.4.4 协同程序形式的作业

解决此问题的一种方法是将基于线程池的作业系统更改为基于协程的作业系统。回想一下第 4.4.8 节，协程具有普通线程不具备的一项重要特性：能够在执行过程中让位于另一个协程，并在稍后另一个协程将控制权交还给它时从中断的地方继续执行。协程可以像这样相互让位于，因为实现实际上会在运行协程的线程内交换传出和传入协程的调用堆栈。因此，与纯基于线程的作业不同，基于协程的作业可以有效地"进入睡眠状态"并允许其他作业在等待诸如光线投射之类的操作完成时运行。

8.6.4.5 作业作为纤程

允许作业睡眠和相互让步的另一种方法是用纤程而不是线程来实现它们。回想一下第 4.4.7 节，纤程和线程之间的主要区别在于纤程之间的上下文切换始终是协作的，而不是抢占的。基于纤程的系统首先将其一个线程转换为纤程。该线程将继续运行该纤程，直到它明确调用 SwitchToFiber() 以明确将控制权交给另一个纤程。就像协程一样，每当发生到另一个纤程的上下文切换时，纤程的整个调用堆栈都会被保存。纤程甚至可以从一个线程迁移到另一个线程。这使得它们非常适合用于实现作业系统。顽皮狗的作业系统基于纤程。

8.6.4.6 作业计数器

如果我们使用协程或纤程来实现作业系统，我们就能够让作业进入睡眠状态（保存其执行上下文），并在将来的某个时间将其唤醒（从而恢复其执行上下文）。这反过来又允许我们为作业系统实现一个 join 函数——该函数使调用作业进入睡眠状态，等待一个或多个其他作业完成执行。该函数大致相当于 POSIX 线程库中的 pthread_join() 或 Windows 下的 WaitForSingleObject()。

实现此目的的一种方法是将句柄与每个作业关联，就像大多数线程库中的线程都有句柄一样。等待作业就相当于调用某种 job::Join() 函数，并传递要等待的作业的句柄。

worker thread's call stack with the incoming job's call stack. There's no simple way to do that when using such a simple job execution method.

8.6.4.4 Jobs as Coroutines

One way to overcome this problem would be to change from a thread pool based job system to one based on *coroutines*. Recall from Section 4.4.8 that a coroutine possesses one important quality that ordinary threads do not: The ability to *yield* to another coroutine part-way through its execution, and to continue from where it left off at some later time when another coroutine yields control back to it. Coroutines can yield to one another like this because the implementation actually does swap the callstacks of the outgoing and incoming coroutines within the thread in which the coroutines are running. So unlike a purely thread-based job, a coroutine-based job could effectively "go to sleep" and allow other jobs to run while it waits for an operation such as a ray cast to complete.

8.6.4.5 Jobs as Fibers

Another way to allow jobs to sleep and yield to one another is to implement them with *fibers* instead of threads. Recall from Section 4.4.7 that the key difference between fibers and threads is that context switches between fibers are always *cooperative*, never *preemptive*. A fiber-based system begins by converting one of its threads into a fiber. The thread will continue to run that fiber until it explicitly calls `SwitchToFiber()` to explicitly *yield* control to another fiber. Just as with coroutines, the entire call stack of a fiber is saved off whenever a context switch to another fiber occurs. Fibers can even migrate from one thread to another. This makes them well suited for use in implementing a job system. Naughty Dog's job system is based on fibers.

8.6.4.6 Job Counters

If we implement our job system with coroutines or fibers, we have the ability to put a job to sleep (saving off its execution context), and to wake it back up at a future time (thus restoring its execution context). This in turn permits us to implement a *join* function for our job system—a function that causes the calling job to go to sleep, waiting until one or more other jobs have completed executing. Such a function would be roughly equivalent to `pthread_join()` in the POSIX thread library or `WaitForSingleObject()` under Windows.

One way to implement this would be to associate a *handle* with each job, just as threads have handles in most thread libraries. Waiting for a job would then amount to calling a `job::Join()` function of some kind, passing the handle of the job for which you wish to wait.

基于句柄的方法的一个缺点是，它无法很好地扩展到等待大量作业。此外，为了等待单个作业完成，我们需要定期轮询以检查系统中所有作业的状态。这种轮询会浪费宝贵的 CPU 周期。出于这些原因，上面介绍的作业系统 API 引入了计数器的概念，它的作用有点像信号量，只是反过来。每当一个作业被踢出时，它可以选择与一个计数器关联（通过 job::Declaration 提供给它）。踢出作业的操作会增加计数器，而当作业终止时，计数器会减少。因此，等待一批作业只需使用相同的计数器将它们全部踢出，然后等到该计数器达到零（这表明所有作业都已完成工作）。等到计数器达到零比轮询单个作业效率高得多，因为可以在计数器减少时进行检查。因此，基于计数器的系统可以提高性能。顽皮狗的工作系统中就使用了类似的计数器。

8.6.4.7 作业同步原语

任何并发程序都需要一种执行原子操作的机制，作业系统也不例外。正如线程库提供了一组线程同步原语（例如互斥锁、条件变量和信号量），作业系统也必须提供一组作业同步原语。

作业同步原语的实现方式因作业系统的实际实现方式而异。但它们通常并非简单地包装内核的线程同步原语。要了解原因，请考虑操作系统互斥锁的作用：当线程尝试获取的锁已被其他线程持有时，它会让该线程进入睡眠状态。如果我们将作业系统实现为线程池，那么在作业中等待互斥锁会导致整个工作线程进入睡眠状态，而不仅仅是那个想要等待锁的作业。显然，这会造成严重的问题，因为在该线程被唤醒之前，任何作业都无法在该线程的核心上运行。这样的系统很可能会出现死锁问题。

为了克服这个问题，作业可以使用自旋锁而不是操作系统互斥锁。只要线程之间锁竞争不多，这种方法就很有效，因为在这种情况下，任何作业都不会忙等待每个尝试获取锁的锁。顽皮狗的作业系统使用自旋锁来满足大多数锁定需求。

然而，有时作业可能会遇到高竞争的情况。设计良好的作业系统可以通过自定义的“互斥”机制来处理这种情况。

One downside of the handle-based approach is that it doesn't scale well to waiting for large numbers of jobs. Also, to wait for an individual job to complete, we'd need to poll periodically to check the status of all jobs in the system. Such polling would waste valuable CPU cycles. For these reasons, the job system API presented above introduces the concept of a *counter*, which acts a bit like a semaphore, only in reverse. Whenever a job is kicked, it can optionally be associated with a counter (provided to it via the `job::Declaration`). The act of kicking the job increments the counter, and when the job terminates the counter is decremented. Waiting for a batch of jobs, then, involves simply kicking them all off with the same counter, and then waiting until that counter reaches zero (which indicates that all jobs have completed their work.) Waiting until a counter reaches zero is much more efficient than polling the individual jobs, because the check can be made at the moment the counter is decremented. As such, a counter based system can be a performance win. Counters like this are used in the Naughty Dog job system.

8.6.4.7 Job Synchronization Primitives

Any concurrent program requires a mechanism for performing *atomic operations*, and a job system is no different. Just as a threading library provides a set of *thread synchronization primitives* such as mutexes, condition variables and semaphores, so too must a job system provide a collection of *job synchronization primitives*.

The implementation of job synchronization primitives varies somewhat depending on how the job system is actually implemented. But they're usually *not* simply wrappers around the kernel's thread synchronization primitives. To see why, consider what an OS mutex does: It puts a thread to sleep whenever the lock it's trying to acquire is already being held by another thread. If we were to implement our job system as a thread pool, then waiting on a mutex within a job would put the *entire worker thread* to sleep, not just the one job that wants to wait for the lock. Clearly this would pose a serious problem, because no jobs would be able to run on that thread's core until the thread wakes back up. Such a system would very likely be fraught with deadlock issues.

To overcome this problem, jobs could use *spin locks* instead of OS mutexes. This approach works well as long as there's not very much lock contention between threads, because in that case no job will ever busy-wait for every lock trying to obtain a lock. Naughty Dog's job system uses spin locks for most of its locking needs.

Sometimes, however, a job may experience a high-contention situation. A well-designed job system can handle this via a custom "mutex" mechanism

可以让作业在等待资源可用时进入休眠状态。当无法获取锁时，此类互斥锁可能会以忙等待的方式启动。如果在短暂的超时后锁仍然不可用，则互斥锁可能会将协程或纤程让给另一个等待的作业，从而使该等待的作业进入休眠状态。正如内核会跟踪所有正在等待互斥锁的休眠线程一样，我们的作业系统也需要跟踪所有休眠的作业，以便在它们的互斥锁释□□放时将其唤醒。

8.6.4.8 作业可□□视化和分析工具

一旦开始使用作业系统，正在运行的作业及其依赖关系图很快就会变得庞大而复杂。为任何作业系统提供可视化和分析工具都是一个好主意。

例如，顽皮狗的作业系统提供了一个可视化视图，如图 8.7 所示。在此视图中，您可以看到七个核心（以及 GPU）中的每一个都沿左侧边缘垂直列出。时间从左到右推进，每个逻辑帧都用垂直标记划分。沿着每个核心的时间轴，细长方块代表在给定帧期间运行的各种作业。每个作业下方还有几行细长矩形，它们代表该作业的调用堆栈（它调用了哪些函数，以及每个函数的运行时间）。

图 8.7。《神秘海域：失落的遗产》（© 2017/TM SIE。由顽皮狗创建和开发，PlayStation 4）及其旗下其他 PS4 游戏中使用的任务系统提供了一个可视化工具，可以显示在给定帧的运行过程中每个核心上运行了哪些任务。时间从左到右递增。任务以每个核心时间线上的细框表示。每个任务调用的函数在其下方显示为额外的细框。

that can put a job to sleep while it waits for a resource to become available. Such a mutex might start by busy-waiting when a lock can't be obtained. If after a brief timeout the lock is still unavailable, the mutex could yield the coroutine or fiber to another waiting job, thereby putting the waiting job to sleep. Just as the kernel keeps track of all sleeping threads that are waiting on a mutex, our job system would need to keep track of all sleeping jobs so that it can wake them up when their mutexes free up.

8.6.4.8 Job Visualization and Profiling Tools

Once you start using a job system, the graph of running jobs and their dependencies becomes large and complex very quickly. It's a good idea to provide visualization and profiling tools with any job system.

For example, the Naughty Dog job system offers a visualization view like the one shown in Figure 8.7. In this display, you can see each of the seven cores (plus the GPU) listed vertically along the left-hand edge. Time progresses from left to right, and each logical frame is demarked with vertical markers. Along each core's timeline, thin blocks represent the various jobs that ran during the given frame. Beneath each job are additional rows of thin rectangles—these represent the call stack of the job (which functions it called, and how long each one took to run).

Figure 8.7. The job system used in *Uncharted: The Lost Legacy* (© 2017/™ SIE. Created and developed by Naughty Dog, PlayStation 4) and Naughty Dog's other PS4 games offers a visualization tool that shows which jobs ran on each core over the course of a given frame. Time increases from left to right. Jobs are represented by thin boxes along the timeline of each core. The functions called by each job are shown beneath it as additional thin boxes.

作业根据其功能进行颜色编码，因此用户可以快速锁定特定作业。例如，假设我们正在寻找花费特别长时间的射线投射。如果射线投射作业标为红色，我们可以直观地扫描显示屏，查找所有比我们需要的宽度更宽的红色作业。单击某个作业会导致所有其他非同一类型的作业都显示为灰色，从而可以轻松查看所选作业类型的所有作业。此外，单击某个作业时，细线会将其与它踢出的作业以及踢出它的作业连接起来。将鼠标悬停在某个作业上，或其下方调用堆栈中的某个函数上，将弹出一些文本，其中包含该作业或函数的名称及其执行时间（以毫秒为单位）。

作业系统的另一个非常有用的功能是我称之为“配置文件陷阱”的功能。假设游戏的某个区域大部分时间都能以 30 FPS 的速度良好运行，但偶尔会降到 24 FPS。我们可以为任何运行时间超过 35 毫秒的帧设置一个陷阱。然后我们就可以正常玩游戏了。一旦检测到运行时间超过 35 毫秒的帧，陷阱系统就会自动暂停游戏，并在屏幕上显示配置文件。这样就可以分析运行该帧的作业，找出导致速度变慢的罪魁祸首。

8.6.4.9 顽皮狗职业系统

顽皮狗在《最后生还者：重制版》、《神秘海域 4：盗贼末路》和《神秘海域：失落的遗产》中使用的作业系统，很大程度上遵循了我们迄今为止讨论过的假设作业系统的设计。它基于纤程（而非线程池或协程）。它使用自旋锁，并提供了一个特殊的作业互斥锁，可以在作业等待锁时使其进入睡眠状态。它使用计数器而不是作业句柄来实现连接操作。

让我们来看看顽皮狗引擎中基于纤程的作业系统是如何工作的。系统首次启动时，主线程会将自身转换为纤程，以便在整个进程中启用纤程。接下来，会生成作业工作线程，每个线程对应 PS4 上开发者可用的七个核心。这些线程通过各自的 CPU 关联设置锁定到各自的核心，因此我们可以将这些工作线程与其核心视为大致相同（尽管实际上，其他优先级更高的线程有时会在帧期间短暂地中断工作线程）。在 PS4 上创建纤程的速度较慢，因此会预先生成一个纤程池，以及用作每个纤程调用堆栈的内存块。

当作业被踢出时，它们的声明会被放入队列。当核心/工作线程空闲时（作业终止），新的作业会从队列中拉出并执行。正在运行的作业也可以将更多作业添加到作业队列中。

为了执行一个作业，从光纤池中拉出一个未使用的光纤，并且

Jobs are color coded by their function, so users can quickly zero in on a particular job. For example, let's say we're looking for a ray cast that's taking a particularly long time. If ray cast jobs are colored red, we can visually scan the display for all the red jobs that are wider than we'd like. Clicking on a job causes all other jobs that are not of the same type to be drawn in grey, allowing all jobs of the selected job's type to be easily seen. Also, when a job is clicked, thin lines connect it to the jobs it kicked, and to the job that kicked it. Hovering your mouse over a job, or over one of the functions in its call stack beneath it, pops up some text with the name of the job or function, and its execution time in milliseconds.

One other very useful job system feature is what I'll call a *profile trap*. Let's say an area of the game is running well at 30 FPS most of the time, but once in a while it dips to 24 FPS. We can set a trap for any frame taking longer than, say, 35 ms. Then we play the game normally. As soon as a frame taking longer than 35 ms is detected, the game is automatically paused by the trap system, and the profile display is put up on-screen. It's then possible to analyze the jobs that ran that frame to find the culprit(s) causing the slowdown.

8.6.4.9 The Naughty Dog Job System

The job system used by Naughty Dog on *The Last of Us: Remastered*, *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy* largely follows the design of the hypothetical job systems we've discussed thus far. It is based on fibers (rather than thread pools or coroutines). It makes use of spin locks and also provides a special job mutex that can put a job to sleep while it waits for a lock. It uses counters rather than job handles to implement a *join* operation.

Let's have a look at how the fiber-based job system used in the Naughty Dog engine works. When the system first boots up, the main thread converts itself to a fiber in order to enable fibers within the process as a whole. Next, job worker threads are spawned, one for each of the seven cores available to developers on the PS4. These threads are each locked to one core via their CPU affinity settings, so we can think of these worker threads and their cores as being roughly synonymous (although in reality, other higher-priority threads do sometimes interrupt the worker threads for very brief periods during the frame). Fiber creation is slow on the PS4, so a pool of fibers is pre-spawned, along with memory blocks to serve as each fiber's call stack.

When jobs are kicked, their declarations are placed onto a queue. As cores/worker threads become free (as jobs terminate), new jobs are pulled from this queue and executed. A running job can also add more jobs to the job queue.

To execute a job, an unused fiber is pulled from the fiber pool, and the

工作线程执行 SwitchToFiber() 来启动作业。当作业从其入口函数返回或以其他方式自行终止时，该作业的最终操作是执行 SwitchToFiber() 返回作业系统本身。然后，它从队列中选择另一个作业，并无限重复该过程。

当作业等待计数器时，该作业将被置于睡眠状态，其纤程（执行上下文）连同其正在等待的计数器一起被放入等待列表。当此计数器归零时，该作业将被唤醒，以便从中断处继续执行。睡眠和唤醒作业同样是通过在每个核心/工作线程上，在作业的纤程和作业系统的管理纤程之间调用 SwitchToFiber() 来实现的。

想要深入了解顽皮狗任务系统的构建方式及其背后的原因，请查看 Christian Gyrling 在 2015 年游戏开发者大会 (GDC 2015) 上的精彩演讲“顽皮狗引擎的并行化”，链接为 https://bit.ly/2H6v0J4。Christian 的幻灯片链接为 https://bit.ly/2ETr5x9。

worker thread performs a `SwitchToFiber()` to start the job running. When a job returns from its entry point function or otherwise self-terminates, the job's final act is to perform a `SwitchToFiber()` back to the job system itself. It then selects another job from the queue, and the process repeats ad infinitum.

When a job waits on a counter, the job is put to sleep and its fiber (execution context) is placed on a wait list, along with the counter it is waiting for. When this counter hits zero, the job is woken back up so it can continue where it left off. Sleeping and waking jobs is again implemented by calling `SwitchToFiber()` between the job's fiber and the job system's management fibers on each core/worker thread.

For an excellent in-depth discussion of how the Naughty Dog job system was built, and why it was built that way, check out Christian Gyrling's excellent GDC 2015 talk entitled, "Parallelizing the Naughty Dog Engine," which is available at https://bit.ly/2H6v0J4. Christian's slides are available at https://bit.ly/2ETr5x9.

9 人机接口设备

游戏是交互式计算机模拟，因此人类玩家需要某种方式为游戏提供输入。游戏中存在各种各样的人机接口设备 (HID)，包括操纵杆、游戏手柄、键盘和鼠标、轨迹球、Wii 遥控器，以及方向盘、钓鱼竿、跳舞毯甚至电吉他等专用输入设备。在本章中，我们将探讨游戏引擎通常如何读取、处理和利用来自人机接口设备的输入。我们还将探讨这些设备的输出如何向人类玩家提供反馈。

9.1 人机接口设备的类型

各种各样的人机界面设备可用于游戏。Xbox 360 和 PS3 等游戏机配备了游戏手柄控制器，如图 9.1 和 9.2 所示。任天堂的 Wii 游戏机以其独特创新的 Wii 遥控器（通常称为“Wiimote”）而闻名，如图 9.3 所示。而 Wii U 则将控制器与半移动游戏设备进行了创新性的组合（图 9.4）。PC 游戏通常通过键盘和鼠标或游戏手柄进行控制。（微软设计的 Xbox 360 游戏手柄既可以在 Xbox 360 上使用，也可以在 Windows/DirectX PC 平台上使用。）如图 9.5 所示，街机内置一个或多个控制器，例如操纵杆和各种按钮，或者轨迹球、方向盘等。街机的输入设备通常会根据游戏进行一定程度的定制。

9 Human Interface Devices

Games are interactive computer simulations, so the human player(s) need some way of providing inputs to the game. All sorts of *human interface devices* (HID) exist for gaming, including joysticks, joypads, keyboards and mice, track balls, the Wii remote and specialized input devices like steering wheels, fishing rods, dance pads and even electric guitars. In this chapter, we'll investigate how game engines typically read, process and utilize the inputs from human interface devices. We'll also have a look at how outputs from these devices provide feedback to the human player.

9.1 Types of Human Interface Devices

A wide range of human interface devices are available for gaming purposes. Consoles like the Xbox 360 and PS3 come equipped with joypad controllers, as shown in Figures 9.1 and 9.2. Nintendo's Wii console is well known for its unique and innovative Wii Remote controller (commonly referred to as the "Wiimote"), shown in Figure 9.3. And with the Wii U, Nintendo has created an innovative mix between a controller and a semi-mobile gaming device (Figure 9.4). PC games are generally either controlled via a keyboard and mouse or via a joypad. (Microsoft designed the Xbox 360 joypad so that it can be used both on the Xbox 360 and on Windows/DirectX PC platforms.) As shown in Figure 9.5, arcade machines have one or more built-in controllers, such as a joystick and various buttons, or a track ball, a steering wheel, etc. An arcade machine's input device is usually somewhat customized to the game in

图 9.1. Xbox 360 和 PlayStation 3 游戏机的标准游戏手柄。

图 9.2. PlayStation 4 的 DualShock 4 游戏手柄。

图 9.3. 适用于任天堂 Wii 的创新型 Wii 遥控器。

图 9.4.任天堂的 Wii U 控制器。

问题，尽管输入硬件经常在同一制造商生产的街机之间重复使用。

在游戏机平台上，除了“标准”输入设备（例如游戏手柄）之外，通常还有专门的输入设备和适配器。例如，《吉他英雄》系列游戏提供吉他和鼓设备；赛车游戏可以购买方向盘；而像《劲舞革命》这样的游戏则使用特殊的跳舞毯设备。其中一些设备

图 9.5.Midway 街机游戏《真人快打 II》的按钮和操纵杆。

Figure 9.1. Standard joypads for the Xbox 360 and PlayStation 3 consoles.

Figure 9.2. The DualShock 4 joypad for the PlayStation 4.

Figure 9.3. The innovative Wii Remote for the Nintendo Wii.

Figure 9.4. The Wii U controller by Nintendo.

question, although input hardware is often reused among arcade machines produced by the same manufacturer.

On console platforms, specialized input devices and adapters are usually available, in addition to the “standard” input device such as the joypad. For example, guitar and drum devices are available for the *Guitar Hero* series of games, steering wheels can be purchased for driving games, and games like *Dance Dance Revolution* use a special dance pad device. Some of these devices

Figure 9.5. Buttons and joysticks for the arcade game *Mortal Kombat II* by Midway.

图 9.6. 许多专用输入设备可与控制台一起使用。

图 9.7. 任天堂 Wii 的方向盘适配器。

如图9.6所示。

任天堂 Wiimote 是目前市面上最灵活的输入设备之一。因此，它经常被改装用于新的用途，而不是被一个全新的设备取代。例如，《马里奥赛车 Wii》配备了一个塑料方向盘适配器，可以将 Wiimote 插入其中（参见图 9.7）。

9.2 与 HID 接口

所有人机接口设备都会为游戏软件提供输入，有些设备还允许游戏软件通过各种输出向人类玩家提供反馈。游戏软件会以各种方式读取和写入 HID 输入和输出，具体取决于相关设备的具体设计。

9.2.1 轮询

一些简单的设备，例如游戏手柄和老式摇杆，是通过定期轮询硬件来读取的（通常主游戏循环每次迭代一次）。这意味着需要显式地查询设备的状态，可以通过读取

Figure 9.6. Many specialized input devices are available for use with consoles.

Figure 9.7. Steering wheel adapter for the Nintendo Wii.

are shown in Figure 9.6.

The Nintendo Wiimote is one of the most flexible input devices on the market today. As such, it is often adapted to new purposes, rather than replaced with an entirely new device. For example, *Mario Kart Wii* comes with a plastic steering wheel adapter into which the Wiimote can be inserted (see Figure 9.7).

9.2 Interfacing with a HID

All human interface devices provide input to the game software, and some also allow the software to provide feedback to the human player via various kinds of outputs as well. Game software reads and writes HID inputs and outputs in various ways, depending on the specific design of the device in question.

9.2.1 Polling

Some simple devices, like game pads and old-school joysticks, are read by *polling* the hardware periodically (usually once per iteration of the main game loop). This means explicitly querying the state of the device, either by read-

直接访问硬件寄存器，读取内存映射的 I/O 端口，或者通过更高级别的软件接口（反过来，读取相应的寄存器或内存映射的 I/O 端口）来实现。同样，输出可以通过写入特殊寄存器或内存映射的 I/O 地址，或者通过更高级别的 API 来发送给 HID，由后者为我们完成繁琐的工作。

微软的 XInput API 是简单轮询机制□□的一个很好的例子，它适用于 Xbox 360 和 Windows PC 平台上的 Xbox 360 游戏手柄。游戏每一帧都会调用函数 XInputGetState() 。该函数与硬件和/或驱动程序通信，以适当的方式读取数据并将其打包，以便软件使用。它返回一个指向 XINPUT_STATE 结构体的指针，该结构体又包含一个名为 XINPUT_GAMEPAD 的结构体的嵌入实例。该结构体包含设备上所有控件（按钮、摇杆和扳机键）的当前状态。

9.2.2 中断

某些 HID 仅在控制器状态发生某种变化时才向游戏引擎发送数据。例如，鼠标大部分时间都静止在鼠标垫上。当鼠标静止不动时，没有必要在鼠标和计算机之间发送连续的数据流——我们只需要在鼠标移动或按下或释放按钮时传输信息。

这类设备通常通过硬件中断与主机通信。中断是由硬件产生的电子信号，它会导致 CPU 暂时停止主程序的执行，并运行一小段称为中断服务例程 (ISR) 的代码。中断可用于各种用途，但对于 HID 来说，ISR 代码可能会读取设备的状态，将其存储起来以供后续处理，然后将 CPU 交还给主程序。游戏引擎可以在下次方便的时候获取数据。

9.2.3 无线设备

蓝牙设备（例如 Wiimote、DualShock 3 和 Xbox 360 无线控制器）的输入和输出无法通过简单地访问寄存器或内存映射的 I/O 端口来读写。相反，软件必须通过蓝牙协议与设备“对话”。软件可以请求 HID 将输入数据（例如其按钮的状态）发送回主机，也可以将输出数据（例如震动设置或音频数据流）发送回设备。这种通信通常由一个独立的线程处理。

ing hardware registers directly, reading a memory-mapped I/O port, or via a higher-level software interface (which, in turn, reads the appropriate registers or memory-mapped I/O ports). Likewise, outputs might be sent to the HID by writing to special registers or memory-mapped I/O addresses, or via a higher-level API that does our dirty work for us.

Microsoft's XInput API, for use with Xbox 360 game pads on both the Xbox 360 and Windows PC platforms, is a good example of a simple polling mechanism. Every frame, the game calls the function `XInputGetState()`. This function communicates with the hardware and/or drivers, reads the data in the appropriate way and packages it all up for convenient use by the software. It returns a pointer to an `XINPUT_STATE` struct, which in turn contains an embedded instance of a struct called `XINPUT_GAMEPAD`. This struct contains the current states of all of the controls (buttons, thumb sticks and triggers) on the device.

9.2.2 Interrupts

Some HIDs only send data to the game engine when the state of the controller changes in some way. For example, a mouse spends a lot of its time just sitting still on the mouse pad. There's no reason to send a continuous stream of data between the mouse and the computer when the mouse isn't moving—we need only transmit information when it moves, or a button is pressed or released.

This kind of device usually communicates with the host computer via *hardware interrupts*. An interrupt is an electronic signal generated by the hardware, which causes the CPU to temporarily suspend execution of the main program and run a small chunk of code called an *interrupt service routine* (ISR). Interrupts are used for all sorts of things, but in the case of a HID, the ISR code will probably read the state of the device, store it off for later processing, and then relinquish the CPU back to the main program. The game engine can pick up the data the next time it is convenient to do so.

9.2.3 Wireless Devices

The inputs and outputs of a Bluetooth device, like the Wiimote, the DualShock 3 and the Xbox 360 wireless controller, cannot be read and written by simply accessing registers or memory-mapped I/O ports. Instead, the software must "talk" to the device via the Bluetooth protocol. The software can request the HID to send input data (such as the states of its buttons) back to the host, or it can send output data (such as rumble settings or a stream of audio data) to the device. This communication is often handled by a thread separate

从游戏引擎的主循环中获取信息，或者至少封装在一个相对简单的接口中，以便从主循环中调用。因此，从游戏程序员的角度来看，蓝牙设备的状态可以看起来与传统的轮询设备几乎没有区别。

9.3 输入类型

尽管游戏的人机界面设备在外形和布局方面差异很大，但它们提供的大多数输入都属于少数几个类别之一。下文我们将深入探讨每个类别。

9.3.1 数字按钮

几乎每个 HID 都至少有几个数字按钮。这些按钮只能处于两种状态之一：按下和未按下。游戏程序员通常将按下的按钮称为按下，将未按下的按钮称为弹起。

电气工程师将包含开关的电路称为闭合（即电流流过电路）或断开（即无电流流过电路——电路具有无穷大电阻）。闭合是否对应于按下或未按下取决于硬件。如果开关为常开，则当未按下（向上）时，电路处于断开状态；而当按下（向下）时，电路处于闭合状态。如果开关为常闭，则情况相反——按下按钮的动作会断开电路。

在软件中，数字按钮的状态（按下或未按下）通常用一位表示。通常用 0 表示未按下（向上），用 1 表示按下（向下）。但同样，根据电路的性质以及编写设备驱动程序的程序员的决策，这些值的含义可能会相反。

将设备上所有按钮的状态打包成一个无符号整数值是很常见的。例如，在微软的 XInput API 中，Xbox 360 游戏手柄的状态在一个名为 XINPUT _GAMEPA D 的结构体中返回，如下所示。

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
```


from the game engine's main loop, or at least encapsulated behind a relatively simple interface that can be called from the main loop. So from the point of view of the game programmer, the state of a Bluetooth device can be made to look pretty much indistinguishable from a traditional polled device.

9.3 Types of Inputs

Although human interface devices for games vary widely in terms of form factor and layout, most of the inputs they provide fall into one of a small number of categories. We'll investigate each category in depth below.

9.3.1 Digital Buttons

Almost every HID has at least a few *digital buttons*. These are buttons that can only be in one of two states: *pressed* and *not pressed*. Game programmers often refer to a pressed button as being *down* and a non-pressed button as being *up*.

Electrical engineers speak of a circuit containing a switch as being *closed* (meaning electricity is flowing through the circuit) or *open* (no electricity is flowing—the circuit has infinite *resistance*). Whether *closed* corresponds to *pressed* or *not pressed* depends on the hardware. If the switch is *normally open*, then when it is not pressed (up), the circuit is *open*, and when it is pressed (down), the circuit is *closed*. If the switch is *normally closed*, the reverse is true—the act of pressing the button opens the circuit.

In software, the state of a digital button (pressed or not pressed) is usually represented by a single bit. It's common for 0 to represent not pressed (up) and 1 to represent pressed (down). But again, depending on the nature of the circuitry and the decisions made by the programmers who wrote the device driver, the sense of these values might be reversed.

It is quite common for the states of all of the buttons on a device to be packed into a single unsigned integer value. For example, in Microsoft's XInput API, the state of the Xbox 360 joypad is returned in a struct called `XINPUT _GAMEPAD`, shown below.

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
```

```
} XINPUT_GAMEPAD;
```

此结构体包含一个名为 wButtons 的 16 位无符号整数 (WORD) 变量，用于保存所有按钮的状态。以下掩码定义了该字中每个位对应的物理按钮。（请注意，第 10 位和第 11 位未使用。）

```
#define XINPUT_GAMEPAD_DPAD_UP          0x0001 // bit 0
#define XINPUT_GAMEPAD_DPAD_DOWN        0x0002 // bit 1
#define XINPUT_GAMEPAD_DPAD_LEFT        0x0004 // bit 2
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x0008 // bit 3
#define XINPUT_GAMEPAD_START            0x0010 // bit 4
#define XINPUT_GAMEPAD_BACK             0x0020 // bit 5
#define XINPUT_GAMEPAD_LEFT_THUMB       0x0040 // bit 6
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x0080 // bit 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100 // bit 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200 // bit 9
#define XINPUT_GAMEPAD_A                0x1000 // bit 12
#define XINPUT_GAMEPAD_B                0x2000 // bit 13
#define XINPUT_GAMEPAD_X                0x4000 // bit 14
#define XINPUT_GAMEPAD_Y                0x8000 // bit 15
```

可以通过 C/C++ 的按位与运算符（&）将 wButtons 字与相应的位掩码进行掩码运算，然后检查结果是否为非零，来读取单个按钮的状态。例如，要判断 A 按钮是否被按下（按下），我们可以这样写：

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
    // Mask off all bits but bit 12 (the A button).
    return ((pad.wButtons & XINPUT_GAMEPAD_A) != 0);
}
```

9.3.2 模拟轴和按钮

模拟输入可以接受一系列值（而不仅仅是 0 或 1）。这类输入通常用于表示扳机按下的程度，或操纵杆的二维位置（使用两个模拟输入表示，一个代表 x 轴，一个代表 y 轴，如图 9.8 所示）。由于这种常见用法，模拟输入有时也称为模拟轴，或简称为轴。

在某些设备上，某些按钮也是模拟的，这意味着游戏实际上可以检测到玩家按下这些按钮的力度。然而，模拟按钮产生的信号通常噪音太大，不太实用。能够有效利用模拟按钮输入的游戏很少。一个好的


```
} XINPUT_GAMEPAD;
```

This struct contains a 16-bit unsigned integer (`WORD`) variable named `wButtons` that holds the state of all buttons. The following masks define which physical button corresponds to each bit in the word. (Note that bits 10 and 11 are unused.)

```
#define XINPUT_GAMEPAD_DPAD_UP          0x0001 // bit 0
#define XINPUT_GAMEPAD_DPAD_DOWN        0x0002 // bit 1
#define XINPUT_GAMEPAD_DPAD_LEFT        0x0004 // bit 2
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x0008 // bit 3
#define XINPUT_GAMEPAD_START            0x0010 // bit 4
#define XINPUT_GAMEPAD_BACK             0x0020 // bit 5
#define XINPUT_GAMEPAD_LEFT_THUMB       0x0040 // bit 6
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x0080 // bit 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100 // bit 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200 // bit 9
#define XINPUT_GAMEPAD_A                0x1000 // bit 12
#define XINPUT_GAMEPAD_B                0x2000 // bit 13
#define XINPUT_GAMEPAD_X                0x4000 // bit 14
#define XINPUT_GAMEPAD_Y                0x8000 // bit 15
```

An individual button's state can be read by masking the `wButtons` word with the appropriate bitmask via C/C++'s bitwise AND operator (&) and then checking if the result is nonzero. For example, to determine if the A button is pressed (down), we would write:

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
    // Mask off all bits but bit 12 (the A button).
    return ((pad.wButtons & XINPUT_GAMEPAD_A) != 0);
}
```

9.3.2 Analog Axes and Buttons

An *analog input* is one that can take on a range of values (rather than just 0 or 1). These kinds of inputs are often used to represent the degree to which a trigger is pressed, or the two-dimensional position of a joystick (which is represented using two analog inputs, one for the *x*-axis and one for the *y*-axis, as shown in Figure 9.8). Because of this common usage, analog inputs are sometimes called *analog axes*, or just *axes*.

On some devices, certain buttons are analog as well, meaning that the game can actually detect how hard the player is pressing on them. However, the signals produced by analog buttons are usually too noisy to be particularly usable. Games that use analog button inputs effectively are rare. One good

图 9.8 两个模拟输入可用于表示操纵杆的 x 和 y 偏转。

例如 PS2 上的《合金装备 2》。它在瞄准模式下使用压敏（模拟）按钮数据来区分快速释放 X 按钮（发射武器）和缓慢释放（中止射击）——这在潜行游戏中非常有用，因为在潜行游戏中，除非万不得已，否则你不会想惊动敌人！

严格来说，模拟输入在进入游戏引擎时并非真正的模拟信号。模拟输入信号通常被数字化，这意味着它被量化并在软件中使用整数表示。例如，如果用 16 位有符号整数表示，模拟输入的取值范围可能是 -32,768 到 32,767。有时，模拟输入会被转换为浮点数——例如，浮点数的取值范围可能是 -1 到 1。但正如我们从 3.3.1.3 节所知，浮点数实际上也只是量化的数字值。

回顾 XINPUT_GAMEPAD 的定义（如下所示），我们可以看到微软选择使用 16 位有符号整数（sThumbLX 和 sThumbLY 表示左摇杆，sThumbRX 和 sThumbRY 表示右摇杆）来表示 Xbox 360 游戏手柄左右摇杆的偏转。因此，这些值的范围从 -32,768（左或下）到 32,767（右或上）。但是，为了表示左右肩部扳机键的位置，微软选择使用 8 位无符号整数（分别为 bLeftTrigger 和 bRightTrigger）。这些输入值的范围从 0（未按下）到 255（完全按下）。不同的游戏机对其模拟轴使用不同的数字表示形式。

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;

    // 8-bit unsigned
    BYTE  bLeftTrigger;
```

Figure 9.8. Two analog inputs can be used to represent the x and y deflection of a joystick.

example is *Metal Gear Solid 2* on the PS2. It uses pressure-sensitive (analog) button data in aim mode to tell the difference between releasing the X button quickly (which fires the weapon) and releasing it slowly (which aborts the shot)—a useful feature in a stealth game, where you don't want to alert the enemies unless you have to!

Strictly speaking, analog inputs are not really analog by the time they make it to the game engine. An analog input signal is usually *digitized*, meaning it is quantized and represented using an integer in software. For example, an analog input might range from −32,768 to 32,767 if represented by a 16-bit signed integer. Sometimes analog inputs are converted to floating point—the values might range from −1 to 1, for instance. But as we know from Section 3.3.1.3, floating-point numbers are really just quantized digital values as well.

Reviewing the definition of `XINPUT_GAMEPAD` (repeated below), we can see that Microsoft chose to represent the deflections of the left and right thumb sticks on the Xbox 360 gamepad using 16-bit signed integers (`sThumbLX` and `sThumbLY` for the left stick and `sThumbRX` and `sThumbRY` for the right). Hence, these values range from −32,768 (left or down) to 32,767 (right or up). However, to represent the positions of the left and right shoulder triggers, Microsoft chose to use eight-bit unsigned integers (`bLeftTrigger` and `bRightTrigger`, respectively). These input values range from 0 (not pressed) to 255 (fully pressed). Different game machines use different digital representions for their analog axes.

```
typedef struct _XINPUT_GAMEPAD
{
    WORD  wButtons;

    // 8-bit unsigned
    BYTE  bLeftTrigger;
```

```
    BYTE  bRightTrigger;

    // 16-bit signed
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

9.3.3 相对轴

模拟按钮、扳机键、操纵杆或拇指摇杆的位置是绝对的，这意味着零点的位置是明确的。然而，某些设备的输入是相对的。对于这些设备，没有明确的零点位置。零输入表示设备的位置没有变化，而非零值则表示与上次读取输入值时相比的变化。例如鼠标、鼠标滚轮和轨迹球。

9.3.4 加速度计

PlayStation 的 DualShock 游戏手柄和任天堂 Wiimote 都包含加速度传感器（加速度计）。这些设备可以检测沿三个主轴（x 、 y 和 z ）的加速度，如图 9.9 所示。这些是相对模拟输入，很像鼠标的二维轴。当控制器未加速时，这些输入为零；但当控制器加速时，它们会测量沿每个轴最高 ± 3 g 的加速度，并量化为三个有符号的八位整数，分别代表 x 、 y 和 z 。

9.3.5 使用 Wiimote 或 DualShock 进行 3D 定位

一些 Wii 和 PS3 游戏利用 Wiimote 或 DualShock 手柄上的三个加速度计来估算玩家手中控制器的方向。例如，在《超级马里奥银河》中，马里奥跳上了一个大型

图 9.9. Wiimote 的加速度计轴。


```
    BYTE  bRightTrigger;

    // 16-bit signed
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

9.3.3 Relative Axes

The position of an analog button, trigger, joystick or thumb stick is *absolute*, meaning that there is a clear understanding of where zero lies. However, the inputs of some devices are *relative*. For these devices, there is no clear location at which the input value should be zero. Instead, a zero input indicates that the position of the device has not changed, while nonzero values represent a delta from the last time the input value was read. Examples include mice, mouse wheels and track balls.

9.3.4 Accelerometers

The PlayStation's DualShock joypads and the Nintendo Wiimote all contain acceleration sensors (accelerometers). These devices can detect acceleration along the three principal axes (x, y and z), as shown in Figure 9.9. These are *relative* analog inputs, much like a mouse's two-dimensional axes. When the controller is not accelerating these inputs are zero, but when the controller is accelerating, they measure the acceleration up to ± 3 g along each axis, quantized into three signed eight-bit integers, one for each of x, y and z.

9.3.5 3D Orientation with the Wiimote or DualShock

Some Wii and PS3 games make use of the three accelerometers in the Wiimote or DualShock joypad to estimate the orientation of the controller in the player's hand. For example, in *Super Mario Galaxy*, Mario hops onto a large

Figure 9.9. Accelerometer axes for the Wiimote.

用脚把球滚来滚去。要在此模式下控制马里奥，需要手持 Wiimote，并将红外传感器朝向天花板。向左、向右、向前或向后倾斜 Wiimote，球就会向相应的方向加速。

由于我们在地球表面玩 Wiimote 或 DualShock 游戏，由于重力作用，地球表面存在一个恒定的向下加速度，即 1g（≈ 9.8 m/s²），因此可以使用三个加速度计来检测游戏手柄的方向。如果控制器保持完全水平，红外传感器指向电视机，则垂直（z）方向的加速度应约为 -1g。

如果控制器直立，红外传感器指向天花板，我们预计 z 轴传感器上的加速度为 0 g，y 轴传感器上的加速度为 +1 g（因为它现在正受到重力作用）。将 Wiimote 以 45 度角握住，y 轴和 z 轴输入上产生的加速度大约为 sin (45 ∘) = cos (45 ∘) = 0.707 g。校准加速度计输入并找到各轴的零点后，我们可以使用反正弦和余弦运算轻松计算俯仰、偏航和滚转。

这里有两点需要注意：首先，如果握持 Wiimote 的人没有将其静止不动，加速度计的输入值将包含该加速度，从而使我们的计算结果无效。其次，加速度计的 z 轴已校准以考虑重力，但其他两个轴尚未校准。这意味着 z 轴在检测方向时精度较低。许多 Wii 游戏要求用户以非标准方向握持 Wiimote，例如将按钮朝向玩家胸部，或将红外传感器指向天花板。通过将加速度计的 x 轴或 y 轴置于重力方向上，而不是重力校准的 z 轴，可以最大限度地提高方向读数的精度。有关此主题的更多信息，请参阅 http://druid.caughq.org/presentations/turbo/Wiimote-Hacking.pdf。

9.3.6 相机

Wiimote 具有其他标准主机 HID 所不具备的独特功能——红外 (IR) 传感器。该传感器本质上是一个低分辨率摄像头，可以记录 Wiimote 所指向物体的二维红外图像。Wii 配备了一个“传感器条”，位于电视机顶部，包含两个红外发光二极管 (LED)。在红外摄像头记录的图像中，这些 LED 在黑暗的背景上显示为两个亮点。Wiimote 中的图像处理软件会分析图像并分离出这两个点的位置和大小。（实际上，它可以检测并传输最多四个点的位置和大小。）这个位置

ball and rolls it around with his feet. To control Mario in this mode, the Wiimote is held with the IR sensor facing the ceiling. Tilting the Wiimote left, right, forward or back causes the ball to accelerate in the corresponding direction.

A trio of accelerometers can be used to detect the orientation of the Wiimote or DualShock joypad, because of the fact that we are playing these games on the surface of the Earth where there is a constant downward acceleration due to gravity of 1 g ($\approx 9.8\ \mathrm{m/s^2}$). If the controller is held perfectly level, with the IR sensor pointing toward your TV set, the vertical (z) acceleration should be approximately -1 g.

If the controller is held upright, with the IR sensor pointing toward the ceiling, we would expect to see a 0 g acceleration on the z sensor, and $+1$ g on the y sensor (because it is now experiencing the full gravitational effect). Holding the Wiimote at a 45-degree angle should produce roughly $\sin(45^\circ) = \cos(45^\circ) = 0.707$ g on both the y and z inputs. Once we've calibrated the accelerometer inputs to find the zero points along each axis, we can calculate pitch, yaw and roll easily, using inverse sine and cosine operations.

Two caveats here: First, if the person holding the Wiimote is not holding it still, the accelerometer inputs will include this acceleration in their values, invalidating our math. Second, the z-axis of the accelerometer has been calibrated to account for gravity, but the other two axes have not. This means that the z-axis has less precision available for detecting orientation. Many Wii games request that the user hold the Wiimote in a nonstandard orientation, such as with the buttons facing the player's chest, or with the IR sensor pointing toward the ceiling. This maximizes the precision of the orientation reading by placing the x- or y-accelerometer axis in line with gravity, instead of the gravity-calibrated z-axis. For more information on this topic, see http://druid.caughq.org/presentations/turbo/Wiimote-Hacking.pdf.

9.3.6 Cameras

The Wiimote has a unique feature not found on any other standard console HID—an infrared (IR) sensor. This sensor is essentially a low-resolution camera that records a two-dimension infrared image of whatever the Wiimote is pointed at. The Wii comes with a "sensor bar" that sits on top of your television set and contains two infrared light emitting diodes (LEDs). In the image recorded by the IR camera, these LEDs appear as two bright dots on an otherwise dark background. Image processing software in the Wiimote analyzes the image and isolates the location and size of the two dots. (Actually, it can detect and transmit the locations and sizes of up to four dots.) This position

图 9.10 Wii 传感器条内有两个红外 LED，它们在 Wiimote 的红外摄像头记录的图像上产生两个亮点。

控制台可以通过蓝牙无线连接读取尺寸信息。

两个点组成的线段的位置和方向可以用来确定 Wiimote 的俯仰、偏航和滚转（只要 Wiimote 指向传感器条即可）。通过观察点之间的距离，软件还可以确定 Wiimote 与电视的距离。有些软件还会利用点的大小。如图 9.10 所示。

图 9.11 索尼为 PS3 设计的 PlayStation Eye。

另一款流行的摄像设备是索尼为 PS3 设计的 PlayStation Eye，如图 9.11 所示。这款设备本质上是一款高质量的彩色摄像头，用途广泛。它可以像任何网络摄像头一样用于简单的视频会议。它还可以像 Wiimote 的红外摄像头一样，用于位置、方向和深度感知。这类先进输入设备的广泛应用前景才刚刚开始被游戏社区所挖掘。

随着 PlayStation 4 的推出，索尼改进了 Eye，并将其重新命名为 PlayStation 摄像头。当与 PlayStation Move 控制器（见图 9.12）或 DualShock 4 控制器结合使用时，PlayStation 可以检测手势，其方式与微软创新的 Kinect 系统（图 9.13）基本相同。

Figure 9.10. The Wii sensor bar houses two infrared LEDs, which produce two bright spots on the image recorded by the Wiimote's IR camera.

and size information can be read by the console via a Bluetooth wireless connection.

The position and orientation of the line segment formed by the two dots can be used to determine the pitch, yaw and roll of the Wiimote (as long as it is being pointed toward the sensor bar). By looking at the separation between the dots, software can also determine how close or far away the Wiimote is from the TV. Some software also makes use of the sizes of the dots. This is illustrated in Figure 9.10.

Figure 9.11. Sony's PlayStation Eye for the PS3.

Another popular camera device is Sony's PlayStation Eye for the PS3, shown in Figure 9.11. This device is basically a high-quality color camera, which can be used for a wide range of applications. It can be used for simple video conferencing, like any web cam. It could also conceivably be used much like the Wiimote's IR camera, for position, orientation and depth sensing. The gamut of possibilities for these kinds of advanced input devices has only begun to be tapped by the gaming community.

With the PlayStation 4, Sony has improved the Eye and re-dubbed it the PlayStation Camera. When combined with the PlayStation Move controller (see Figure 9.12) or the DualShock 4 controller, the PlayStation can detect gestures in basically the same way as Microsoft's innovative Kinect system can (Figure 9.13).

图 9.12 索尼的 PlayStation 摄像头、PlayStation Move 控制器和用于 PS4 的 DualShock 4 游戏手柄。

图 9.13。适用于 Xbox 360（上）和 Xbox One（下）的 Microsoft Kinect。

9.4 输出类型

人机接口设备主要用于将玩家的输入传输到游戏软件。然而，一些 HID 也可以通过各种输出向人类玩家提供反馈。

9.4.1 隆隆声

PlayStation DualShock 系列控制器以及 Xbox 和 Xbox 360 控制器等游戏手柄都具有震动功能。这使得控制器能够在玩家手中震动，模拟游戏世界中角色可能经历的湍流或冲击。震动通常

Figure 9.12. Sony's PlayStation Camera, PlayStation Move controller and DualShock 4 joypad for the PS4.

Figure 9.13. The Microsoft Kinect for Xbox 360 (top) and Xbox One (bottom).

9.4 Types of Outputs

Human interface devices are primarily used to transmit inputs from the player to the game software. However, some HIDs can also provide feedback to the human player via various kinds of outputs.

9.4.1 Rumble

Game pads like the PlayStation's DualShock line of controllers and the Xbox and Xbox 360 controllers have a *rumble* feature. This allows the controller to vibrate in the player's hands, simulating the turbulence or impacts that the character in the game world might be experiencing. Vibrations are usually

由一个或多个电机驱动，每个电机以不同的速度旋转一个略微不平衡的重物。游戏可以打开或关闭这些电机，并控制它们的速度，从而在玩家手中产生不同的触觉效果。

9.4.2 力反馈

力反馈是一种技术，利用电机驱动 HID 上的执行器，从而略微抵抗人类操作员试图施加的运动。这种技术在街机赛车游戏中很常见，方向盘会抵抗玩家转动方向盘的尝试，模拟复杂的驾驶条件或急转弯。与隆隆声一样，游戏软件通常可以打开和关闭电机，还可以控制施加到执行器上的力的强度和方向。

9.4.3 音频

音频通常是一个独立的引擎系统。然而，一些 HID 提供可供音频系统使用的输出。例如，Wiimote 包含一个小型低质量扬声器。Xbox 360、Xbox One 和 DualShock 4 控制器都带有耳机插孔，可以像任何 USB 音频设备一样用于输出（扬声器）和输入（麦克风）。USB 耳机的一个常见用途是多人游戏，玩家可以通过 IP 语音 (VoIP) 连接相互通信。

9.4.4 其他输入和输出

当然，人机界面设备可能支持许多其他类型的输入和输出。在一些较老的游戏机上，例如世嘉 Dreamcast，存储卡插槽位于游戏手柄上。Xbox 360 游戏手柄、Sixaxis 和 DualShock 3 以及 Wiimote 都配有四个 LED，可以根据需要通过游戏软件点亮。DualShock 4 控制器正面的灯条颜色可以通过游戏软件控制。当然，乐器、跳舞毯等专用设备也有其特定的输入和输出类型。

人机界面领域的创新正在蓬勃发展。手势界面和意念控制设备是当今最受关注的领域之一。我们完全可以期待未来几年游戏机和 HID 制造商带来更多创新。

9.5 游戏引擎HID系统

大多数游戏引擎不会直接使用“原始” HID 输入。数据通常会经过各种处理，以确保来自 HID 的输入

produced by one or more motors, each of which rotates a slightly unbalanced weight at various speeds. The game can turn these motors on and off, and control their speeds to produce different tactile effects in the player's hands.

9.4.2 Force-Feedback

Force-feedback is a technique in which an actuator on the HID is driven by a motor in order to slightly resist the motion the human operator is trying to impart to it. It is common in arcade driving games, where the steering wheel resists the player's attempt to turn it, simulating difficult driving conditions or tight turns. As with rumble, the game software can typically turn the motor(s) on and off, and can also control the strength and direction of the forces applied to the actuator.

9.4.3 Audio

Audio is usually a stand-alone engine system. However, some HIDs provide outputs that can be utilized by the audio system. For example, the Wiimote contains a small, low-quality speaker. The Xbox 360, Xbox One and DualShock 4 controllers have a headphone jack and can be used just like any USB audio device for both output (speakers) and input (microphone). One common use of USB headsets is for multiplayer games, in which human players can communicate with one another via a voice over IP (VoIP) connection.

9.4.4 Other Inputs and Outputs

Human interface devices may of course support many other kinds of inputs and outputs. On some older consoles like the Sega Dreamcast, the memory card slots were located on the game pad. The Xbox 360 game pad, the Sixaxis and DualShock 3, and the Wiimote all have four LEDs which can be illuminated by game software if desired. The color of the light bar on the front of the DualShock 4 controller can be controlled by game software. And of course specialized devices like musical instruments, dance pads, etc. have their own particular kinds of inputs and outputs.

Innovation is actively taking place in the field of human interfaces. Some of the most interesting areas today are gestural interfaces and thought-controlled devices. We can certainly expect more innovation from console and HID manufacturers in years to come.

9.5 Game Engine HID Systems

Most game engines don't use "raw" HID inputs directly. The data is usually massaged in various ways to ensure that the inputs coming from the HID

转化为流畅、愉悦且直观的游戏行为。此外，大多数引擎会在 HID 和游戏之间引入至少一个额外的间接层级，以便以各种方式抽象 HID 输入。例如，可以使用按钮映射表将原始按钮输入转换为逻辑游戏操作，以便人类玩家可以根据自己的需要重新分配按钮的功能。在本节中，我们将概述游戏引擎 HID 系统的典型要求，然后深入探讨每个要求。

9.5.1 典型要求

游戏引擎的 HID 系统通常提供以下部分或全部功能：

- 盲区，
- 模拟信号滤波，
- 事件检测（例如，按钮弹起、按钮按下），
- 检测按钮序列和多按钮组合（称为和弦），
- 手势检测，
- 为多个玩家管理多个 HID，
- 多平台 HID 支持，
- 控制器输入重新映射，
- 上下文相关输入，以及
- 暂时禁用某些输入的能力。

9.5.2 盲区

操纵杆、拇指摇杆、肩部扳机或任何其他模拟轴都会□□产生介于预定义最小值和最大值之间的输入值，我们将其称为 I min 和 I max 。当控件未被触摸时，我们期望它产生一个稳定清晰的“未受干扰”值，我们将其称为 I 0 。未受干扰值通常在数值上等于零，对于像操纵杆轴这样的居中双向控件，它位于 I min 和 I max 之间的中间值；对于像扳机这样的单向控件，它与 I min 重合。

遗憾的是，由于 HID 本质上是模拟设备，因此设备产生的电压噪声很大，我们观察到的实际输入可能会在 I 0 附近略有波动。解决此问题最常见的方法是在 I 0 附近引入一个小的死区。对于操纵杆，死区可以定义为 [I 0 − δ , I 0 + δ] ，对于扳机，死区可以定义为 [I 0 , I 0 + δ]。任何在死区内的输入值都会被简单地钳制在 I 0 上。死区必须足够宽，以

translate into smooth, pleasing, intuitive behaviors in-game. In addition, most engines introduce at least one additional level of indirection between the HID and the game in order to abstract HID inputs in various ways. For example, a button-mapping table might be used to translate raw button inputs into logical game actions, so that human players can reassign the buttons' functions as they see fit. In this section, we'll outline the typical requirements of a game engine HID system and then explore each one in some depth.

9.5.1 Typical Requirements

A game engine's HID system usually provides some or all of the following features:

- dead zones,
- analog signal filtering,
- event detection (e.g., button up, button down),
- detection of button *sequences* and multibutton combinations (known as *chords*),
- gesture detection,
- management of multiple HIDs for multiple players,
- multiplatform HID support,
- controller input remapping,
- context-sensitive inputs, and
- the ability to temporarily disable certain inputs.

9.5.2 Dead Zone

A joystick, thumb stick, shoulder trigger, or any other analog axis produces input values that range between a predefined minimum and maximum value, which we'll call I_{min} and I_{max}. When the control is not being touched, we would expect it to produce a steady and clear "undisturbed" value, which we'll call I_0. The undisturbed value is usually numerically equal to zero, and it either lies halfway between I_{min} and I_{max} for a centered, two-way control like a joystick axis, or it coincides with I_{min} for a one-way control like a trigger.

Unfortunately, because HIDs are analog devices by nature, the voltage produced by the device is noisy, and the actual inputs we observe may fluctuate slightly around I_0. The most common solution to this problem is to introduce a small *dead zone* around I_0. The dead zone might be defined as $[\, I_0 - \delta, I_0 + \delta \,]$ for a joy stick, or $[\, I_0, I_0 + \delta \,]$ for a trigger. Any input values that are within the dead zone are simply clamped to I_0. The dead zone must be wide enough to

考虑未受干扰的控制产生的最嘈杂的输入，但足够小，不会干扰玩家对 HID 响应的感觉。

9.5.3 模拟信号滤波

即使控制器不在盲区内，信号噪声也是一个问题。这种噪声有时会导致 HID 控制的游戏内行为显得不流畅或不自然。因此，许多游戏会过滤来自 HID 的原始输入。噪声信号的频率通常高于人类玩家产生的信号。因此，一种解决方案是在游戏使用原始输入数据之前，先将其通过一个简单的低通滤波器。

离散的一阶低通滤波器可以通过将当前未滤波的输入值与上一帧的滤波输入值相结合来实现。如果我们用时变函数 u(t) 表示未滤波输入序列，用 f(t) 表示滤波输入序列，其中 t 表示时间，那么我们可以写成

$$f(t) = (1-a)f(t-\Delta t) + au(t), \tag{9.1}$$

其中参数 α 由帧持续时间 Δ t 和滤波常数 RC（在传统模拟 RC 低通滤波器电路中，它只是电阻和电容的乘积）决定：

$$a = \frac{\Delta t}{RC + \Delta t}. \tag{9.2}$$

这可以用 C 或 C++ 轻松实现，如下所示，其中假设调用代码将跟踪上一帧的滤波输入，以便在后续帧中使用。更多信息，请参阅 http://en.wikipedia.org/wiki/Low-pass_filter。

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{
    F32 a = dt / (rc + dt);

    返回 (1 α) * lastFramesFilteredInput + α * unfilteredInput;

}
```

过滤 HID 输入数据的另一种方法是计算简单的移动平均值。例如，如果我们希望在 3/30 秒（3 帧）的间隔内对输入数据进行平均，我们只需将原始输入值存储在一个 3 元素循环中

account for the noisiest inputs generated by an undisturbed control, but small enough not to interfere with the player's sense of the HID's responsiveness.

9.5.3 Analog Signal Filtering

Signal noise is a problem even when the controls are not within their dead zones. This noise can sometimes cause the in-game behaviors controlled by the HID to appear jerky or unnatural. For this reason, many games *filter* the raw inputs coming from the HID. A noise signal is usually of a high frequency relative to the signal produced by the human player. Therefore, one solution is to pass the raw input data through a simple *low-pass filter*, prior to it being used by the game.

A discrete first-order low-pass filter can be implemented by combining the current unfiltered input value with last frame's filtered input. If we denote the sequence of unfiltered inputs by the time-varying function $u(t)$ and the filtered inputs by $f(t)$, where t denotes time, then we can write

$$f(t) = (1-a)f(t-\Delta t) + au(t), \tag{9.1}$$

where the parameter a is determined by the frame duration Δt and a filtering constant RC (which is just the product of the resistance and the capacitance in a traditional analog RC low-pass filter circuit):

$$a = \frac{\Delta t}{RC + \Delta t}. \tag{9.2}$$

This can be implemented trivially in C or C++ as follows, where it is assumed the calling code will keep track of the last frame's filtered input for use on the subsequent frame. For more information, see http://en.wikipedia.org/wiki/Low-pass_filter.

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{
    F32 a = dt / (rc + dt);

    return (1 - a) * lastFramesFilteredInput
          + a * unfilteredInput;
}
```

Another way to filter HID input data is to calculate a simple moving average. For example, if we wish to average the input data over a 3/30 second (3 frame) interval, we simply store the raw input values in a 3-element circular

缓冲区。过滤后的输入值等于该数组中任意时刻值的总和除以 3。实现这样的过滤器时需要考虑一些细节。例如，我们需要正确处理输入的前两帧，在此期间，包含 3 个元素的数组尚未填充有效数据。不过，实现起来并不特别复杂。以下代码展示了一种正确实现 N 个元素移动平均线的方法。

```
template< typename TYPE, int SIZE >
class MovingAverage
{
    TYPE  m_samples[SIZE];
    TYPE  m_sum;
    U32   m_curSample;
    U32   m_sampleCount;

public:
    MovingAverage() :
        m_sum(static_cast<TYPE>(0)),
        m_curSample(0),
        m_sampleCount(0)
    {
    }

    void addSample(TYPE data)
    {
        if (m_sampleCount == SIZE)
        {
            m_sum -= m_samples[m_curSample];
        }
        else
        {
            m_sampleCount++;
        }

        m_samples[m_curSample] = data;
        m_sum += data;
        m_curSample++;

        if (m_curSample >= SIZE)
        {
            m_curSample = 0;
        }
    }

    F32 getCurrentAverage() const
    {
```


buffer. The filtered input value is then the sum of the values in this array at any moment, divided by 3. There are a few minor details to account for when implementing such a filter. For example, we need to properly handle the first two frames of input, during which the 3-element array has not yet been filled with valid data. However, the implementation is not particularly complicated. The code below shows one way to properly implement an *N*-element moving average.

```
template< typename TYPE, int SIZE >
class MovingAverage
{
    TYPE  m_samples[SIZE];
    TYPE  m_sum;
    U32   m_curSample;
    U32   m_sampleCount;

public:
    MovingAverage() :
        m_sum(static_cast<TYPE>(0)),
        m_curSample(0),
        m_sampleCount(0)
    {
    }

    void addSample(TYPE data)
    {
        if (m_sampleCount == SIZE)
        {
            m_sum -= m_samples[m_curSample];
        }
        else
        {
            m_sampleCount++;
        }

        m_samples[m_curSample] = data;
        m_sum += data;
        m_curSample++;

        if (m_curSample >= SIZE)
        {
            m_curSample = 0;
        }
    }

    F32 getCurrentAverage() const
    {
```

```
            if (m_sampleCount != 0)
            {
                return static_cast<F32>(m_sum)
                     / static_cast<F32>(m_sampleCount);
            }
            return 0.0f;
        }
    };
```

9.5.4 检测输入事件

低级 HID 接口通常为游戏提供设备各种输入的当前状态。然而，游戏通常更关注检测事件，例如状态变化，而不仅仅是检查每帧的当前状态。最常见的 HID 事件可能是按钮按下（按下）和按钮弹起（释放），当然我们也可以检测其他类型的事件。

9.5.4.1 按钮向上和按钮向下

暂时假设按钮的输入位在未按下时为 0，按下时为 1。检测按钮状态变化的最简单方法是跟踪上一帧观察到的按钮状态位，并将其与本帧观察到的状态位进行比较。如果它们不同，我们就知道发生了事件。每个按钮的当前状态告诉我们事件是按钮弹起还是按钮按下。

我们可以使用简单的位运算符来检测按钮按下和按钮弹起事件。给定一个 32 位字 buttonStates ，其中包含最多 32 个按钮的当前状态位，我们需要生成两个新的 32 位字：一个用于按钮按下事件，我们将其称为 buttonDowns ；另一个用于按钮弹起事件，我们将其称为 buttonUps 。在这两种情况下，如果事件在本帧未发生，则每个按钮对应的位为 0；如果事件已发生，则为 1。为了实现这一点，我们还需要最后一帧的按钮状态，

`prevButtonStates`.
如果两个输入相同，则异或 (XOR) 运算符的结果为 0；如果不同，则结果为 1 。因此，如果我们将 XOR 运算符应用于上一帧和当前按钮状态字，则只有状态在上一帧和当前帧之间发生变化的按钮才会得到 1。要确定事件是按钮弹起还是按钮按下，我们需要查看每个按钮的当前状态。任何状态发生变化且当前处于按下状态的按钮都会生成按钮按下事件，反之亦然。以下代码应用了这些思想来生成两个按钮事件字：


```
            if (m_sampleCount != 0)
            {
                return static_cast<F32>(m_sum)
                     / static_cast<F32>(m_sampleCount);
            }
            return 0.0f;
        }
    };
```

9.5.4 Detecting Input Events

The low-level HID interface typically provides the game with the current states of the device's various inputs. However, games are often interested in detecting *events*, such as changes in state, rather than just inspecting the current state each frame. The most common HID events are probably button-down (pressed) and button-up (released), but of course we can detect other kinds of events as well.

9.5.4.1 Button Up and Button Down

Let's assume for the moment that our buttons' input bits are 0 when not pressed and 1 when pressed. The easiest way to detect a change in button state is to keep track of the buttons' state bits as observed last frame and compare them to the state bits observed this frame. If they differ, we know an event occurred. The current state of each button tells us whether the event is a button-up or a button-down.

We can use simple bit-wise operators to detect button-down and button-up events. Given a 32-bit word `buttonStates` containing the current state bits of up to 32 buttons, we want to generate two new 32-bit words: one for button-down events, which we'll call `buttonDowns`, and one for button-up events, which we'll call `buttonUps`. In both cases, the bit corresponding to each button will be 0 if the event has not occurred this frame and 1 if it has. To implement this, we also need the last frame's button states, `prevButtonStates`.

The exclusive OR (XOR) operator produces a 0 if its two inputs are identical and a 1 if they differ. So if we apply the XOR operator to the previous and current button state words, we'll get 1's only for buttons whose states have changed between the last frame and this frame. To determine whether the event is a button-up or a button-down, we need to look at the current state of each button. Any button whose state has changed that is currently down generates a button-down event, and vice versa for button-up events. The following code applies these ideas in order to generate our two button event words:

```
class ButtonState
{
    U32 m_buttonStates;      // current frame's button
                             // states
    U32 m_prevButtonStates;  // previous frame's states
    U32 m_buttonDowns;       // 1 = button pressed this
                             // frame
    U32 m_buttonUps;         // 1 = button released this
                             // frame

    void DetectButtonUpDownEvents()
    {
        // Assuming that m_buttonStates and
        // m_prevButtonStates are valid, generate
        // m_buttonDowns and m_buttonUps.

        // First determine which bits have changed via
        // XOR.
        U32 buttonChanges = m_buttonStates
                          ^ m_prevButtonStates;

        // Now use AND to mask off only the bits that
        // are DOWN.
        m_buttonDowns = buttonChanges & m_buttonStates;

        // Use AND-NOT to mask off only the bits that
        // are UP.
        m_buttonUps = buttonChanges & (~m_buttonStates);
    }

    // ...
};
```

9.5.4.2 和弦

和弦是指一组按钮，当它们同时按下时，会在游戏中产生独特的行为。以下是一些示例：

- 超级马里奥银河的启动屏幕要求您同时按下 Wiimote 上的 A 和 B 按钮才能开始新游戏。
- 同时按下 Wiimote 上的 1 和 2 按钮可使其进入蓝牙发现模式（无论您正在玩什么游戏）。


```
class ButtonState
{
    U32 m_buttonStates;      // current frame's button
                             // states
    U32 m_prevButtonStates;  // previous frame's states
    U32 m_buttonDowns;       // 1 = button pressed this
                             // frame
    U32 m_buttonUps;         // 1 = button released this
                             // frame

    void DetectButtonUpDownEvents()
    {
        // Assuming that m_buttonStates and
        // m_prevButtonStates are valid, generate
        // m_buttonDowns and m_buttonUps.

        // First determine which bits have changed via
        // XOR.
        U32 buttonChanges = m_buttonStates
                          ^ m_prevButtonStates;

        // Now use AND to mask off only the bits that
        // are DOWN.
        m_buttonDowns = buttonChanges & m_buttonStates;

        // Use AND-NOT to mask off only the bits that
        // are UP.
        m_buttonUps = buttonChanges & (~m_buttonStates);
    }

    // ...
};
```

9.5.4.2 Chords

A *chord* is a group of buttons that, when pressed at the same time, produce a unique behavior in the game. Here are a few examples:

- *Super Mario Galaxy*'s start-up screen requires you to press the A and B buttons on the Wiimote together in order to start a new game.
- Pressing the 1 and 2 buttons on the Wiimote at the same time puts it into Bluetooth discovery mode (no matter what game you're playing).

• 许多格斗游戏中的“擒抱”动作都是通过两个按钮的组合来触发的。

• 出于开发目的，在《神秘海域》中，同时按住 DualShock 3 手柄的左右扳机键，玩家角色就可以在游戏世界中任意飞行，并且碰撞检测会被关闭。（抱歉，这在正式版游戏中无效！）许多游戏都提供类似的作弊功能，以简化开发过程。（当然，它可能会被和弦触发，也可能不会被触发。）在 Quake 引擎中，它被称为无裁剪模式，因为角色的碰撞体积不会被裁剪到游戏世界的有效可玩区域内。其他引擎使用不同的术语。

检测和弦在原理上非常简单：我们仅观察两个或多个按钮的状态，并且只有当所有按钮都按下时才执行请求的操作。

然而，有一些细节需要考虑。首先，如果和弦中包含一个或多个在游戏中有其他用途的按钮，我们必须注意，不要同时执行单个按钮的操作和和弦的操作。这通常是通过在检测单个按钮按下时检查和弦中的其他按钮是否处于按下状态来实现的。

另一个美中不足的是，人类并非完美，他们经常会比其他人更早按下和弦中的一个或多个按钮。因此，我们的和弦检测代码必须具备鲁棒性，以应对我们可能在第 i 帧观察到一个或多个单独按钮，而在第 i + 1 帧（甚至更晚的几帧）观察到和弦的其余部分的可能性。有多种方法可以解决这个问题：

• 您可以设计按钮输入，使组合键始终执行单个按钮的操作以及一些附加操作。例如，如果按下 L1 键发射主武器，按下 L2 键投掷手榴弹，那么 L1 + L 2 组合键或许可以发射主武器、投掷手榴弹，并发出能量波，使这些武器造成的伤害加倍。这样，无论单个按钮是否在组合键之前被检测到，从玩家的角度来看，行为都是相同的。

• 您可以在单个按键按下事件被检测到和其“计入”有效游戏事件之间引入延迟。在延迟期间（例如 2 或 3 帧），如果检测到和弦，则该和弦优先于单个按键按下事件。这为人类玩家演奏和弦提供了一些余地。

• 按下按钮时您可以检测到和弦，但要等到再次释放按钮才能触发效果。

- The “grapple” move in many fighting games is triggered by a two-button combination.
- For development purposes, holding down both the left and right triggers on the DualShock 3 in *Uncharted* allows the player character to fly anywhere in the game world, with collisions turned off. (Sorry, this doesn’t work in the shipping game!) Many games have a cheat like this to make development easier. (It may or may not be triggered by a chord, of course.) It is called *no-clip mode* in the Quake engine, because the character’s collision volume is not *clipped* to the valid playable area of the world. Other engines use different terminology.

Detecting chords is quite simple in principle: We merely watch the states of two or more buttons and only perform the requested operation when *all* of them are down.

There are some subtleties to account for, however. For one thing, if the chord includes a button or buttons that have other purposes in the game, we must take care not to perform *both* the actions of the individual buttons and the action of chord when it is pressed. This is usually done by including a check that the other buttons in the chord are *not* down when detecting the individual button presses.

Another fly in the ointment is that humans aren’t perfect, and they often press one or more of the buttons in the chord slightly earlier than the rest. So our chord-detection code must be robust to the possibility that we’ll observe one or more individual buttons on frame i and the rest of the chord on frame $i + 1$ (or even multiple frames later). There are a number of ways to handle this:

- You can design your button inputs such that a chord always does the actions of the individual buttons *plus* some additional action. For example, if pressing L1 fires the primary weapon and L2 lobs a grenade, perhaps the L1 + L2 chord could fire the primary weapon, lob a grenade, *and* send out an energy wave that doubles the damage done by these weapons. That way, whether or not the individual buttons are detected before the chord or not, the behavior will be identical from the point of view of the player.
- You can introduce a delay between when an individual button-down event is seen and when it “counts” as a valid game event. During the delay period (say 2 or 3 frames), if a chord is detected, then it takes precedence over the individual button-down events. This gives the human player some leeway in performing the chord.
- You can detect the chord when the buttons are pressed, but wait to trigger the effect until the buttons are released again.

• 您可以立即开始单键移动并允许和弦移动抢占它。

9.5.4.3 序列和手势检测

在按钮实际按下和真正“计”为按下之间引入延迟的想法是手势检测的一个特例。手势是人类玩家在一段时间内通过 HID 执行的一系列动作。例如，在格斗游戏或摔跤游戏中，我们可能想要检测一系列按钮按下操作，例如 ABA。我们也可以将此想法扩展到非按钮输入。例如，ABA-左-右-左，其中后三个动作是游戏手柄上拇指杆之一的左右移动。通常，只有在某个最大时间范围内执行的序列或手势才被视为有效。因此，四分之一秒内的快速 ABA 可能“算数”，但一两秒内执行的慢速 ABA 可能不算数。

手势检测通常通过记录玩家执行的 HID 操作的简要历史记录来实现。检测到手势的第一个组成部分时，它会连同指示其发生时间的时间戳一起存储在历史记录缓冲区中。检测到每个后续组成部分时，会检查它与前一个组成部分之间的时间。如果它在允许的时间窗口内，也会将其添加到历史记录缓冲区中。如果整个序列在规定时间内完成（即历史记录缓冲区已填满），则会生成一个事件，告知游戏引擎的其余部分该手势已发生。但是，如果检测到任何无效的中间输入，或者手势的任何组成部分发生在其有效时间窗口之外，则整个历史记录缓冲区将被重置，玩家必须重新开始该手势。

让我们看三个具体的例子，这样我们才能真正理解它是如何工作的。

快速点击按钮

许多游戏要求用户快速点击按钮才能执行操作。按钮按下的频率可能会或可能不会转化为游戏中的某些量，例如玩家角色奔跑或执行其他动作的速度。频率通常也用于定义手势的有效性——如果频率低于某个最小值，则该手势不再被视为有效。

我们可以通过简单地跟踪上次观察到的按钮按下事件来检测按钮按下的频率。我们将其称为 T last 。频率 f 就是时间间隔的倒数

- You can begin the single-button move immediately and allow it to be preempted by the chord move.

9.5.4.3 Sequences and Gesture Detection

The idea of introducing a delay between when a button actually goes down and when it really “counts” as down is a special case of *gesture detection*. A gesture is a sequence of actions performed via a HID by the human player over a period of time. For example, in a fighting game or brawler, we might want to detect a *sequence* of button presses, such as A-B-A. We can extend this idea to non-button inputs as well. For example, A-B-A-Left-Right-Left, where the latter three actions are side-to-side motions of one of the thumb sticks on the game pad. Usually a sequence or gesture is only considered to be valid if it is performed within some maximum time frame. So a rapid A-B-A within a quarter of a second might “count,” but a slow A-B-A performed over a second or two might not.

Gesture detection is generally implemented by keeping a brief history of the HID actions performed by the player. When the first component of the gesture is detected, it is stored in the history buffer, along with a time stamp indicating when it occurred. As each subsequent component is detected, the time between it and the previous component is checked. If it is within the allowable time window, it too is added to the history buffer. If the entire sequence is completed within the allotted time (i.e., the history buffer is filled), an event is generated telling the rest of the game engine that the gesture has occurred. However, if any non-valid intervening inputs are detected, or if any component of the gesture occurs outside of its valid time window, the entire history buffer is reset and the player must start the gesture over again.

Let’s look at three concrete examples, so we can really understand how this works.

Rapid Button Tapping

Many games require the user to tap a button rapidly in order to perform an action. The frequency of the button presses may or may not translate into some quantity in the game, such as the speed with which the player character runs or performs some other action. The frequency is usually also used to define the validity of the gesture—if the frequency drops below some minimum value, the gesture is no longer considered valid.

We can detect the frequency of a button press by simply keeping track of the last time we saw a button-down event for the button in question. We’ll call this T_{last}. The frequency f is then just the inverse of the time interval between

按下 Δ T = T cur − T last 且 f = 1/ Δ T 。每次检测到新的按钮按下事件时，我们都会计算一个新的频率 f 。为了实现最小有效频率，我们只需检查 f 与最小频率 f min 的关系（或者我们可以直接检查 Δ T 与最大周期 Δ T max = 1/ f min 的关系）。如果满足此阈值，则更新 T last 的值，并且该手势被视为正在进行。如果不满足阈值，则不更新 T last 。在发生一对新的足够快速的按钮按下事件之前，该手势将被视为无效。以下伪代码说明了这一点：

按钮按下事件发生。以下伪代码演示了这一点：

```
class ButtonTapDetector
{
    U32  m_buttonMask; // which button to observe (bit
                       // mask)
    F32  m_dtMax;      // max allowed time between
                       // presses
    F32  m_tLast;      // last button-down event, in
                       // seconds

public:

    // Construct an object that detects rapid tapping of
    // the given button (identified by an index).
    ButtonTapDetector(U32 buttonId, F32 dtMax) :
        m_buttonMask(1U << buttonId),
        m_dtMax(dtMax),
        m_tLast(CurrentTime() - dtMax) // start out
                                       // invalid
    {
    }

    // Call this at any time to query whether or not
    // the gesture is currently being performed.
    bool IsGestureValid() const
    {
        F32 t = CurrentTime();
        F32 dt = t - m_tLast;
        return (dt < m_dtMax);
    }

    // Call this once per frame.
    void Update()
    {
        if (ButtonsJustWentDown(m_buttonMask))
        {
            m_tLast = CurrentTime();
        }
```


presses $\Delta T = T_{\text{cur}} - T_{\text{last}}$ and $f = 1/\Delta T$. Every time we detect a new button-down event, we calculate a new frequency f. To implement a minimum valid frequency, we simply check f against the minimum frequency f_{min} (or we can just check ΔT against the maximum period $\Delta T_{\text{max}} = 1/f_{\text{min}}$ directly). If this threshold is satisfied, we update the value of T_{last}, and the gesture is considered to be on-going. If the threshold is not satisfied, we simply don't update T_{last}. The gesture will be considered invalid until a new pair of rapid-enough button-down events occurs. This is illustrated by the following pseudocode:

```
class ButtonTapDetector
{
    U32  m_buttonMask; // which button to observe (bit
                       // mask)
    F32  m_dtMax;      // max allowed time between
                       // presses
    F32  m_tLast;      // last button-down event, in
                       // seconds

public:

    // Construct an object that detects rapid tapping of
    // the given button (identified by an index).
    ButtonTapDetector(U32 buttonId, F32 dtMax) :
        m_buttonMask(1U << buttonId),
        m_dtMax(dtMax),
        m_tLast(CurrentTime() - dtMax) // start out
                                       // invalid
    {
    }

    // Call this at any time to query whether or not
    // the gesture is currently being performed.
    bool IsGestureValid() const
    {
        F32 t = CurrentTime();
        F32 dt = t - m_tLast;
        return (dt < m_dtMax);
    }

    // Call this once per frame.
    void Update()
    {
        if (ButtonsJustWentDown(m_buttonMask))
        {
            m_tLast = CurrentTime();
        }
```

```
        }
    };
```

在上面的代码片段中，我们假设每个按钮都由一个唯一的 ID 标识。该 ID 实际上只是一个索引，范围从 0 到 N - 1（其中 N 是相关 HID 上的按钮数量）。我们将按钮 ID 转换为位掩码，方法是将无符号 1 位向左移动等于按钮索引的量（1U << buttonId）。如果给定位掩码指定的任何一个按钮刚刚在此帧中按下，则函数 ButtonsJustWentDown() 将返回非零值。在这里，我们只检查单个按钮按下事件，但我们可以并且稍后会使用相同的函数来检查多个同时按下的按钮按下事件。

多按钮序列

假设我们要检测最多一秒内完成的 ABA 序列。我们可以按如下方式检测此按钮序列：我们维护一个变量，用于跟踪我们当前正在寻找的序列中的哪个按钮。如果我们使用按钮 ID 数组定义该序列（例如，aButtons[3] = {A, B, A}），那么我们的变量只是该数组的索引 i。它最初初始化为序列中的第一个按钮，i = 0。我们还维护整个序列的开始时间 T start，就像我们在快速按下按钮的示例中所做的那样。

逻辑如下：每当我们看到与当前正在寻找的按钮匹配的按钮按下事件时，我们都会将其时间戳与整个序列的开始时间 T start 进行对比。如果它发生在有效的时间窗口内，我们将当前按钮推进到序列中的下一个按钮；对于序列中的第一个按钮（仅 i = 0），我们还会更新 T start 。如果我们看到与序列中的下一个按钮不匹配的按钮按下事件，或者时间增量过大，我们将按钮索引 i 重置回序列的开头，并将 T start 设置为某个无效值（例如 0）。下面的代码说明了这一点。

```
class ButtonSequenceDetector
{
    U32* m_aButtonIds;  // sequence of buttons to watch for
    U32  m_buttonCount; // number of buttons in sequence
    F32  m_dtMax;       // max time for entire sequence
    U32  m_iButton;     // next button to watch for in seq.
    F32  m_tStart;      // start time of sequence, in
                        // seconds
```



```
        }
    };
```

In the above code excerpt, we assume that each button is identified by a unique id. The id is really just an index, ranging from 0 to $N-1$ (where N is the number of buttons on the HID in question). We convert the button id to a bitmask by shifting an unsigned 1 bit to the left by an amount equaling the button's index (`1U << buttonId`). The function `ButtonsJustWentDown()` returns a nonzero value if *any one* of the buttons specified by the given bitmask just went down this frame. Here, we're only checking for a single button-down event, but we can and will use this same function later to check for multiple simultaneous button-down events.

Multibutton Sequence

Let's say we want to detect the sequence A-B-A, performed within at most one second. We can detect this button sequence as follows: We maintain a variable that tracks which button in the sequence we're currently looking for. If we define the sequence with an array of button ids (e.g., `aButtons[3] = {A, B, A}`), then our variable is just an index i into this array. It starts out initialized to the first button in the sequence, $i = 0$. We also maintain a start time for the entire sequence, T_{start}, much as we did in the rapid button-pressing example.

The logic goes like this: Whenever we see a button-down event that matches the button we're currently looking for, we check its time stamp against the start time of the entire sequence, T_{start}. If it occurred within the valid time window, we advance the current button to the next button in the sequence; for the first button in the sequence only ($i = 0$), we also update T_{start}. If we see a button-down event that doesn't match the next button in the sequence, or if the time delta has grown too large, we reset the button index i back to the beginning of the sequence and set T_{start} to some invalid value (such as 0). This is illustrated by the code below.

```
class ButtonSequenceDetector
{
    U32* m_aButtonIds;  // sequence of buttons to watch for
    U32  m_buttonCount; // number of buttons in sequence
    F32  m_dtMax;       // max time for entire sequence
    U32  m_iButton;     // next button to watch for in seq.
    F32  m_tStart;      // start time of sequence, in
                        // seconds
```

```
public:

    // Construct an object that detects the given button
    // sequence. When the sequence is successfully
    // detected, the given event is broadcast so that the
    // rest of the game can respond in an appropriate way.

    ButtonSequenceDetector(U32* aButtonIds,
                           U32 buttonCount,
                           F32 dtMax,
                           EventId eventIdToSend) :
        m_aButtonIds(aButtonIds),
        m_buttonCount(buttonCount),
        m_dtMax(dtMax),
        m_eventId(eventIdToSend), // event to send when
                                  // complete
        m_iButton(0),             // start of sequence
        m_tStart(0)               // initial value
                                  // irrelevant
    {
    }

    // Call this once per frame.
    void Update()
    {
        ASSERT(m_iButton < m_buttonCount);

        // Determine which button we're expecting next, as
        // a bitmask (shift a 1 up to the correct bit
        // index).

        U32 buttonMask = (1U << m_aButtonId[m_iButton]);

        // If any button OTHER than the expected button
        // just went down, invalidate the sequence. (Use
        // the bitwise NOT operator to check for all other
        // buttons.)

        if (ButtonsJustWentDown(~buttonMask))
        {
            m_iButton = 0; // reset
        }

        // Otherwise, if the expected button just went
        // down, check dt and update our state appropriately.

        else if (ButtonsJustWentDown(buttonMask))
        {
```



```
public:

    // Construct an object that detects the given button
    // sequence. When the sequence is successfully
    // detected, the given event is broadcast so that the
    // rest of the game can respond in an appropriate way.

    ButtonSequenceDetector(U32* aButtonIds,
                           U32 buttonCount,
                           F32 dtMax,
                           EventId eventIdToSend) :
        m_aButtonIds(aButtonIds),
        m_buttonCount(buttonCount),
        m_dtMax(dtMax),
        m_eventId(eventIdToSend), // event to send when
                                  // complete
        m_iButton(0),             // start of sequence
        m_tStart(0)               // initial value
                                  // irrelevant
    {
    }

    // Call this once per frame.
    void Update()
    {
        ASSERT(m_iButton < m_buttonCount);

        // Determine which button we're expecting next, as
        // a bitmask (shift a 1 up to the correct bit
        // index).

        U32 buttonMask = (1U << m_aButtonId[m_iButton]);

        // If any button OTHER than the expected button
        // just went down, invalidate the sequence. (Use
        // the bitwise NOT operator to check for all other
        // buttons.)

        if (ButtonsJustWentDown(~buttonMask))
        {
            m_iButton = 0; // reset
        }

        // Otherwise, if the expected button just went
        // down, check dt and update our state appropriately.

        else if (ButtonsJustWentDown(buttonMask))
        {
```

```
                if (m_iButton == 0)
                {
                    // This is the first button in the
                    // sequence.
                    m_tStart = CurrentTime();
                    m_iButton++; // advance to next button
                }
                else
                {
                    F32 dt = CurrentTime() - m_tStart;

                    if (dt < m_dtMax)
                    {
                        // Sequence is still valid.

                        m_iButton++; // advance to next button

                        // Is the sequence complete?
                        if (m_iButton == m_buttonCount)
                        {
                            BroadcastEvent(m_eventId);
                            m_iButton = 0; // reset
                        }
                    }
                    else
                    {
                        // Sorry, not fast enough.
                        m_iButton = 0; // reset
                    }
                }
            }
        }
    }
};
```

图 9.14.通过将摇杆输入的 2D 范围划分为象限来检测摇杆的圆周旋转。


```
                if (m_iButton == 0)
                {
                    // This is the first button in the
                    // sequence.
                    m_tStart = CurrentTime();
                    m_iButton++; // advance to next button
                }
                else
                {
                    F32 dt = CurrentTime() - m_tStart;

                    if (dt < m_dtMax)
                    {
                        // Sequence is still valid.

                        m_iButton++; // advance to next button

                        // Is the sequence complete?
                        if (m_iButton == m_buttonCount)
                        {
                            BroadcastEvent(m_eventId);
                            m_iButton = 0; // reset
                        }
                    }
                    else
                    {
                        // Sorry, not fast enough.
                        m_iButton = 0; // reset
                    }
                }
            }
        }
    }
};
```

Figure 9.14. Detecting circular rotations of the stick by dividing the 2D range of stick inputs into quadrants.

拇指摇杆旋转

举一个更复杂手势的例子，让我们看看如何检测玩家何时顺时针旋转左摇杆。我们可以通过将摇杆可能的二维位置范围划分为几个象限来轻松检测这一点，如图 9.14 所示。顺时针旋转时，摇杆会依次经过左上象限、右上象限、右下象限，最后到达左下象限。我们可以将每种情况都视为按钮按下，并使用略微修改的上述序列检测代码来检测完整的旋转。我们将此留给读者作为练习。试试看！

9.5.5 管理多个播放器的多个 HID

大多数游戏机允许连接两个或多个 HID 设备，以实现多人游戏。引擎必须跟踪当前连接的设备，并将每个设备的输入路由到游戏中相应的玩家。这意味着我们需要某种方式将控制器映射到玩家。这可以简单到只是控制器索引和玩家索引之间的一对一映射，也可以更复杂一些，例如在用户点击“开始”按钮时将控制器分配给玩家。

即使在只有一个 HID 的单人游戏中，引擎也需要能够应对各种异常情况，例如控制器意外拔出或电池耗尽。当控制器连接断开时，大多数游戏会暂停游戏，显示一条消息并等待控制器重新连接。一些多人游戏会暂停或暂时移除与已移除控制器对应的虚拟形象，但允许其他玩家继续游戏；当控制器重新连接时，已移除/暂停的虚拟形象可能会重新激活。

在配备电池供电 HID 的系统上，游戏或操作系统负责检测电池电量不足的情况。作为响应，通常会以某种方式向玩家发出警告，例如通过不显眼的屏幕消息和/或音效。

9.5.6 跨平台 HID 系统

许多游戏引擎都是跨平台的。在这样的引擎中处理 HID 输入和输出的一种方法是在代码中散布条件编译指令，只要与 HID 发生交互，如下所示。这显然不是一个理想的解决方案，但它确实有效。

```
#if TARGET_XBOX360
if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
```


Thumb Stick Rotation

As an example of a more complex gesture, let's see how we might detect when the player is rotating the left thumb stick in a clockwise circle. We can detect this quite easily by dividing the two-dimensional range of possible stick positions into quadrants, as shown in Figure 9.14. In a clockwise rotation, the stick passes through the upper-left quadrant, then the upper-right, then the lower-right and finally the lower-left. We can treat each of these cases like a button press and detect a full rotation with a slightly modified version of the sequence detection code shown above. We'll leave this one as an exercise for the reader. Try it!

9.5.5 Managing Multiple HIDs for Multiple Players

Most game machines allow two or more HIDs to be attached for multiplayer games. The engine must keep track of which devices are currently attached and route each one's inputs to the appropriate player in the game. This implies that we need some way of mapping controllers to players. This might be as simple as a one-to-one mapping between controller index and player index, or it might be something more sophisticated, such as assigning controllers to players at the time the user hits the Start button.

Even in a single-player game with only one HID, the engine needs to be robust to various exceptional conditions, such as the controller being accidentally unplugged or running out of batteries. When a controller's connection is lost, most games pause gameplay, display a message and wait for the controller to be reconnected. Some multiplayer games suspend or temporarily remove the avatar corresponding to a removed controller, but allow the other players to continue playing the game; the removed/suspended avatar might reactivate when the controller is reconnected.

On systems with battery-operated HIDs, the game or the operating system is responsible for detecting low-battery conditions. In response, the player is usually warned in some way, for example via an unobtrusive on-screen message and/or a sound effect.

9.5.6 Cross-Platform HID Systems

Many game engines are cross-platform. One way to handle HID inputs and outputs in such an engine would be to sprinkle conditional compilation directives all over the code, wherever interactions with the HID take place, as shown below. This is clearly not an ideal solution, but it does work.

```
#if TARGET_XBOX360
if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
```

```
#elif TARGET_PS3
if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
    // do something...
}
```

更好的解决方案是提供某种硬件抽象层，从而将游戏代码与硬件特定的细节隔离开来。

如果幸运的话，我们可以通过合理选择抽象的按钮和轴 ID，来抽象不同平台上 HID 之间的大部分差异。例如，如果我们的游戏要在 Xbox 360 和 PS3 上发行，那么这两个手柄上的控件布局（按钮、轴和扳机键）几乎完全相同。每个平台上的控件 ID 各不相同，但我们可以很容易地设计出一个通用的控件 ID，涵盖这两种类型的手柄。例如：

```
enum AbstractControlIndex
{
    // Start and back buttons
    AINDEX_START,         // Xbox 360 Start, PS3 Start
    AINDEX_BACK_SELECT,   // Xbox 360 Back, PS3 Select

    // Left D-pad
    AINDEX_LPAD_DOWN,
    AINDEX_LPAD_UP,
    AINDEX_LPAD_LEFT,
    AINDEX_LPAD_RIGHT,

    // Right "pad" of four buttons
    AINDEX_RPAD_DOWN,      // Xbox 360 A, PS3 X
    AINDEX_RPAD_UP,        // Xbox 360 Y, PS3 Triangle
    AINDEX_RPAD_LEFT,      // Xbox 360 X, PS3 Square
    AINDEX_RPAD_RIGHT,     // Xbox 360 B, PS3 Circle

    // Left and right thumb stick buttons
    AINDEX_LSTICK_BUTTON,  // Xbox 360 LThumb, PS3 L3,
                           // Xbox white
    AINDEX_RSTICK_BUTTON,  // Xbox 360 RThumb, PS3 R3,
                           // Xbox black

    // Left and right shoulder buttons
    AINDEX_LSHOULDER,      // Xbox 360 L shoulder, PS3 L1
    AINDEX_RSHOULDER,      // Xbox 360 R shoulder, PS3 R1
```



```
#elif TARGET_PS3
if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
    // do something...
}
```

A better solution is to provide some kind of hardware abstraction layer, thereby insulating the game code from hardware-specific details.

If we're lucky, we can abstract most of the differences beween the HIDs on the different platforms by a judicious choice of abstract button and axis ids. For example, if our game is to ship on Xbox 360 and PS3, the layout of the controls (buttons, axes and triggers) on these two joypads are almost identical. The controls have different ids on each platform, but we can come up with generic control ids that cover both types of joypad quite easily. For example:

```
enum AbstractControlIndex
{
    // Start and back buttons
    AINDEX_START,         // Xbox 360 Start, PS3 Start
    AINDEX_BACK_SELECT,   // Xbox 360 Back, PS3 Select

    // Left D-pad
    AINDEX_LPAD_DOWN,
    AINDEX_LPAD_UP,
    AINDEX_LPAD_LEFT,
    AINDEX_LPAD_RIGHT,

    // Right "pad" of four buttons
    AINDEX_RPAD_DOWN,      // Xbox 360 A, PS3 X
    AINDEX_RPAD_UP,        // Xbox 360 Y, PS3 Triangle
    AINDEX_RPAD_LEFT,      // Xbox 360 X, PS3 Square
    AINDEX_RPAD_RIGHT,     // Xbox 360 B, PS3 Circle

    // Left and right thumb stick buttons
    AINDEX_LSTICK_BUTTON,  // Xbox 360 LThumb, PS3 L3,
                           // Xbox white
    AINDEX_RSTICK_BUTTON,  // Xbox 360 RThumb, PS3 R3,
                           // Xbox black

    // Left and right shoulder buttons
    AINDEX_LSHOULDER,      // Xbox 360 L shoulder, PS3 L1
    AINDEX_RSHOULDER,      // Xbox 360 R shoulder, PS3 R1
```

```
        // Left thumb stick axes
        AINDEX_LSTICK_X,
        AINDEX_LSTICK_Y,

        // Right thumb stick axes
        AINDEX_RSTICK_X,
        AINDEX_RSTICK_Y,

        // Left and right trigger axes
        AINDEX_LTRIGGER,     // Xbox 360 -Z, PS3 L2
        AINDEX_RTRIGGER,     // Xbox 360 +Z, PS3 R2
    };
```

我们的抽象层可以将当前目标硬件上的原始控件 ID 转换为抽象控件索引。例如，每当我们将按钮状态读入 32 位字时，我们都可以执行位交换操作，将位重新排列成正确的顺序，以对应我们的抽象索引。模拟输入也可以同样地被重新排列成正确的顺序。

在执行物理控件和抽象控件之间的映射时，我们有时需要稍微巧妙一些。例如，在 Xbox 上，左右扳机键充当单个轴，按下左扳机键时产生负值，两个扳机键均未按下时产生零值，按下右扳机键时产生正值。为了匹配 PlayStation DualShock 控制器的行为，我们可能需要在 Xbox 上将此轴分成两个不同的轴，并适当缩放值，使所有平台上的有效值范围相同。

这当然不是在多平台引擎中处理 HID I/O 的唯一方法。我们或许可以采用更实用的方法，例如，根据抽象控件在游戏中的功能来命名，而不是根据它们在游戏手柄上的物理位置来命名。我们或许可以引入更高级的函数来检测抽象手势，并在每个平台上编写自定义的检测代码，或者干脆硬着头皮为所有需要 HID I/O 的游戏代码编写平台特定的版本。可能性有很多，但几乎所有跨平台游戏引擎都以某种方式将游戏与硬件细节隔离开来。

9.5.7 输入重映射

许多游戏允许玩家在一定程度上选择物理 HID 上各种控件的功能。在主机游戏中，一个常见的选项是使用右拇指摇杆的垂直轴来控制视角。有些人喜欢向前推摇杆来调整视角角度。


```
        // Left thumb stick axes
        AINDEX_LSTICK_X,
        AINDEX_LSTICK_Y,

        // Right thumb stick axes
        AINDEX_RSTICK_X,
        AINDEX_RSTICK_Y,

        // Left and right trigger axes
        AINDEX_LTRIGGER,     // Xbox 360 -Z, PS3 L2
        AINDEX_RTRIGGER,     // Xbox 360 +Z, PS3 R2
    };
```

Our abstraction layer can translate between the raw control ids on the current target hardware into our abstract control indices. For example, whenever we read the state of the buttons into a 32-bit word, we can perform a bit-swizzling operation that rearranges the bits into the proper order to correspond to our abstract indices. Analog inputs can likewise be shuffled around into the proper order.

In performing the mapping between physical and abstract controls, we'll sometimes need to get a bit clever. For example, on the Xbox, the left and right triggers act as a single axis, producing negative values when the left trigger is pressed, zero when neither is trigger is pressed, and positive values when the right trigger is pressed. To match the behavior of the PlayStation's DualShock controller, we might want to separate this axis into two distinct axes on the Xbox, scaling the values appropriately so the range of valid values is the same on all platforms.

This is certainly not the only way to handle HID I/O in a multiplatform engine. We might want to take a more functional approach, for example, by naming our abstract controls according to their function in the game, rather than their physical locations on the joypad. We might introduce higher-level functions that detect abstract gestures, with custom detection code on each platform, or we might just bite the bullet and write platform-specific versions of all of the game code that requires HID I/O. The possibilities are numerous, but virtually all cross-platform game engines insulate the game from hardware details in *some* manner.

9.5.7 Input Remapping

Many games allow the player some degree of choice with regard to the functionality of the various controls on the physical HID. A common option is the sense of the vertical axis of the right thumb stick for camera control in a console game. Some folks like to push forward on the stick to angle the camera

有些游戏喜欢向上倾斜，而有些游戏则喜欢反向控制方案，即向后拉动摇杆会使视角向上倾斜（很像飞机的操纵杆）。其他游戏允许玩家在两个或多个预定义的按键映射之间进行选择。一些电脑游戏允许用户完全控制键盘上各个按键、鼠标按键和鼠标滚轮的功能，还可以选择鼠标两个轴的各种控制方案。

为了实现这一点，我们引用了我一位老教授——滑铁卢大学杰伊·布莱克教授——最喜欢的一句话："计算机科学中的每个问题都可以通过一定程度的间接来解决。"我们为游戏中的每个函数分配一个唯一的ID，然后提供一个简单的表格，将每个物理或抽象控件的索引映射到游戏中的逻辑函数。每当游戏需要确定是否应该激活某个特定的逻辑游戏函数时，它都会在表中查找相应的抽象或物理控件ID，然后读取该控件的状态。要更改映射，我们可以批量替换整个表格，也可以允许用户编辑表中的单个条目。

我们在这里忽略了一些细节。首先，不同的控件会产生不同类型的输入。模拟轴可能产生从 $-32{,}768$ 到 $32{,}767$ 的值，或者从 0 到 255 或其他范围的值。HID上所有数字按钮的状态通常打包成一个机器字。因此，我们必须小心，只允许有意义的控件映射。例如，我们不能将按钮用作需要轴的逻辑游戏功能的控件。解决此问题的一种方法是规范化所有输入。例如，我们可以将所有模拟轴和按钮的输入重新缩放到 $[0, 1]$ 范围内。这并不像您最初想象的那么有用，因为某些轴本质上是双向的（例如操纵杆），而其他轴是单向的（例如扳机）。但是，如果我们将控件分组为几个类，我们可以规范化这些类中的输入，并仅允许在兼容的类内重新映射。标准控制台游戏手柄的合理类别集及其规范化的输入值可能是：

- 数字按钮。状态被打包成一个 32 位字，每个按钮一位。
- 单向绝对轴（例如，触发器、模拟按钮）。生成 $[0, 1]$ 范围内的浮点输入值。
- 双向绝对轴（例如，操纵杆）。生成 $[-1, 1]$ 范围内的浮点输入值。
- 相对轴（例如，鼠标轴、滚轮、轨迹球）。生成 $[-1, 1]$ 范围内的浮点输入值，其中 ± 1 表示单个游戏帧内可能的最大相对偏移（即，在 1/30 或 1/60 秒的时间段内）。

up, while others like an inverted control scheme, where pulling back on the stick angles the camera up (much like an airplane control stick). Other games allow the player to select between two or more predefined button mappings. Some PC games allow the user full control over the functions of individual keys on the keyboard, the mouse buttons and the mouse wheel, plus a choice between various control schemes for the two mouse axes.

To implement this, we turn to a favorite saying of an old professor of mine, Professor Jay Black of the University of Waterloo, "Every problem in computer science can be solved with a level of indirection." We assign each function in the game a unique id and then provide a simple table, which maps each physical or abstract control index to a logical function in the game. Whenever the game wishes to determine whether a particular logical game function should be activated, it looks up the corresponding abstract or physical control id in the table and then reads the state of that control. To change the mapping, we can either swap out the entire table wholesale, or we can allow the user to edit individual entries in the table.

We're glossing over a few details here. For one thing, different controls produce different kinds of inputs. Analog axes may produce values ranging from $-32{,}768$ to $32{,}767$, or from 0 to 255, or some other range. The states of all the digital buttons on a HID are usually packed into a single machine word. Therefore, we must be careful to only permit control mappings that make sense. We cannot use a button as the control for a logical game function that requires an axis, for example. One way around this problem is to normalize all of the inputs. For example, we could re-scale the inputs from all analog axes and buttons into the range $[0, 1]$. This isn't quite as helpful as you might at first think, because some axes are inherently bidirectional (like a joy stick) while others are unidirectional (like a trigger). But, if we group our controls into a few classes, we can normalize the inputs within those classes and permit remapping only within compatible classes. A reasonable set of classes for a standard console joypad and their normalized input values might be:

- *Digital buttons*. States are packed into a 32-bit word, one bit per button.
- *Unidirectional absolute axes (e.g., triggers, analog buttons)*. Produce floating-point input values in the range $[0, 1]$.
- *Bidirectional absolute axes (e.g., joy sticks)*. Produce floating-point input values in the range $[-1, 1]$.
- *Relative axes (e.g., mouse axes, wheels, track balls)*. Produce floating-point input values in the range $[-1, 1]$, where ± 1 represents the maximum relative offset possible within a single game frame (i.e., during a period of 1/30 or 1/60 of a second).

9.5.8 上下文相关控件

在许多游戏中，单个物理控件可能根据具体情况具有不同的功能。一个简单的例子就是无处不在的“使用”按钮。如果站在门前按下“使用”按钮，角色可能会打开门。如果站在物体附近按下该按钮，则可能导致玩家角色拾起该物体等等。另一个常见的例子是模态控制方案。当玩家四处走动时，控件用于导航和控制摄像机。当玩家驾驶车辆时，控件用于操纵车辆，并且摄像机控制也可能不同。

通过状态机实现上下文相关控制相当简单。根据我们所处的状态，特定的 HID 控件可能具有不同的用途。棘手的部分在于确定要处于哪种状态。例如，当按下上下文相关的“使用”按钮时，玩家可能站在武器和医疗包等距的位置，面向它们之间的中心点。在这种情况下，我们应该使用哪个对象？有些游戏会实现优先级系统来打破这种平局。也许武器的重量比医疗包重，所以在这个例子中它会“获胜”。实现上下文相关控制并非高深莫测，但总是需要大量的反复试验才能让它的感觉和行为恰到好处。计划进行大量的迭代和重点测试！

另一个相关概念是控制所有权。HID 上的某些控件可能由游戏的不同部分“拥有”。例如，一些输入用于玩家控制，一些用于摄像头控制，还有一些供游戏的包装器和菜单系统使用（暂停游戏等等）。一些游戏引擎引入了逻辑设备的概念，它仅由物理设备上的输入子集组成。一个逻辑设备可能用于玩家控制，而另一个逻辑设备由摄像头系统使用，还有一个逻辑设备由菜单系统使用。

9.5.9 禁用输入

在大多数游戏中，有时需要禁止玩家控制其角色。例如，当玩家角色参与游戏过场动画时，我们可能希望暂时禁用所有玩家控制；或者当玩家穿过狭窄的门口时，我们可能希望暂时禁用自由视角旋转。

一种相当粗暴的方法是使用位掩码来禁用输入设备上的单个控件。每当读取控件时，都会检查禁用掩码，如果设置了相应的位，则返回中性值或零值。

9.5.8 Context-Sensitive Controls

In many games, a single physical control can have different functions, depending on context. A simple example is the ubiquitous “use” button. If pressed while standing in front of a door, the “use” button might cause the character to open the door. If it is pressed while standing near an object, it might cause the player character to pick up the object and so on. Another common example is a modal control scheme. When the player is walking around, the controls are used to navigate and control the camera. When the player is riding a vehicle, the controls are used to steer the vehicle, and the camera controls might be different as well.

Context-sensitive controls are reasonably straightforward to implement via a state machine. Depending on what state we're in, a particular HID control may have a different purpose. The tricky part is deciding what state to be in. For example, when the context-sensitive “use” button is pressed, the player might be standing at a point equidistant between a weapon and a health pack, facing the center point between them. Which object do we use in this case? Some games implement a priority system to break ties like this. Perhaps the weapon has a higher weight than the health pack, so it would “win” in this example. Implementing context-sensitive controls isn't rocket science, but it invariably requires lots of trial and error to get it feeling and behaving just right. Plan on lots of iteration and focus testing!

Another related concept is that of *control ownership*. Certain controls on the HID might be “owned” by different parts of the game. For example, some inputs are for player control, some for camera control and still others are for use by the game's wrapper and menu system (pausing the game, and so on). Some game engines introduce the concept of a logical device, which is composed of only a subset of the inputs on the physical device. One logical device might be used for player control, while another is used by the camera system, and another by the menu system.

9.5.9 Disabling Inputs

In most games, it is sometimes necessary to disallow the player from controlling his or her character. For example, when the player character is involved in an in-game cinematic, we might want to disable all player controls temporarily; or when the player is walking through a narrow doorway, we might want to temporarily disable free camera rotation.

One rather heavy-handed approach is to use a bitmask to disable individual controls on the input device itself. Whenever the control is read, the disable mask is checked, and if the corresponding bit is set, a neutral or zero value is

返回的值，而不是从设备读取的实际值。然而，禁用控件时必须格外小心。如果我们忘记重置禁用掩码，游戏可能会陷入玩家永远失去所有控制权并必须重新开始游戏的状态。仔细检查逻辑非常重要，同时，最好添加一些故障安全机制，以确保在某些关键时刻（例如玩家死亡并重生时）清除禁用掩码。

禁用 HID 输入会屏蔽所有可能的客户端，这可能会造成过度限制。更好的方法可能是将禁用特定玩家动作或摄像机行为的逻辑直接放入玩家或摄像机代码中。这样，例如，如果摄像机决定忽略右拇指摇杆的偏转，其他游戏引擎系统仍然可以自由地读取该摇杆的状态以用于其他目的。

9.6 人机接口设备实践

正确流畅地处理人机界面设备是任何优秀游戏的重要组成部分。从概念上讲，HID 似乎非常简单。然而，其中可能存在不少“陷阱”，包括不同物理输入设备之间的差异、低通滤波器的正确实现、控制方案映射的无错误处理、游戏手柄震动的恰到好处的“手感”、游戏主机制造商通过其技术要求清单 (TRC) 施加的限制等等。游戏团队应该投入大量的时间和工程资源来精心完整地实现人机界面设备系统。这一点至关重要，因为 HID 系统构成了游戏最宝贵资源——玩家机制——的基础。

returned instead of the actual value read from the device. We must be particularly cautious when disabling controls, however. If we forget to reset the disable mask, the game can get itself into a state where the player loses all control forever and must restart the game. It's important to check our logic carefully, and it's also a good idea to put in some fail-safe mechanisms to ensure that the disable mask is cleared at certain key times, such as whenever the player dies and re-spawns.

Disabling a HID input masks it for all possible clients, which can be overly limiting. A better approach is probably to put the logic for disabling specific player actions or camera behaviors directly into the player or camera code itself. That way, if the camera decides to ignore the deflection of the right thumb stick, for example, other game engine systems still have the freedom to read the state of that stick for other purposes.

9.6 Human Interface Devices in Practice

Correct and smooth handling of human interface devices is an important part of any good game. Conceptually speaking, HIDs may seem quite straightforward. However, there can be quite a few “gotchas” to deal with, including variations between different physical input devices, proper implementation of low-pass filtering, bug-free handling of control scheme mappings, achieving just the right “feel” in your joypad rumble, limitations imposed by console manufacturers via their technical requirements checklists (TRCs), and the list goes on. A game team should expect to devote a nontrivial amount of time and engineering bandwidth to a careful and complete implementation of the human interface device system. This is extremely important because the HID system forms the underpinnings of your game's most precious resource—its player mechanics.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

10
调试和开发工具

开发游戏软件是一项复杂、精细、数学密集且容易出错的工作。因此，几乎每个专业游戏团队都会为自己构建一套工具，以使游戏开发过程更轻松、更不容易出错，这不足为奇。在本章中，我们将介绍专业级游戏引擎中最常见的开发和调试工具。

10.1 日志记录和跟踪

还记得你用 BASIC 或 Pascal 编写的第一个程序吗？（好吧，也许你不记得了。如果你比我年轻很多——而且很有可能——你可能用 Java、Python 或 Lua 编写了你的□□第一个程序。）无论如何，你可能还记得当时是如何调试程序的。你知道吗，那时候你以为调试器就是那种发光的蓝色灭虫器？你可能用过 print 语句来输出程序的内部状态。C/C++ 程序员称之为 printf 调试（源自 C 标准库函数 printf()）。

事实证明，printf 调试仍然是一件非常有效的事情——即使你知道调试器不是用来煎倒霉的昆虫的设备

10
Tools for Debugging and Development

Developing game software is a complex, intricate, math-intensive and error-prone business. So it should be no surprise that virtually every professional game team builds a suite of tools for themselves, in order to make the game development process easier and less error-prone. In this chapter, we'll take a look at the development and debugging tools most often found in professional-grade game engines.

10.1 Logging and Tracing

Remember when you wrote your first program in BASIC or Pascal? (OK, maybe you don't. If you're significantly younger than me—and there's a pretty good chance of that—you probably wrote your first program in Java, or maybe Python or Lua.) In any case, you probably remember how you debugged your programs back then. You know, back when you thought a *debugger* was one of those glowing blue insect zapper things? You probably used *print statements* to dump out the internal state of your program. C/C++ programmers call this *printf debugging* (after the C standard library function, `printf()`).

It turns out that *printf debugging* is still a perfectly valid thing to do—even *if* you know that a debugger isn't a device for frying hapless insects at

夜晚。尤其是在实时编程中，使用断点和监视窗口来追踪某些类型的错误可能非常困难。有些错误与时间相关：它们仅在程序全速运行时发生。其他错误是由复杂的事件序列引起的，这些事件序列太长且错综复杂，无法逐一手动追踪。在这些情况下，最强大的调试工具通常是一系列打印语句。

每个游戏平台都有某种控制台或电传打字机 (TTY) 输出设备。以下是一些示例：

- 在 Linux 或 Win32 下运行的用 C/C++ 编写的控制台应用程序中，您可以通过 printf()、fprintf() 或 C++ 标准库的 iostream 接口打印到 stdout 或 stderr，从而在控制台中生成输出。
- 遗憾的是，如果你的游戏是在 Win32 下构建的窗口应用程序，printf() 和 iostream 将无法工作，因为没有控制台来显示输出。但是，如果你在 Visual Studio 调试器下运行游戏，它会提供一个调试控制台，你可以通过 Win32 函数 OutputDebugString() 进行打印。
- 在 PlayStation 3 和 PlayStation 4 上，一个名为“目标管理器”（或 PS4 上的“PlayStation Neighborhood”）的应用程序会在您的电脑上运行，允许您在主机上启动程序。“目标管理器”包含一组 TTY 输出窗口，游戏引擎可以将消息打印到这些窗口。

因此，为了调试目的而打印信息几乎总是像在整个代码中添加 printf() 调用一样简单。然而，大多数游戏引擎的功能远不止于此。在接下来的章节中，我们将探讨大多数游戏引擎提供的打印功能。

10.1.1 使用 OutputDebugString() 格式化输出

Windows SDK 函数 OutputDebugString() 非常适合将调试信息打印到 Visual Studio 的“调试输出”窗口。然而，与 printf() 不同，OutputDebugString() 不支持格式化输出——它只能以数组的形式打印原始字符串。因此，大多数 Windows 游戏引擎将其包装在一个自定义函数中，如下所示：

```
#include <stdio.h>     // for va_list et al

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1
#endif
#include <windows.h>   // for OutputDebugString()
```


night. Especially in real-time programming, it can be difficult to trace certain kinds of bugs using breakpoints and watch windows. Some bugs are timing-dependent: they only happen when the program is running at full speed. Other bugs are caused by a complex sequence of events too long and intricate to trace manually one-by-one. In these situations, the most powerful debugging tool is often a sequence of print statements.

Every game platform has some kind of console or teletype (TTY) output device. Here are some examples:

- In a console application written in C/C++, running under Linux or Win32, you can produce output in the console by printing to `stdout` or `stderr` via `printf()`, `fprintf()` or the C++ standard library's `iostream` interface.
- Unfortunately, `printf()` and `iostream` don't work if your game is built as a windowed application under Win32, because there's no console in which to display the output. However, if you're running under the Visual Studio debugger, it provides a debug console to which you can print via the Win32 function `OutputDebugString()`.
- On the PlayStation 3 and PlayStation 4, an application known as the Target Manager (or PlayStation Neighborhood on the PS4) runs on your PC and allows you to launch programs on the console. The Target Manager includes a set of TTY output windows to which messages can be printed by the game engine.

So printing out information for debugging purposes is almost always as easy as adding calls to `printf()` throughout your code. However, most game engines go a bit farther than this. In the following sections, we'll investigate the kinds of printing facilities most game engines provide.

10.1.1 Formatted Output with `OutputDebugString()`

The Windows SDK function `OutputDebugString()` is great for printing debugging information to Visual Studio's Debug Output window. However, unlike `printf()`, `OutputDebugString()` does not support *formatted* output—it can only print raw strings in the form of arrays. For this reason, most Windows game engines wrap it in a custom function, like this:

```
#include <stdio.h>     // for va_list et al

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1
#endif
#include <windows.h>   // for OutputDebugString()
```

```
int VDebugPrintF(const char* format, va_list argList)
{
    const U32 MAX_CHARS = 1024;
    static char s_buffer[MAX_CHARS];

    int charsWritten
     = vsnprintf(s_buffer, MAX_CHARS, format, argList);

    // Now that we have a formatted string, call the
    // Win32 API.
    OutputDebugString(s_buffer);

    return charsWritten;
}

int DebugPrintF(const char* format, ...)
{
    va_list argList;
    va_start(argList, format);

    int charsWritten = VDebugPrintF(format, argList);

    va_end(argList);
    return charsWritten;
}
```

请注意，这里实现了两个函数：DebugPrintF() 接受一个可变长度的参数列表（通过省略号 ... 指定），而 VDebug PrintF() 接受一个 va_list 参数。这样做是为了让程序员能够基于 VDebugPrintF() 构建额外的打印函数。（无法将省略号从一个函数传递到另一个函数，但可以传递 va_list。）

10.1.2 详细程度

一旦你费尽心思在代码中精心选择的位置添加了一堆打印语句，最好能把它们留在那里，以备日后再次需要。为了实现这一点，大多数引擎都提供了某种机制，可以通过命令行或在运行时动态控制详细程度。当详细程度处于最低值（通常为零）时，只会打印关键错误消息。当详细程度较高时，代码中嵌入的更多打印语句开始对输出做出贡献。


```
int VDebugPrintF(const char* format, va_list argList)
{
    const U32 MAX_CHARS = 1024;
    static char s_buffer[MAX_CHARS];

    int charsWritten
     = vsnprintf(s_buffer, MAX_CHARS, format, argList);

    // Now that we have a formatted string, call the
    // Win32 API.
    OutputDebugString(s_buffer);

    return charsWritten;
}

int DebugPrintF(const char* format, ...)
{
    va_list argList;
    va_start(argList, format);

    int charsWritten = VDebugPrintF(format, argList);

    va_end(argList);
    return charsWritten;
}
```

Notice that two functions are implemented: `DebugPrintF()` takes a variable-length argument list (specified via the ellipsis, ...), while `VDebugPrintF()` takes a `va_list` argument. This is done so that programmers can build additional printing functions in terms of `VDebugPrintF()`. (It's impossible to pass ellipses from one function to another, but it *is* possible to pass `va_lists` around.)

10.1.2 Verbosity

Once you've gone to the trouble of adding a bunch of print statements to your code in strategically chosen locations, it's nice to be able to leave them there, in case they're needed again later. To permit this, most engines provide some kind of mechanism for controlling the level of *verbosity* via the command line, or dynamically at runtime. When the verbosity level is at its minimum value (usually zero), only critical error messages are printed. When the verbosity is higher, more of the print statements embedded in the code start to contribute to the output.

实现此目的的最简单方法是将当前的详细级别存储在一个全局整数变量中，例如 g_verbosity 。然后，我们提供一个 VerboseDebugPrintF() 函数，其第一个参数是详细级别，超过该级别时将打印消息。该函数可以按如下方式实现：

```
int g_verbosity = 0;

void VerboseDebugPrintF(int verbosity,
                        const char* format, ...)
{
    // Only print when the global verbosity level is
    // high enough.
    if (g_verbosity >= verbosity)
    {
        va_list argList;
        va_start(argList, format);

        VDebugPrintF(format, argList);

        va_end(argList);
    }
}
```

10.1.3 通道

将调试输出分类到不同的通道也非常有用。例如，一个通道可能包含来自动画系统的消息，而另一个通道可能用于打印来自物理系统的消息。

在某些平台上，例如 PlayStation 3，调试输出可以定向到 14 个不同的 TTY 窗口之一。此外，消息会被镜像到一个特殊的 TTY 窗口，该窗口包含来自其他 14 个窗口的输出。这使得开发者可以轻松地专注于自己想要查看的消息。在处理动画问题时，只需切换到动画 TTY 并忽略所有其他输出即可。在处理来源不明的一般问题时，可以参考“所有”TTY 来寻找线索。

其他平台（如 Windows）仅提供单个调试输出控制台。然而，即使在这些系统上，将输出划分为多个通道也会很有帮助。每个通道的输出可能会被分配不同的颜色。您还可以实现过滤器，这些过滤器可以在运行时打开和关闭，并将输出限制为仅指定一个或一组通道。在这种模型中，例如，如果开发人员正在调试与动画相关的问题，他或她可以简单地过滤掉动画通道之外的所有通道。

The simplest way to implement this is to store the current verbosity level in a global integer variable, perhaps called `g_verbosity`. We then provide a `VerboseDebugPrintF()` function whose first argument is the verbosity level at or above which the message will be printed. This function could be implemented as follows:

```
int g_verbosity = 0;

void VerboseDebugPrintF(int verbosity,
                        const char* format, ...)
{
    // Only print when the global verbosity level is
    // high enough.
    if (g_verbosity >= verbosity)
    {
        va_list argList;
        va_start(argList, format);

        VDebugPrintF(format, argList);

        va_end(argList);
    }
}
```

10.1.3 Channels

It's also extremely useful to be able to categorize your debug output into *channels*. One channel might contain messages from the animation system, while another might be used to print messages from the physics system, for example.

On some platforms, like the PlayStation 3, debug output can be directed to one of 14 distinct TTY windows. In addition, messages are mirrored to a special TTY window that contains the output from all of the other 14 windows. This makes it very easy for a developer to focus in on only the messages he or she wants to see. When working on an animation problem, one can simply flip to the animation TTY and ignore all the other output. When working on a general problem of unknown origin, the "all" TTY can be consulted for clues.

Other platforms like Windows provide only a single debug output console. However, even on these systems it can be helpful to divide your output into channels. The output from each channel might be assigned a different color. You might also implement *filters*, which can be turned on and off at runtime, and restrict output to only a specified channel or set of channels. In this model, if a developer is debugging an animation-related problem, for example, he or she can simply filter out all of the channels except the animation channel.

通过在调试打印函数中添加一个额外的通道参数，可以非常轻松地实现基于通道的调试输出系统。通道可以进行编号，或者更好的方法是通过 C/C++ 枚举声明为其分配符号值。或者，可以使用字符串或散列字符串 ID 来命名通道。打印函数只需查询活动通道列表，并仅在指定通道位于其中时才打印消息。

如果您的通道数不超过 32 或 64 个，那么通过 32 位或 64 位掩码来识别通道会很有帮助。这使得实现通道过滤器就像指定一个整数一样简单。当掩码中的某个位为 1 时，相应的通道处于活动状态；当该位为 0 时，该通道处于静音状态。

10.1.3.1 使用 Redis 管理 TTY 通道

顽皮狗的开发者使用名为 Connector 的网页界面，作为他们查看游戏引擎在运行时发出的各种调试信息流的窗口。游戏会将其调试文本输出到各种命名通道，每个通道都与不同的引擎系统（动画、渲染、AI、音效等）相关联。这些数据流由轻量级的 Redis 键值存储系统收集（有关 Redis 的更多信息，请参阅 http://redis.io）。Connector 界面允许用户通过任何网页浏览器轻松查看和过滤这些 Redis 数据。

10.1.4 将输出镜像到文件

将所有调试输出镜像到一个或多个日志文件（例如，每个通道一个文件）是一个好主意。这样可以在问题发生后进行诊断。理想情况下，日志文件应该包含所有调试输出，与当前的详细程度和活动通道掩码无关。这样，只需检查最新的日志文件，就可以捕获和追踪意外问题。

您可能需要考虑在每次调用调试输出函数后刷新日志文件，以确保如果游戏崩溃，日志文件不会丢失最后一个充满缓冲区的输出。最后打印的数据通常对于确定崩溃原因最有用，因此我们希望确保日志文件始终包含最新的输出。当然，刷新输出缓冲区的开销可能很大。因此，您应该仅在以下情况下在每次调试输出调用后刷新缓冲区：(a) 您不需要进行大量日志记录，或者 (b) 您发现在您的特定平台上确实有必要刷新缓冲区。如果认为刷新是必要的，您可以随时提供引擎配置选项来打开和关闭它。

A channel-based debug output system can be implemented quite easily by adding an additional channel argument to our debug printing function. Channels might be numbered, or better, assigned symbolic values via a C/C++ `enum` declaration. Or channels might be named using a string or hashed string id. The printing function can simply consult the list of active channels and only print the message if the specified channel is among them.

If you don't have more than 32 or 64 channels, it can be helpful to identify the channels via a 32- or 64-bit mask. This makes implementing a channel filter as easy as specifying a single integer. When a bit in the mask is 1, the corresponding channel is active; when the bit is 0, the channel is muted.

10.1.3.1 Using Redis to Manage TTY Channels

The developers at Naughty Dog use a web-based interface called *Connector* as their window into the various streams of debugging information that are emitted by the game engine at runtime. The game spits out its debug text into various named channels, each associated with a different engine system (animation, rendering, AI, sound, etc.) These data streams are collected by a lightweight Redis key-value store (see http://redis.io for more information on Redis). The Connector interface allows users to view and filter this Redis data easily from any web browser.

10.1.4 Mirroring Output to a File

It's a good idea to mirror all debug output to one or more log files (e.g., one file per channel). This permits problems to be diagnosed after the fact. Ideally the log file(s) should contain *all* of the debug output, independent of the current verbosity level and active channels mask. This allows *unexpected* problems to be caught and tracked down by simply inspecting the most-recent log files.

You may want to consider *flushing* your log file(s) after every call to your debug output function to ensure that if the game crashes, the log file(s) won't be missing the last buffer-full of output. The last data printed are usually the most useful for determining the cause of a crash, so we want to be sure that the log file always contains the most up-to-date output. Of course, flushing the output buffer can be expensive. So you should only flush buffers after every debug output call if either (a) you are not doing a lot of logging, or (b) you discover that it is truly necessary on your particular platform. If flushing is deemed to be necessary, you can always provide an engine configuration option to turn it on and off.

10.1.5 崩溃报告

一些游戏引擎会在游戏崩溃时生成特殊的文本输出和/或日志文件。大多数操作系统都安装了一个顶级异常处理程序，可以捕获大多数崩溃。在这个函数中，您可以打印出各种有用的信息。您甚至可以考虑将崩溃报告通过电子邮件发送给整个编程团队。这对程序员来说可能非常有启发：当他们看到美术和设计团队崩溃的频率时，他们可能会在调试任务中发现新的紧迫感！

以下只是您可以在崩溃报告中包含的信息类型的几个示例：

- 崩溃时正在播放的当前级别。
- 发生碰撞时玩家角色的世界空间位置。
- 游戏崩溃时玩家的动画/动作状态。
- 崩溃时正在运行的游戏脚本。（如果崩溃是由脚本导致的，这一点尤其有用！）
- 堆栈跟踪。大多数操作系统都提供了遍历调用堆栈的机制（尽管这些机制并非标准且高度依赖于特定平台）。借助此功能，您可以打印出崩溃发生时堆栈上所有非内联函数的符号名称。
- 引擎中所有内存分配器的状态（可用内存量、碎片化程度等）。例如，当内存不足导致 bug 时，这类数据会很有帮助。
- 您认为在追查事故原因时可能相关的任何其他信息。
- 游戏崩溃时的屏幕截图。

10.2 调试绘图工具

现代交互式游戏几乎完全由数学驱动。我们使用数学来定位和定向游戏世界中的物体，移动它们，检测碰撞，投射光线以确定视线，当然，我们还使用矩阵乘法将物体从对象空间转换到世界空间，最终转换到屏幕空间进行渲染。几乎所有现代游戏都是三维的，但即使在二维游戏中，也很难在脑海中想象所有这些数学计算的结果。因此，大多数优秀的游戏引擎都提供了用于绘制彩色线条的 API。

10.1.5 Crash Reports

Some game engines produce special text output and/or log files when the game crashes. In most operating systems, a top-level exception handler can be installed that will catch most crashes. In this function, you could print out all sorts of useful information. You could even consider emailing the crash report to the entire programming team. This can be incredibly enlightening for the programmers: When they see just how often the art and design teams are crashing, they may discover a renewed sense of urgency in their debugging tasks!

Here are just a few examples of the kinds of information you can include in a crash report:

- Current level(s) being played at the time of the crash.
- World-space location of the player character when the crash occurred.
- Animation/action state of the player when the game crashed.
- Gameplay script(s) that were running at the time of the crash. (This can be especially helpful if the script is the cause of the crash!)
- Stack trace. Most operating systems provide a mechanism for walking the call stack (although they are nonstandard and highly platform specific). With such a facility, you can print out the symbolic names of all non-inline functions on the stack at the time the crash occurred.
- State of all memory allocators in the engine (amount of memory free, degree of fragmentation, etc.). This kind of data can be helpful when bugs are caused by low-memory conditions, for example.
- Any other information you think might be relevant when tracking down the cause of a crash.
- A screenshot of the game at the moment it crashed.

10.2 Debug Drawing Facilities

Modern interactive games are driven almost entirely by math. We use math to position and orient objects in the game world, move them around, test for collisions, and cast rays to determine lines of sight, and of course we use matrix multiplication to transform objects from object space to world space and eventually into screen space for rendering. Almost all modern games are three-dimensional, but even in a two-dimensional game it can be very difficult to mentally visualize the results of all these mathematical calculations. For this reason, most good game engines provide an API for drawing colored lines,

简单的形状和 3D 文本。我们称之为调试绘图工具，因为用它绘制的线条、形状和文本旨在在开发和调试期间实现可视化，并在游戏发布前被删除。

调试绘图 API 可以为您节省大量时间。例如，如果您想知道为什么您的射弹没有击中敌方角色，哪个更容易？是在调试器中解读一堆数字？还是在游戏中绘制一条显示射弹轨迹的三维线？有了调试绘图 API，逻辑和数学错误就会立即显现出来。可以说，一张图片胜过一千分钟的调试时间。

以下是顽皮狗引擎中一些调试绘图的示例。以下截图均来自我们的游戏测试关卡，这是我们用来测试新功能和调试游戏中问题的众多特殊关卡之一。

图 10.1 展示了敌方 NPC 对玩家感知的可视化效果。小“火柴人”代表 NPC 感知到的玩家位置。当玩家脱离与 NPC 之间的视线时，“火柴人”仍会停留在玩家最后已知的位置，即使玩家潜行离开。

- 图 10.2 显示了如何使用线框球体来可视化爆炸的动态扩展爆炸半径。

图 10.1.《最后生还者：重制版》中 NPC 到玩家的视线可视化（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

simple shapes and 3D text. We call this a *debug drawing* facility, because the lines, shapes and text that are drawn with it are intended for visualization during development and debugging and are removed prior to shipping the game.

A *debug drawing* API can save you huge amounts of time. For example, if you are trying to figure out why your projectiles are not hitting the enemy characters, which is easier? Deciphering a bunch of numbers in the debugger? Or drawing a line showing the trajectory of the projectile in three dimensions within your game? With a debug drawing API, logical and mathematical errors become immediately obvious. One might say that a picture is worth a thousand minutes of debugging.

Here are some examples of debug drawing in action within Naughty Dog's engine. The following screenshots were all taken within our *play-test* level, one of many special levels we use for testing out new features and debugging problems in the game.

- Figure 10.1 shows a visualization of an enemy NPC's perception of the player. The little "stick man" figure represents the location of the player as perceived by the NPC. When the player has broken the line of sight between himself and the NPC, the "stick man" will remain at the player's last known location, even after the player sneaks away.
- Figure 10.2 shows how a wireframe sphere can be used to visualize the dynamically expanding blast radius of an explosion.

Figure 10.1. Visualizing the line of sight from an NPC to the player in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

图 10.2. 在顽皮狗引擎中可视化爆炸的扩大爆炸球。

图 10.3 展示了如何使用圆圈来直观地表示 Drake 在游戏中寻找可悬挂的壁架时所使用的半径。一条线表示他当前悬挂的壁架。

图 10.4 展示了一个处于特殊调试模式的 AI 角色。在此模式下，角色的大脑实际上处于关闭状态，开发者可以通过一个简单的抬头菜单完全控制角色的移动和动作。开发者可以绘制

图 10.3.《神秘海域》系列中 Drake 的壁架悬挂和摆动系统中使用的球体和矢量（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 3）。

Figure 10.2. Visualizing the expanding blast sphere of an explosion in the Naughty Dog engine.

- Figure 10.3 shows how circles can be used to visualize the radii used by Drake when searching for ledges to hang from in the game. A line shows the ledge he is currently hanging from.
- Figure 10.4 shows an AI character that has been placed in a special debugging mode. In this mode, the character's brain is effectively turned off, and the developer is given full control over the character's movements and actions via a simple heads-up menu. The developer can paint

Figure 10.3. Spheres and vectors used in Drake's ledge hang and shimmy system in the *Uncharted* series (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 3).

图 10.4. 在《最后生还者：重制版》中手动控制 NPC 的动作以进行调试（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

只需瞄准摄像头，即可在游戏世界中锁定目标点，然后指示角色步行、奔跑或冲刺到指定点。用户还可以指示角色进入或离开附近的掩体、开火等等。

10.2.1 调试绘图 API

一个调试绘图API一般需要满足以下要求：

- API 应该简单且易于使用。
- 它应该支持一组有用的原语，包括（但不限于）：
  - 线条，
  - 球体，
  - 点（通常表示为小十字或球体，因为单个像素很难看到），
  - 坐标轴（通常，x 轴用红色绘制，y 轴用绿色绘制，z 轴用蓝色绘制），
  - 边界框，以及
  - 格式化文本。
- 它应该在控制图元绘制方式方面提供很大的灵活性，包括：

Figure 10.4. Manually controlling an NPC's actions for debugging purposes in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

target points in the game world by simply aiming the camera and can then instruct the character to walk, run or sprint to the specified points. The user can also tell the character to enter or leave nearby cover, fire its weapon and so on.

10.2.1 Debug Drawing API

A debug drawing API generally needs to satisfy the following requirements:

- The API should be simple and easy to use.
- It should support a useful set of primitives, including (but not limited to):
  - lines,
  - spheres,
  - points (usually represented as small crosses or spheres, because a single pixel is very difficult to see),
  - coordinate axes (typically, the x-axis is drawn in red, y in green and z in blue),
  - bounding boxes, and
  - formatted text.
- It should provide a good deal of flexibility in controlling how primitives are drawn, including:

- 颜色，
- 线宽，
- 球体半径，
- 点的大小、坐标轴的长度以及其他“固定”图元的尺寸。

• 应该能够在世界空间（全3D，使用游戏摄像机的透视投影矩阵）或屏幕空间（使用正交投影或透视投影）中绘制图元。世界空间图元对于注释3D场景中的对象非常有用。屏幕空间图元有助于以独立于摄像机位置或方向的平视显示形式显示调试信息。

• 无论是否启用深度测试，都应该能够绘制图元。

- 启用深度测试后，图元将被场景中的真实物体遮挡。这使得它们的深度易于可视化，但这也意味着图元有时可能难以被看到，甚至完全被场景的几何形状遮挡。
- 禁用深度测试后，图元将“悬停”在场景中的真实物体上方。这使得测量其真实深度变得更加困难，但也确保了没有任何图元被隐藏在视野之外。

• 应该能够在代码的任何位置调用绘图 API。大多数渲染引擎要求在游戏循环的特定阶段（通常是在每一帧的末尾）提交几何图形进行渲染。此要求意味着系统必须将所有传入的调试绘图请求排队，以便它们可以在稍后的适当时间提交。

• 理想情况下，每个调试原语都应该具有与其相关的生命周期。
生命周期控制着图元在被请求后在屏幕上停留的时间。如果绘制图元的代码每帧都会被调用，则生命周期可以是一帧——图元将停留在屏幕上，因为它每帧都会被刷新。但是，如果绘制图元的代码很少被调用或只是间歇性地调用（例如，计算抛射物初速度的函数），那么您肯定不希望图元在屏幕上闪烁一帧就消失。在这种情况下，程序员应该能够赋予其调试图元更长的生命周期，大约几秒。

- color,
- line width,
- sphere radii,
- the size of points, lengths of coordinate axes, and dimensions of other “canned” primitives.

- It should be possible to draw primitives in world space (full 3D, using the game camera’s perspective projection matrix) or in screen space (either using an orthographic projection, or possibly a perspective projection). World-space primitives are useful for annotating objects in the 3D scene. Screen-space primitives are helpful for displaying debugging information in the form of a heads-up display that is independent of camera position or orientation.
- It should be possible to draw primitives with or without *depth testing* enabled.
  - When depth testing is enabled, the primitives will be occluded by real objects in your scene. This makes their depth easy to visualize, but it also means that the primitives may sometimes be difficult to see or totally hidden by the geometry of your scene.
  - With depth testing disabled, the primitives will “hover” over the real objects in the scene. This makes it harder to gauge their real depth, but it also ensures that no primitive is ever hidden from view.
- It should be possible to make calls to the drawing API from anywhere in your code. Most rendering engines require that geometry be submitted for rendering during a specific phase of the game loop, usually at the end of each frame. This requirement implies that the system must queue up all incoming debug drawing requests, so that they may be submitted at the proper time later on.
- Ideally, every debug primitive should have a *lifetime* associated with it. The lifetime controls how long the primitive will remain on-screen after having been requested. If the code that is drawing the primitive is called every frame, the lifetime can be one frame—the primitive will remain on-screen because it will be refreshed every frame. However, if the code that draws the primitive is called rarely or intermittently (e.g., a function that calculates the initial velocity of a projectile), then you do not want the primitive to flicker on-screen for just one frame and then disappear. In such situations, the programmer should be able to give his or her debug primitives a longer lifetime, on the order of a few seconds.

• 调试绘制系统能够高效处理大量调试原语也至关重要。当您为 1,000 个游戏对象绘制调试信息时，原语的数量会非常庞大□□，您肯定不希望在启用调试绘制功能后游戏无法使用。

Naughty Dog 引擎中的调试绘图 API 类似于这：

```
class DebugDrawManager
{
public:
    // Adds a line segment to the debug drawing queue.
    void AddLine(const Point& fromPosition,
                 const Point& toPosition,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

    // Adds an axis-aligned cross (3 lines converging at
    // a point) to the debug drawing queue.
    void AddCross(const Point& position,
                  Color color,
                  float size,
                  float duration = 0.0f,
                  bool depthEnabled = true);

    // Adds a wireframe sphere to the debug drawing queue.
    void AddSphere(const Point& centerPosition,
                   float radius,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);

    // Adds a circle to the debug drawing queue.
    void AddCircle(const Point& centerPosition,
                   const Vector& planeNormal,
                   float radius,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);

    // Adds a set of coordinate axes depicting the
    // position and orientation of the given
    // transformation to the debug drawing queue.
    void AddAxes(const Transform& xfm,
                 Color color,
                 float size,
```


- It's also important that the debug drawing system be capable of handling a large number of debug primitives efficiently. When you're drawing debug information for 1,000 game objects, the number of primitives can really add up, and you don't want your game to be unusable when debug drawing is turned on.

The debug drawing API in Naughty Dog's engine looks something like this:

```
class DebugDrawManager
{
public:
    // Adds a line segment to the debug drawing queue.
    void AddLine(const Point& fromPosition,
                 const Point& toPosition,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

    // Adds an axis-aligned cross (3 lines converging at
    // a point) to the debug drawing queue.
    void AddCross(const Point& position,
                  Color color,
                  float size,
                  float duration = 0.0f,
                  bool depthEnabled = true);

    // Adds a wireframe sphere to the debug drawing queue.
    void AddSphere(const Point& centerPosition,
                   float radius,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);

    // Adds a circle to the debug drawing queue.
    void AddCircle(const Point& centerPosition,
                   const Vector& planeNormal,
                   float radius,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);

    // Adds a set of coordinate axes depicting the
    // position and orientation of the given
    // transformation to the debug drawing queue.
    void AddAxes(const Transform& xfm,
                 Color color,
                 float size,
```

```
                float duration = 0.0f,
                bool depthEnabled = true);

    // Adds a wireframe triangle to the debug drawing
    // queue.
    void AddTriangle(const Point& vertex0,
                     const Point& vertex1,
                     const Point& vertex2,
                     Color color,
                     float lineWidth = 1.0f,
                     float duration = 0.0f,
                     bool depthEnabled = true);

    // Adds an axis-aligned bounding box to the debug
    // queue.
    void AddAABB(const Point& minCoords,
                 const Point& maxCoords,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

    // Adds an oriented bounding box to the debug queue.
    void AddOBB(const Mat44& centerTransform,
                const Vector& scaleXYZ,
                Color color,
                float lineWidth = 1.0f,
                float duration = 0.0f,
                bool depthEnabled = true);

    // Adds a text string to the debug drawing queue.
    void AddString(const Point& pos,
                   const char* text,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);
};

// This global debug drawing manager is configured for
// drawing in full 3D with a perspective projection.
extern DebugDrawManager g_debugDrawMgr;


// This global debug drawing manager draws its
// primitives in 2D screen space. The (x,y) coordinates
// of a point specify a 2D location on-screen, and the
// z coordinate contains a special code that indicates
// whether the (x,y) coordidates are measured in absolute
```



```
                float duration = 0.0f,
                bool depthEnabled = true);

    // Adds a wireframe triangle to the debug drawing
    // queue.
    void AddTriangle(const Point& vertex0,
                     const Point& vertex1,
                     const Point& vertex2,
                     Color color,
                     float lineWidth = 1.0f,
                     float duration = 0.0f,
                     bool depthEnabled = true);

    // Adds an axis-aligned bounding box to the debug
    // queue.
    void AddAABB(const Point& minCoords,
                 const Point& maxCoords,
                 Color color,
                 float lineWidth = 1.0f,
                 float duration = 0.0f,
                 bool depthEnabled = true);

    // Adds an oriented bounding box to the debug queue.
    void AddOBB(const Mat44& centerTransform,
                const Vector& scaleXYZ,
                Color color,
                float lineWidth = 1.0f,
                float duration = 0.0f,
                bool depthEnabled = true);

    // Adds a text string to the debug drawing queue.
    void AddString(const Point& pos,
                   const char* text,
                   Color color,
                   float duration = 0.0f,
                   bool depthEnabled = true);
};

// This global debug drawing manager is configured for
// drawing in full 3D with a perspective projection.
extern DebugDrawManager g_debugDrawMgr;


// This global debug drawing manager draws its
// primitives in 2D screen space. The (x,y) coordinates
// of a point specify a 2D location on-screen, and the
// z coordinate contains a special code that indicates
// whether the (x,y) coordidates are measured in absolute
```

```
// pixels or in normalized coordinates that range from
// 0.0 to 1.0. (The latter mode allows drawing to be
// independent of the actual resolution of  the screen.)
extern DebugDrawManager g_debugDrawMgr2D;
```

以下是游戏代码中使用此 API 的示例：

```
void Vehicle::Update()
{
    // Do some calculations...

    // Debug-draw my velocity vector.
    const Point& start = GetWorldSpacePosition();
    Point end = start + GetVelocity();
    g_debugDrawMgr.AddLine(start, end, kColorRed);

    // Do some other calculations...

    // Debug-draw my name and number of passengers.
    {
        char buffer[128];
        sprintf(buffer, "Vehicle %s: %d passengers",
                GetName(), GetNumPassengers());

        const Point& pos = GetWorldSpacePosition();
        g_debugDrawMgr.AddString(pos,
            buffer, kColorWhite, 0.0f, false);
    }
}
```

你会注意到，绘图函数的名称使用了动词“add”而不是“draw”。这是因为调试原语通常不会在调用绘图函数时立即绘制。相反，它们会被添加到一个可视元素列表中，这些元素稍后会被绘制。大多数高速 3D 渲染引擎要求将所有可视元素维护在一个场景数据结构中，以便高效地绘制它们，通常是在游戏循环结束时。我们将在第 11 章中学习更多关于渲染引擎工作原理的知识。

10.3 游戏内菜单

每个游戏引擎都有大量的配置选项和功能。事实上，每个主要子系统，包括渲染、动画、碰撞、物理、音频、网络、玩家机制、AI 等等，都公开了各自专门的配置选项。这对程序员、艺术家来说非常有用。


```
// pixels or in normalized coordinates that range from
// 0.0 to 1.0. (The latter mode allows drawing to be
// independent of the actual resolution of  the screen.)
extern DebugDrawManager g_debugDrawMgr2D;
```

Here’s an example of this API being used within game code:

```
void Vehicle::Update()
{
    // Do some calculations...

    // Debug-draw my velocity vector.
    const Point& start = GetWorldSpacePosition();
    Point end = start + GetVelocity();
    g_debugDrawMgr.AddLine(start, end, kColorRed);

    // Do some other calculations...

    // Debug-draw my name and number of passengers.
    {
        char buffer[128];
        sprintf(buffer, "Vehicle %s: %d passengers",
                GetName(), GetNumPassengers());

        const Point& pos = GetWorldSpacePosition();
        g_debugDrawMgr.AddString(pos,
            buffer, kColorWhite, 0.0f, false);
    }
}
```

You’ll notice that the names of the drawing functions use the verb “add” rather than “draw.” This is because the debug primitives are typically not drawn immediately when the drawing function is called. Instead, they are added to a list of visual elements that will be drawn at a later time. Most high-speed 3D rendering engines require that all visual elements be maintained in a *scene* data structure so that they can be drawn efficiently, usually at the end of the game loop. We’ll learn a lot more about how rendering engines work in Chapter 11.

10.3 In-Game Menus

Every game engine has a large number of configuration options and features. In fact, each major subsystem, including rendering, animation, collision, physics, audio, networking, player mechanics, AI and so on, exposes its own specialized configuration options. It is highly useful to programmers, artists

游戏开发者和游戏设计师都能够在游戏运行时配置这些选项，而无需修改源代码、重新编译和链接游戏可执行文件，然后重新运行游戏。这可以大大减少游戏开发团队在调试问题和设置新关卡或游戏机制上花费的时间。

实现此类功能的一个简单便捷的方法是提供游戏内菜单系统。游戏内菜单上的菜单项可以执行多种操作，包括（但不限于）：

- 切换全局布尔设置，
- 调整全局整数和浮点值，
- 调用任意函数，这些函数可以在引擎内执行任何任务，并且
- 调出子菜单，允许菜单系统按层次组织，以便于导航。

游戏内菜单应该易于调出，例如只需按下手柄上的按钮即可。（当然，你需要选择在正常游戏过程中不会出现的按键组合。）调出菜单通常会暂停游戏。这样，开发者就可以一直玩游戏到问题发生前，然后通过调出菜单暂停游戏，调整引擎设置，以便直观地看到问题。

图 10.5.《最后生还者：重制版》的主开发菜单（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

and game designers alike to be able to configure these options while the game is running, without having to change the source code, recompile and relink the game executable, and then rerun the game. This can greatly reduce the amount of time the game development team spends on debugging problems and setting up new levels or game mechanics.

One simple and convenient way to permit this kind of thing is to provide a system of *in-game menus*. Items on an in-game menu can do any number of things, including (but certainly not limited to):

- toggling global Boolean settings,
- adjusting global integer and floating-point values,
- calling arbitrary functions, which can perform literally any task within the engine, and
- bringing up submenus, allowing the menu system to be organized hierarchically for easy navigation.

An in-game menu should be easy and convenient to bring up, perhaps via a simple button press on the joypad. (Of course, you'll want to choose a button combination that doesn't occur during normal gameplay.) Bringing up the menus usually pauses the game. This allows the developer to play the game until the moment just before a problem occurs, then pause the game by bringing up the menus, adjust engine settings in order to visualize the

Figure 10.5. Main development menu in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

图 10.6.《最后生还者：重制版》中的渲染子菜单（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

更清楚地了解问题，然后取消暂停游戏以深入检查问题。

让我们简单了解一下顽皮狗引擎的菜单系统是如何运作的。图 10.5 展示了顶级菜单。它包含引擎中每个主要子系统的子菜单。在图 10.6 中，我们向下钻取了一层菜单。

图 10.7.《最后生还者：重制版》中的网格选项子菜单（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

Figure 10.6. Rendering submenu in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

problem more clearly, and then un-pause the game to inspect the problem in depth.

Let's take a brief look at how the menu system works in the Naughty Dog engine. Figure 10.5 shows the top-level menu. It contains submenus for each major subsystem in the engine. In Figure 10.6, we've drilled down one level

Figure 10.7. Mesh options subsubmenu in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

图 10.8。背景网格已关闭（《最后生还者：重制版》© 2014/TM SIE。由顽皮狗创建和开发，Play Station 4）。

进入“渲染...”子菜单。由于渲染引擎是一个高度复杂的系统，其菜单包含许多子菜单，用于控制渲染的各个方面。为了控制 3D 网格的渲染方式，我们进一步深入到“网格选项...”子菜单，如图 10.7 所示。在此菜单中，我们可以关闭所有静态背景网格的渲染，只保留动态前景网格可见。如图 10.8 所示。（啊哈，那只讨厌的鹿！）

10.4 游戏内控制台

一些引擎会提供游戏内控制台，用于替代或补充游戏内菜单系统。游戏内控制台为游戏引擎的功能提供了命令行界面，类似于 DOS 命令提示符让用户访问 Windows 操作系统的各种功能，或 csh、tcsh、ksh 或 bash shell 提示符让用户访问类 UNIX 操作系统的功能。与菜单系统类似，游戏引擎控制台可以提供命令，允许开发者查看和操作全局引擎设置，以及运行任意命令。

控制台比菜单系统略逊一筹，尤其对于那些打字速度不快的人来说。然而，控制台的功能远比菜单强大。有些游戏内置控制台只提供一组基本的硬编码命令，这使得它们的灵活性与菜单系统差不多。

Figure 10.8. Background meshes turned off (*The Last of Us: Remastered* © 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

into the *Rendering…* submenu. Since the rendering engine is a highly complex system, its menu contains many submenus controlling various aspects of rendering. To control the way in which 3D meshes are rendered, we drill down further into the *Mesh Options…* submenu, shown in Figure 10.7. On this menu, we can turn off rendering of all static background meshes, leaving only the dynamic foreground meshes visible. This is shown in Figure 10.8. (Ah ha, there's that pesky deer!)

10.4 In-Game Console

Some engines provide an in-game console, either in lieu of or in addition to an in-game menu system. An in-game console provides a command-line interface to the game engine's features, much as a DOS *command prompt* provides users with access to various features of the Windows operating system, or a csh, tcsh, ksh or bash *shell prompt* provides users with access to the features of UNIX-like operating systems. Much like a menu system, the game engine console can provide commands allowing a developer to view and manipulate global engine settings, as well as running arbitrary commands.

A console is somewhat less convenient than a menu system, especially for those who aren't very fast typists. However, a console can be much more powerful than a menu. Some in-game consoles provide only a rudimentary set of hard-coded commands, making them about as flexible as a menu system.

图 10.9. Minecraft 中的游戏内控制台，覆盖在主游戏屏幕的顶部并显示有效命令列表。

但其他引擎几乎为引擎的所有功能提供了丰富的界面。Minecraft 游戏内控制台的屏幕截图如图 10.9 所示。

一些游戏引擎提供了强大的脚本语言，程序员和游戏设计师可以使用它来扩展引擎的功能，甚至构建全新的游戏。如果游戏内的控制台也支持这种脚本语言，那么任何可以用脚本实现的功能，都可以通过控制台以交互方式完成。我们将在第 16.9 节深入探讨脚本语言。

10.5 调试摄像机和暂停游戏

游戏内菜单或控制台系统最好搭配另外两个关键功能：(a) 能够将摄像机从玩家角色身上分离，并在游戏世界中飞行，以便仔细观察场景的各个方面；(b) 能够暂停、取消暂停和单步运行游戏（参见第 8.5.5 节）。即使游戏暂停，能够控制摄像机仍然很重要。为了支持这一点，我们可以简单地保持渲染引擎和摄像机控制运行，即使游戏的逻辑时钟暂停。

慢动作模式是另一个非常有用的功能，可以用来仔细观察动画、粒子效果、物理和碰撞行为、AI行为等等。这个功能很容易实现。假设我们已使用一个不同于实时时钟的时钟来更新所有游戏元素，那么我们只需以比平时更慢的速度更新游戏时钟，就能让游戏进入慢动作。这种方法也可以用来实现快动作模式，这在快速切换耗时的游戏环节以找到感兴趣的区域时非常有用（更不用说它还能带来很多笑料，尤其是在配上糟糕的Benny Hill音乐演唱时……）。

Figure 10.9. The in-game console in *Minecraft*, overlaid on top of the main game screen and displaying a list of valid commands.

But others provide a rich interface to virtually every feature of the engine. A screenshot of the in-game console in Minecraft is shown in Figure 10.9.

Some game engines provide a powerful scripting language that can be used by programmers and game designers to extend the functionality of the engine, or even build entirely new games. If the in-game console "speaks" this same scripting language, then anything you can do in script can also be done interactively via the console. We'll explore scripting languages in depth in Section 16.9.

10.5 Debug Cameras and Pausing the Game

An in-game menu or console system is best accompanied by two other crucial features: (a) the ability to detach the camera from the player character and fly it around the game world in order to scrutinize any aspect of the scene, and (b) the ability to pause, un-pause and single-step the game (see Section 8.5.5). When the game is paused, it is still important to be able to control the camera. To support this, we can simply keep the rendering engine and camera controls running, even when the game's logical clock is paused.

Slow motion mode is another incredibly useful feature for scrutinizing animations, particle effects, physics and collision behaviors, AI behaviors, and the list goes on. This feature is easy to implement. Presuming we've taken care to update all gameplay elements using a clock that is distinct from the real-time clock, we can put the game into slo-mo by simply updating the gameplay clock at a rate that is slower than usual. This approach can also be used to implement a fast-motion mode, which can be useful for moving rapidly through time-consuming portions of gameplay in order to get to an area of interest (not to mention being a great source of laughs, especially when accompanied by a bad vocal rendition of Benny Hill music…).

10.6 作弊

在开发或调试游戏时，允许用户以权宜之计打破游戏规则非常重要。这类功能恰如其分地被称为“作弊”。例如，许多引擎允许您“拾起”玩家角色，并让其在游戏世界中飞行，同时禁用碰撞，以便其能够穿过所有障碍物。这对于测试游戏玩法非常有帮助。您无需花时间实际玩游戏，试图让玩家角色到达某个理想的位置，只需简单地将他拾起，飞到您希望他到达的位置，然后将其放回其常规游戏模式即可。

其他有用的作弊手段包括但不限于：

- 无敌的玩家。作为一名开发人员，在测试功能或查找错误时，您通常不想被敌人角色的攻击所困扰，也不想担心从太高的地方掉下来。
- 为玩家提供武器。为了测试目的，在游戏中为玩家提供任何武器通常很有用。
- 无限弹药。当您试图杀死坏人来测试武器系统或AI命中反应时，您肯定不想到处寻找弹夹！
- 选择玩家网格。如果玩家角色有多个“服装”，那么选择其中任何一个进行测试都会很有用。

显然，这份清单可以写上好几页。没有限制——你可以添加任何你需要的作弊代码来开发或调试游戏。你甚至可以把一些你最喜欢的作弊代码分享给最终发售游戏的玩家。玩家通常可以通过在游戏手柄或键盘上输入未公开的作弊码，或者完成游戏中的某些目标来激活作弊代码。

10.7 屏幕截图和影片捕捉

另一个非常有用的功能是能够截取屏幕截图，并以合适的图像格式（例如 Windows 位图文件 (.bmp)、JPEG (.jpg) 或 Targa (.tga)）写入磁盘。截取屏幕截图的具体操作因平台而异，但通常需要调用图形 API，以便将帧缓冲区的内容从视频 RAM 传输到主 RAM，然后在主 RAM 中进行扫描并将其转换为您选择的图像文件格式。图像文件通常被写入预定义的

10.6 Cheats

When developing or debugging a game, it's important to allow the user to break the rules of the game in the name of expediency. Such features are aptly named *cheats*. For example, many engines allow you to "pick up" the player character and fly him or her around in the game world, with collisions disabled so he or she can pass through all obstacles. This can be incredibly helpful for testing out gameplay. Rather than taking the time to actually play the game in an attempt to get the player character into some desirable location, you can simply pick him up, fly him over to where you want him to be, and then drop him back into his regular gameplay mode.

Other useful cheats include, but are certainly not limited to:

- *Invincible player*. As a developer, you often don't want to be bothered having to defend yourself from enemy characters, or worrying about falling from too high a height, as you test out a feature or track down a bug.
- *Give player weapon*. It's often useful to be able to give the player any weapon in the game for testing purposes.
- *Infinite ammo*. When you're trying to kill bad guys to test out the weapon system or AI hit reactions, you don't want to be scrounging for clips!
- *Select player mesh*. If the player character has more than one "costume," it can be useful to be able to select any of them for testing purposes.

Obviously this list could go on for pages. The sky's the limit—you can add whatever cheats you need in order to develop or debug the game. You might even want to expose some of your favorite cheats to the players of the final shipping game. Players can usually activate cheats by entering unpublished *cheat codes* on the joypad or keyboard and/or by accomplishing certain objectives in the game.

10.7 Screenshots and Movie Capture

Another extremely useful facility is the ability to capture screenshots and write them to disk in a suitable image format such as Windows Bitmap files (.bmp), JPEG (.jpg) or Targa (.tga). The details of how to capture a screenshot vary from platform to platform, but they typically involve making a call to the graphics API that allows the contents of the frame buffer to be transferred from video RAM to main RAM, where it can be scanned and converted into the image file format of your choice. The image files are typically written to a predefined

文件夹位于磁盘上并使用日期和时间戳命名以保证文件名的唯一性。

您可能希望为用户提供各种选项来控制屏幕截图的截取方式。一些常见示例包括：

- 是否在屏幕截图中包含调试原语和文本。
- 是否在屏幕截图中包含平视显示器 (HUD) 元素。
- 截取的分辨率。某些引擎允许截取高分辨率的屏幕截图，可能是通过修改投影矩阵来实现的，这样就可以在正常分辨率下分别截取屏幕四个象限的屏幕截图，然后将其组合成最终的高分辨率图像。
- 简单的相机动画。例如，您可以允许用户标记相机的起始和终止位置及方向。然后，您可以拍摄一系列屏幕截图，同时逐步将相机从起始位置插入到终止位置。

一些引擎还提供功能齐全的电影捕捉模式。该系统会以游戏的目标帧率捕捉一系列屏幕截图，然后离线或运行时处理这些屏幕截图，生成合适格式（例如 MPEG-2 (H.262) 或 MPEG-4 Part 10 (H.264)）的电影文件。即使您的引擎不支持实时视频捕捉，也可以使用 Roxio Game Capture HD Pro 等外部硬件来捕捉游戏主机或 PC 的输出。对于 PC 和 Mac 游戏，有很多软件视频捕捉工具可供选择，包括 Beepa 的 Fraps、Camtasia Software 的 Camtasia、ExKode 的 Dxtory、NCH Software 的 Debut 和 Mirillis 的 Action!。

PlayStation 4 内置了游戏截图和视频片段的分享功能。在游戏过程中，PS4 会持续捕捉用户最近 15 分钟的游戏体验视频。用户可以随时点击控制器上的“分享”按钮，选择以各种方式分享截图或录制的视频——例如保存到 PS4 的硬盘或 U 盘，或者上传到各种在线服务。在顽皮狗，我们会使用这些功能在游戏崩溃时捕捉视频和截图，从而了解导致崩溃的具体情况。

PlayStation 4 用户还可以直播游戏过程。出于开发目的，可以使用 PC 连接到远程 PS4（可能放在工作室内或工作室外其他开发者的桌子上），以便

folder on disk and named using a date and time stamp to guarantee unique file names.

You may want to provide your users with various options controlling how screenshots are to be captured. Some common examples include:

- Whether or not to include debug primitives and text in the screenshot.
- Whether or not to include heads-up display (HUD) elements in the screenshot.
- The resolution at which to capture. Some engines allow high-resolution screenshots to be captured, perhaps by modifying the projection matrix so that separate screenshots can be taken of the four quadrants of the screen at normal resolution and then combined into the final high-res image.
- Simple camera animations. For example, you could allow the user to mark the starting and ending positions and orientations of the camera. A sequence of screenshots could then be taken while gradually interpolating the camera from the starting location to the ending location.

Some engines also provide a full-fledged movie capture mode. Such a system captures a sequence of screenshots at the target frame rate of the game, which are processed either offline or at runtime to generate a movie file in a suitable format such as MPEG-2 (H.262) or MPEG-4 Part 10 (H.264). But even if your engine doesn't support real-time video capture, external hardware like *Roxio Game Capture HD Pro* can always be used to capture the output from your game console or PC. And for PC and Mac games, a great many software video capture tools are available, including *Fraps* by Beepa, *Camtasia* by Camtasia Software, *Dxtory* by ExKode, *Debut* by NCH Software and *Action!* by Mirillis.

The PlayStation 4 has built-in support for sharing screenshots and video clips taken from within the game. During gameplay, the PS4 is continually capturing video of the most-recent 15 minutes of the user's in-game experience. At any moment, the user can hit the Share button on the controller and opt to share a screenshot or the recorded video in various ways—by saving it to the PS4's HDD or a thumb drive, or by uploading it to one of a number of online services. At Naughty Dog, we use these facilities to capture video and a screenshot of the game any time it crashes, thereby allowing us to see what situation led up to the crash.

PlayStation 4 users can also live-stream video of their playthroughs. For development purposes, it's possible to use a PC to connect to a remote PS4 (perhaps sitting on another developer's desk within or outside your studio), to

图 10.10。顽皮狗引擎提供了配置文件层次结构显示，允许用户深入了解特定的函数调用以检查其成本。

通过流媒体视频观看游戏进程，甚至可以通过将 PS4 控制器插入 PC 的 USB 插槽来控制远程游戏。这项功能对于那些“但它在我的机器上可以正常使用”的情况非常有用，因为您可以直接在对方的 PS4 上调试问题。

10.8 游戏内分析

游戏是实时系统，因此实现和保持高帧速率（通常为 30 FPS 或 60 FPS）非常重要。因此，任何游戏程序员的工作之一就是确保他或她的代码高效运行且不超出预算。正如我们在第 2 章讨论 80/20 规则时所看到的，很大一部分代码可能不需要优化。了解哪些部分需要优化的唯一方法是测量游戏的性能。我们在第 2 章中讨论了各种第三方分析工具。但是，这些工具有各种限制，并且可能在控制台上根本无法使用。出于这个原因，和/或为了方便，许多游戏引擎都提供了某种游戏内分析工具。

通常，游戏内置的分析器允许程序员注释需要计时的代码块，并为其赋予易于理解的名称。分析器通过 CPU 的高分辨率计时器测量每个注释块的执行时间，并将结果存储在内存中。它提供了一个抬头显示器，显示每个代码块的最新执行时间（示例如图 10.10 和 10.11 所示）。显示器通常以各种形式提供数据，

Figure 10.10. The Naughty Dog engine provides a profile hierarchy display that allows the user to drill down into particular function calls to inspect their costs.

see the game being played via streaming video, and even to control the remote game by plugging a PS4 controller into a USB slot on the PC. This facility can be incredibly useful for those “but it works on my machine” moments, because you can debug the problem directly on the other person’s PS4.

10.8 In-Game Profiling

Games are real-time systems, so achieving and maintaining a high frame rate (usually 30 FPS or 60 FPS) is important. Therefore, part of any game programmer’s job is ensuring that his or her code runs efficiently and within budget. As we saw when we discussed the 80/20 rule in Chapter 2, a large percentage of your code probably doesn’t need to be optimized. The only way to know *which* bits require optimization is to *measure your game’s performance*. We discussed various third-party profiling tools in Chapter 2. However, these tools have various limitations and may not be available at all on a console. For this reason, and/or for convenience, many game engines provide an in-game profiling tool of some sort.

Typically an in-game profiler permits the programmer to annotate blocks of code that should be timed and give them human-readable names. The profiler measures the execution time of each annotated block via the CPU’s hi-res timer and stores the results in memory. A heads-up display is provided, which shows up-to-date execution times for each code block (examples are shown in Figures 10.10 and 10.11). The display often provides the data in various forms,

包括原始周期数、以微秒为单位的执行时间以及相对于整个帧的执行时间的百分比。

图 10.11。《神秘海域：失落的遗产》（© 2017/TM SIE。由顽皮狗创建和开发，PlayStation 4）中的时间线模式准确显示了 PS4 的七个 CPU 核心在单个帧上执行各种操作的时间。

10.8.1 层次分析

用命令式语言编写的计算机程序本质上是分层的——一个函数调用其他函数，而这些函数又会调用更多的函数。例如，假设函数 a() 调用函数 b() 和 c()，而函数 b() 又调用函数 d()、e() 和 f()。伪代码如下所示。

```
void a()
{
    b();
    c();
}

void b()
{
    d();
    e();
    f();
}

void c() { ... }
```


including raw numbers of cycles, execution times in microseconds, and percentages relative to the execution time of the entire frame.

Figure 10.11. The timeline mode in *Uncharted: The Lost Legacy* (© 2017/™ SIE. Created and developed by Naughty Dog, PlayStation 4) shows exactly when various operations are performed across a single frame on the PS4's seven CPU cores.

10.8.1 Hierarchical Profiling

Computer programs written in an imperative language are inherently *hierarchical*—a function calls other functions, which in turn call still more functions. For example, let's imagine that function `a()` calls functions `b()` and `c()`, and function `b()` in turn calls functions `d()`, `e()` and `f()`. The pseudocode for this is shown below.

```
void a()
{
    b();
    c();
}

void b()
{
    d();
    e();
    f();
}

void c() { ... }
```

图 10.13. 在函数 e() 中设置断点后产生的调用堆栈。

```
void d() { ... }

void e() { ... }

void f() { ... }
```

图 10.12。假设的函数调用层次结构。

假设函数 a() 直接从 main() 调用，该函数调用层次如图 10.12 所示。

调试程序时，调用堆栈仅显示这棵树的快照。具体来说，它显示了从层次结构中当前正在执行的函数一直到树中根函数的路径。在 C/C++ 中，根函数通常是 main() 或 WinMain()，尽管从技术上讲，此函数是由标准 C 运行时库 (CRT) 中的启动函数调用的，因此该函数才是层次结构的真正根。例如，如果我们在函数 e() 中设置断点，调用堆栈将如下所示：

```
e()                 ← 当前正在执行的函数。
b()
a()
main()
_crt_startup() ← 调用层次结构的根。
```

图 10.13 将此调用堆栈描绘为从函数 e() 到函数调用树根的路径。

10.8.1.1 分层测量执行时间

如果我们测量单个函数的执行时间，我们测量的时间包括调用的任何子函数的执行时间以及它们的所有

Figure 10.13. Call stack resulting from setting a breakpoint in function `e()`.

```
void d() { ... }

void e() { ... }

void f() { ... }
```

Figure 10.12. A hypothetical function call hierarchy.

Assuming function `a()` is called directly from `main()`, this function call hierarchy is shown in Figure 10.12.

When debugging a program, the *call stack* shows only a snapshot of this tree. Specifically, it shows us the path from whichever function in the hierarchy is *currently executing* all the way to the root function in the tree. In C/C++, the root function is usually `main()` or `WinMain()`, although technically this function is called by a start-up function that is part of the standard C runtime library (CRT), so that function is the true root of the hierarchy. If we set a breakpoint in function `e()`, for example, the call stack would look something like this:

```
e()                 ← The currently executing function.
b()
a()
main()
_crt_startup()      ← Root of the call hierarchy.
```

This call stack is depicted in Figure 10.13 as a pathway from function `e()` to the root of the function call tree.

10.8.1.1 Measuring Execution Times Hierarchically

If we measure the execution time of a single function, the time we measure includes the execution time of any the child functions called and all of their

孙子、曾孙等等。为了正确解释我们可能收集的任何分析数据，我们必须确保将函数调用层次结构考虑在内。

许多商业性能分析器可以自动检测程序中的每个函数。这使得它们能够测量性能分析会话期间调用的每个函数的包含执行时间和独占执行时间。顾名思义，包含时间测量的是包含所有子函数的函数执行时间，而独占时间仅测量函数本身的执行时间。（函数的独占时间可以通过从该函数的包含时间中减去其所有直接子函数的包含时间计算得出。）此外，一些性能分析器会记录每个函数的调用次数。这是优化程序时需要掌握的重要信息，因为它可以帮助您区分内部耗费大量时间的函数和由于调用次数过多而耗费时间的函数。

相比之下，游戏内性能分析工具则不那么复杂，通常依赖于手动检测代码。如果我们的游戏引擎主循环结构足够简单，我们或许能够在粗略的层面上获取有效数据，而无需过多考虑函数调用层次结构。例如，一个典型的游戏循环可能大致如下：

```
while (!quitGame)
{
    PollJoypad();
    UpdateGameObjects();
    UpdateAllAnimations();
    PostProcessJoints();
    DetectCollisions();
    RunPhysics();
    GenerateFinalAnimationPoses();
    UpdateCameras();
    RenderScene();
    UpdateAudio();
}
```

我们可以通过测量游戏循环每个主要阶段的执行时间来粗略地分析这个游戏：

```
while (!quitGame)
{
    {
        PROFILE(SID("Poll Joypad"));
        PollJoypad();
    }
```


grandchildren, great-grandchildren and so on as well. To properly interpret any profiling data we might collect, we must be sure to take the function call hierarchy into account.

Many commercial profilers can *automatically* instrument every single function in your program. This permits them to measure both the *inclusive* and *exclusive* execution times of every function that is called during a profiling session. As the name implies, inclusive times measure the execution time of the function including all of its children, while exclusive times measure only the time spent in the function itself. (The exclusive time of a function can be calculated by subtracting the inclusive times of all its immediate children from the inclusive time of the function in question.) In addition, some profilers record how many times each function is called. This is an important piece of information to have when optimizing a program, because it allows you to differentiate between functions that eat up a lot of time internally and functions that eat up time because they are called a very large number of times.

In contrast, in-game profiling tools are not so sophisticated and usually rely on *manual* instrumentation of the code. If our game engine's main loop is structured simply enough, we may be able to obtain valid data at a coarse level without thinking much about the function call hierarchy. For example, a typical game loop might look roughly like this:

```
while (!quitGame)
{
    PollJoypad();
    UpdateGameObjects();
    UpdateAllAnimations();
    PostProcessJoints();
    DetectCollisions();
    RunPhysics();
    GenerateFinalAnimationPoses();
    UpdateCameras();
    RenderScene();
    UpdateAudio();
}
```

We could profile this game at a very coarse level by measuring the execution times of each major phase of the game loop:

```
while (!quitGame)
{
    {
        PROFILE(SID("Poll Joypad"));
        PollJoypad();
    }
```

```
    {
        PROFILE(SID("Game Object Update"));
        UpdateGameObjects();
    }
    {
        PROFILE(SID("Animation"));
        UpdateAllAnimations();
    }
    {
        PROFILE(SID("Joint Post-Processing"));
        PostProcessJoints();
    }
    {
        PROFILE(SID("Collision"));
        DetectCollisions();
    }
    {
        PROFILE(SID("Physics"));
        RunPhysics();
    }
    {
        PROFILE(SID("Animation Finaling"));
        GenerateFinalAnimationPoses();
    }
    {
        PROFILE(SID("Cameras"));
        UpdateCameras();
    }
    {
        PROFILE(SID("Rendering"));
        RenderScene();
    }
    {
        PROFILE(SID("Audio"));
        UpdateAudio();
    }
}
```

上面显示的 PROFILE() 宏很可能被实现为一个类，其构造函数启动计时器，其析构函数停止计时器并以给定的名称记录执行时间。因此，它只会对其包含块内的代码进行计时，这本质上是 C++ 在对象进入和离开作用域时自动构造和销毁的方式。

```
struct AutoProfile
{
    AutoProfile(const char* name)
```



```
    {
        PROFILE(SID("Game Object Update"));
        UpdateGameObjects();
    }
    {
        PROFILE(SID("Animation"));
        UpdateAllAnimations();
    }
    {
        PROFILE(SID("Joint Post-Processing"));
        PostProcessJoints();
    }
    {
        PROFILE(SID("Collision"));
        DetectCollisions();
    }
    {
        PROFILE(SID("Physics"));
        RunPhysics();
    }
    {
        PROFILE(SID("Animation Finaling"));
        GenerateFinalAnimationPoses();
    }
    {
        PROFILE(SID("Cameras"));
        UpdateCameras();
    }
    {
        PROFILE(SID("Rendering"));
        RenderScene();
    }
    {
        PROFILE(SID("Audio"));
        UpdateAudio();
    }
}
```

The `PROFILE()` macro shown above would probably be implemented as a class whose constructor starts the timer and whose destructor stops the timer and records the execution time under the given name. Thus, it only times the code within its containing block, by nature of the way C++ automatically constructs and destroys objects as they go in and out of scope.

```
struct AutoProfile
{
    AutoProfile(const char* name)
```

```
        {
            m_name = name;
            m_startTime = QueryPerformanceCounter();
        }

        ~AutoProfile()
        {
            std::int64_t endTime = QueryPerformanceCounter();
            std::int64_t elapsedTime = endTime - m_startTime;

            g_profileManager.storeSample(m_name, elapsedTime);
        }

        const char*   m_name;
        std::int64_t  m_startTime;
    };

    #define PROFILE(name) AutoProfile p(name)
```

这种过于简单的方法的问题在于，当在更深层次的函数调用嵌套中使用时，它会失效。例如，如果我们在 RenderScene() 函数中嵌入额外的 PROFILE() 注释，我们需要了解函数调用的层次结构，才能正确解释这些测量值。

解决这个问题的一个方法是允许注释代码的程序员指明性能分析样本之间的层级关系。例如，在 RenderScene() 函数中获取的任何 PROFILE(...) 样本都可以声明为 PROFILE(SID("Rendering")) 样本的子样本。这些关系通常与注释本身分开设置，方法是预先声明所有样本箱。例如，我们可以在引擎初始化期间按如下方式设置游戏内性能分析器：

```
    // This code declares various profile sample "bins",
    // listing the name of the bin and the name of its
    // parent bin, if any.

    ProfilerDeclareSampleBin(SID("Rendering"), nullptr);
        ProfilerDeclareSampleBin(SID("Visibility"), SID("Rendering"));
        ProfilerDeclareSampleBin(SID("Shaders"), SID("Rendering"));
            ProfilerDeclareSampleBin(SID("Materials"), SID("Shaders"));

        ProfilerDeclareSampleBin(SID("SubmitGeo"), SID("Rendering"));

    ProfilerDeclareSampleBin(SID("Audio"), nullptr);

    // ...
```



```
        {
            m_name = name;
            m_startTime = QueryPerformanceCounter();
        }

        ~AutoProfile()
        {
            std::int64_t endTime = QueryPerformanceCounter();
            std::int64_t elapsedTime = endTime - m_startTime;

            g_profileManager.storeSample(m_name, elapsedTime);
        }

        const char*   m_name;
        std::int64_t  m_startTime;
    };

    #define PROFILE(name) AutoProfile p(name)
```

The problem with this simplistic approach is that it breaks down when used within deeper levels of function call nesting. For example, if we embed additional `PROFILE()` annotations within the `RenderScene()` function, we need to understand the function call hierarchy in order to properly interpret those measurements.

One solution to this problem is to allow the programmer who is annotating the code to indicate the hierarchical interrelationships between profiling samples. For example, any `PROFILE(...)` samples taken within the `RenderScene()` function could be declared to be children of the `PROFILE(SID("Rendering"))` sample. These relationships are usually set up separately from the annotations themselves, by predeclaring all of the sample bins. For example, we might set up the in-game profiler during engine initialization as follows:

```
    // This code declares various profile sample "bins",
    // listing the name of the bin and the name of its
    // parent bin, if any.

    ProfilerDeclareSampleBin(SID("Rendering"), nullptr);
        ProfilerDeclareSampleBin(SID("Visibility"), SID("Rendering"));
        ProfilerDeclareSampleBin(SID("Shaders"), SID("Rendering"));
            ProfilerDeclareSampleBin(SID("Materials"), SID("Shaders"));

        ProfilerDeclareSampleBin(SID("SubmitGeo"), SID("Rendering"));

    ProfilerDeclareSampleBin(SID("Audio"), nullptr);

    // ...
```

这种方法仍然存在问题。具体来说，当调用层次结构中的每个函数只有一个父函数时，这种方法运行良好，但当我们尝试分析由多个父函数调用的函数时，这种方法就会失效。原因显而易见。我们静态地声明了样本箱，就好像每个函数在函数调用层次结构中只能出现一次一样，但实际上同一个函数可以在树中多次出现，每次都有不同的父函数。结果可能会产生误导性的数据，因为函数的时间会被包含在其中一个父函数箱中，但实际上应该分布在其所有父函数箱中。大多数游戏引擎不会尝试解决这个问题，因为它们主要关注的是分析那些只从函数调用层次结构中的特定位置调用的粗粒度函数。但是，在使用大多数游戏引擎中常见的那种简单的引擎内配置文件来分析代码时，需要注意这个限制。

当然，我们也可以编写一个更复杂的性能分析系统，来妥善处理 AutoProfile 的嵌套实例。这只是设计游戏引擎时需要考虑的诸多因素之一。我们应该投入大量的工程时间来创建一个完全分层的分析器吗？还是应该将就使用更简单的方案，并将这些编程资源投入到其他地方？最终，这取决于你。

我们还想计算给定函数被调用的次数。
在上面的例子中，我们知道我们分析的每个函数每帧都恰好调用一次。但是，函数调用层次结构中更深层的其他函数每帧可能被调用多次。如果我们测量函数 x() 的执行时间为 2 毫秒，那么重要的是要知道它本身的执行时间是 2 毫秒，还是它的执行时间是 2 毫秒，但它在该帧期间被调用了 1,000 次。跟踪每帧函数的调用次数非常简单——分析系统只需在每次接收到样本时递增计数器，并在每帧开始时重置计数器即可。

10.8.2 导出到 Excel

一些游戏引擎允许将游戏内性能分析器捕获的数据转储到文本文件中，以便后续分析。我发现逗号分隔值 (CSV) 格式是最好的，因为这种文件可以轻松加载到 Microsoft Excel 电子表格中，从而可以以多种方式操作和分析数据。我为《荣誉勋章：太平洋突袭》引擎编写了一个这样的导出器。列对应于各种带注释的块，每一行代表在游戏运行过程中某一帧采集的性能分析样本。

This approach still has its problems. Specifically, it works well when every function in the call hierarchy has only one parent, but it breaks down when we try to profile a function that is called by more than one parent function. The reason for this should be pretty obvious. We're statically declaring our sample bins *as if* every function can only appear once in the function call hierarchy, but actually the same function can reappear many times in the tree, each time with a different parent. The result can be misleading data, because a function's time will be included in one of the parent bins, but really should be distributed across all of its parents' bins. Most game engines don't make an attempt to remedy this problem, since they are primarily interested in profiling coarse-grained functions that are only called from one specific location in the function call hierarchy. But this limitation is something to be aware of when profiling your code with a simple in-engine profile of the sort found in most game engines.

Of course, it is also possible to write a much more sophisticated profiling system that handles nested instances of AutoProfile properly. This is an example of the many trade-offs one makes when designing a game engine. Do we invest the engineering time to create a fully hierarchical profiler? Or, do we make do with something simpler and invest those programming resources elsewhere? Ultimately, it's up to you.

We would also like to account for how many *times* a given function is called. In the example above, we know that each of the functions we profiled is called exactly once per frame. But other functions, deeper in the function call hierarchy, may be called more than once per frame. If we measure function `x()` to take 2 ms to execute, it's important to know whether it takes 2 ms to execute on its own, or whether its execution time is 2 ms but it was called 1,000 times during the frame. Keeping track of the number of times a function is called per frame is quite simple—the profiling system can simply increment a counter each time a sample is received and reset the counters at the start of each frame.

10.8.2 Exporting to Excel

Some game engines permit the data captured by the in-game profiler to be dumped to a text file for subsequent analysis. I find that a comma-separated values (CSV) format is best, because such files can be loaded easily into a Microsoft Excel spreadsheet, where the data can be manipulated and analyzed in myriad ways. I wrote such an exporter for the *Medal of Honor: Pacific Assault* engine. The columns corresponded to the various annotated blocks, and each row represented the profiling sample taken during one frame of the game's

执行。第一列包含帧数，第二列包含实际游戏时间（以秒为单位）。这使得团队能够绘制性能统计数据随时间变化的图表，并确定每帧实际执行的时间。通过在导出的电子表格中添加一些简单的公式，我们可以计算帧速率、执行时间百分比等等。

10.9 游戏内内存统计和泄漏检测

除了运行时性能（即帧速率）之外，大多数游戏引擎还受到目标硬件可用内存量的限制。PC 游戏受此类限制的影响最小，因为现代 PC 拥有先进的虚拟内存管理器。但即使是 PC 游戏，也会受到其所谓"最低配置"机器的内存限制——即发行商承诺并在游戏包装上注明的、保证游戏运行的最低配置机器。

因此，大多数游戏引擎都实现了自定义的内存跟踪工具。这些工具允许开发者查看每个引擎子系统的内存使用情况，以及是否存在内存泄漏（即分配了内存却从未释放）。掌握这些信息非常重要，这样您在尝试减少游戏内存使用量时就能做出明智的决定，使其能够适配目标主机或PC类型。

跟踪游戏实际使用的内存量可能出乎意料地棘手。你可能认为只需将 malloc() / free() 或 new / delete 包装成一对函数或宏，就能跟踪分配和释放的内存量。然而，由于以下几个原因，事情远没有那么简单：

1. 你通常无法控制其他人代码的内存分配行为。除非你完全从头编写操作系统、驱动程序和游戏引擎，否则你很有可能最终会将你的游戏与至少一些第三方库链接起来。大多数优秀的库都提供了内存分配钩子，以便你可以用自己的分配器替换它们的分配器。但有些库并没有这样做。跟踪游戏引擎中使用的每个第三方库分配的内存通常很困难——但如果你在选择第三方库时足够彻底和谨慎，通常可以做到。

2. 内存有不同的类型。例如，PC 有两种 RAM：主 RAM 和视频 RAM（驻留在图形处理器上的内存）。

execution. The first column contained frame numbers and the second actual game time measured in seconds. This allowed the team to graph how the performance statistics varied over time and to determine how long each frame actually took to execute. By adding some simple formulae to the exported spreadsheet, we could calculate frame rates, execution time percentages and so on.

10.9 In-Game Memory Stats and Leak Detection

In addition to runtime performance (i.e., frame rate), most game engines are also constrained by the amount of memory available on the target hardware. PC games are least affected by such constraints, because modern PCs have sophisticated virtual memory managers. But even PC games are constrained by the memory limitations of their so-called "min spec" machine—the least-powerful machine on which the game is guaranteed to run, as promised by the publisher and stated on the game's packaging.

For this reason, most game engines implement custom memory-tracking tools. These tools allow the developers to see how much memory is being used by each engine subsystem and whether or not any memory is leaking (i.e., memory is allocated but never freed). It's important to have this information so that you can make informed decisions when trying to cut back the memory usage of your game so that it will fit onto the console or type of PC you are targeting.

Keeping track of how much memory a game actually uses can be a surprisingly tricky job. You'd think you could simply wrap `malloc()/free()` or `new/delete` in a pair of functions or macros that keep track of the amount of memory that is allocated and freed. However, it's never that simple for a few reasons:

1. *You often can't control the allocation behavior of other people's code.* Unless you write the operating system, drivers and the game engine entirely from scratch, there's a good chance you're going to end up linking your game with at least some third-party libraries. Most good libraries provide *memory allocation hooks* so that you can replace their allocators with your own. But some do not. It's often difficult to keep track of the memory allocated by each and every third-party library you use in your game engine—but it usually *can* be done if you're thorough and selective in your choice of third-party libraries.

2. *Memory comes in different flavors.* For example, a PC has two kinds of RAM: main RAM and video RAM (the memory residing on your graph-

ics 卡，主要用于存储几何和纹理数据。即使您设法跟踪主 RAM 中发生的所有内存分配和释放，也几乎不可能跟踪视频 RAM 的使用情况。这是因为像 DirectX 这样的图形 API 实际上隐藏了视频 RAM 的分配和使用细节，不让开发者知晓。在游戏机上，情况会稍微好一些，只是因为您通常需要自己编写视频 RAM 管理器。这比使用 DirectX 更难，但至少您完全了解正在发生的事情。

3. 分配器种类繁多。许多游戏使用专门的分配器来实现各种目的。例如，顽皮狗引擎有一个用于通用分配的全局堆，一个用于管理游戏对象在游戏世界中生成和销毁时创建的内存的特殊堆，一个用于在游戏过程中将数据流式传输到内存中的关卡加载堆，一个用于单帧分配的堆栈分配器（堆栈每帧自动清除），一个用于视频RAM的分配器，以及一个仅用于最终发行游戏中不需要的分配的调试内存堆。每个分配器在游戏启动时都会抓取一大块内存，然后自行管理该内存块。如果我们要跟踪所有对new和delete的调用，我们会发现这六个分配器各有一个new，仅此而已。要获取任何有用的信息，我们实际上需要跟踪每个分配器内存块中的所有分配。

大多数专业游戏团队都投入了大量精力来开发引擎内内存跟踪工具，以提供准确详细的信息。这些工具通常以多种形式提供输出。例如，引擎可能会生成游戏在特定时间段内所有内存分配的详细转储。这些数据可能包括每个内存分配器或每个游戏系统的高位标记，指示每个分配器或游戏系统所需的最大物理 RAM 量。一些引擎还会在游戏运行时提供内存使用情况的抬头显示。这些数据可能是表格形式（如图 10.14 所示），也可能是图形形式（如图 10.15 所示）。

此外，当出现内存不足或内存不足的情况时，优秀的引擎会尽可能提供有用的信息。开发 PC 游戏时，游戏团队通常会使用性能强大的 PC，其内存比目标最低配置的机器更大。同样，主机游戏则使用特殊的开发套件开发，其内存比零售主机更大。因此，在这两种情况下，即使游戏运行速度慢，游戏也能继续运行。

ics card, which is used primarily for geometry and texture data). Even if you manage to track all of the memory allocations and deallocations occurring within main RAM, it can be well neigh impossible to track video RAM usage. This is because graphics APIs like DirectX actually hide the details of how video RAM is being allocated and used from the developer. On a console, life is a bit easier only because you often end up having to write a video RAM manager yourself. This is more difficult than using DirectX, but at least you have complete knowledge of what's going on.

3. *Allocators come in different flavors*. Many games make use of specialized allocators for various purposes. For example, the Naughty Dog engine has a *global heap* for general-purpose allocations, a special heap for managing the memory created by *game objects* as they spawn into the game world and are destroyed, a *level-loading heap* for data that is *streamed* into memory during gameplay, a stack allocator for *single-frame allocations* (the stack is cleared automatically every frame), an allocator for *video RAM*, and a *debug memory heap* used only for allocations that will not be needed in the final shipping game. Each of these allocators grabs a large hunk of memory when the game starts up and then manages that memory block itself. If we were to track all the calls to new and delete, we'd see one new for each of these six allocators and that's all. To get any useful information, we really need to track all of the allocations *within* each of these allocators' memory blocks.

Most professional game teams expend a significant amount of effort on creating in-engine memory-tracking tools that provide accurate and detailed information. The resulting tools usually provide their output in a variety of forms. For example, the engine might produce a detailed dump of all memory allocations made by the game during a specific period of time. The data might include high water marks for each memory allocator or each game system, indicating the maximum amount of physical RAM required by each. Some engines also provide heads-up displays of memory usage while the game is running. This data might be tabular, as shown in Figure 10.14, or graphical as shown in Figure 10.15.

In addition, when low-memory or out-of-memory conditions arise, a good engine will provide this information in as helpful a way as possible. When PC games are developed, the game team usually works on high-powered PCs with more RAM than the min-spec machine being targeted. Likewise, console games are developed on special *development kits* that have more memory than a retail console. So in both cases, the game can continue to run even when it

图 10.14.顽皮狗引擎的表格内存统计信息。

图 10.15. 图形内存使用情况显示，同样来自《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4。

从技术上讲，游戏内存已经耗尽（即不再适合零售主机或低配PC）。当出现这种内存不足的情况时，游戏引擎会显示类似“内存不足——此关卡无法在零售主机上运行”的消息。

游戏引擎的内存跟踪系统还有很多其他方法可以帮助开发人员尽早、尽可能方便地查明问题

Figure 10.14. Tabular memory statistics from Naughty Dog's engine.

Figure 10.15. A graphical memory usage display, also from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

technically has run out of memory (i.e., would no longer fit on a retail console or min-spec PC). When this kind of out-of-memory condition arises, the game engine can display a message saying something like, "Out of memory—this level will not run on a retail system."

There are lots of other ways in which a game engine's memory tracking system can aid developers in pinpointing problems as early and as conveniently

尽可能。以下仅举几个例子：

- 如果模型加载失败，则会在游戏世界中该对象原本所在的位置以 3D 形式显示一个鲜红色的文本字符串。
- 如果纹理加载失败，对象可能会被绘制成难看的粉红色纹理，这显然不是最终游戏的一部分。
- 如果动画加载失败，角色可能会摆出一个特殊的（可能是幽默的）姿势来表示缺少动画，并且缺少的资产的名称可能会悬停在角色的头上。

提供良好的内存分析工具的关键是 (a) 提供准确的信息，(b) 以方便的方式呈现数据，使问题显而易见，以及 (c) 提供上下文信息以帮助团队在问题发生时追踪问题的根本原因。

as possible. Here are just a few examples:

- If a model fails to load, a bright red text string could be displayed in 3D hovering in the game world where that object would have been.
- If a texture fails to load, the object could be drawn with an ugly pink texture that is very obviously not part of the final game.
- If an animation fails to load, the character could assume a special (possibly humorous) pose that indicates a missing animation, and the name of the missing asset could hover over the character's head.

The key to providing good memory analysis tools is (a) to provide accurate information, (b) to present the data in a way that is convenient and that makes problems obvious and (c) to provide contextual information to aid the team in tracking down the root cause of problems when they occur.

第三部分 图形、动作和声音

Part III Graphics, Motion and Sound

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

11
渲染引擎

大多数人想到电脑和视频游戏时，首先想到的都是令人惊叹的三维图形。实时3D渲染是一个非常广泛而深奥的主题，因此根本不可能在一章之内涵盖所有细节。值得庆幸的是，关于这个主题有很多优秀的书籍和其他资源。事实上，在构成游戏引擎的所有技术中，实时3D图形或许是涵盖最全面的技术之一。因此，本章的目标是帮助您全面了解实时渲染技术，并作为进一步学习的起点。读完这些内容后，您会发现阅读其他关于3D图形的书籍就像是一次熟悉的旅程。您甚至可能在聚会上给您的朋友留下深刻印象（……或者疏远他们……）。

我们将首先扎实掌握实时 3D 渲染引擎的基本概念、理论和数学知识。接下来，我们将了解将理论框架转化为现实所需的软件和硬件流水线。我们将讨论一些最常见的优化技术，并了解它们如何驱动大多数引擎中的工具流水线结构和运行时渲染 API。最后，我们将概述当今游戏引擎中使用的一些高级渲染技术和光照模型。在本章中，我将推荐一些我最喜欢的书籍和

11
The Rendering Engine

When most people think about computer and video games, the first thing that comes to mind is the stunning three-dimensional graphics. Real-time 3D rendering is an exceptionally broad and profound topic, so there's simply no way to cover all of the details in a single chapter. Thankfully there are a great many excellent books and other resources available on this topic. In fact, real-time 3D graphics is perhaps one of the best covered of all the technologies that make up a game engine. The goal of this chapter, then, is to provide you with a broad understanding of real-time rendering technology and to serve as a jumping-off point for further learning. After you've read through these pages, you should find that reading other books on 3D graphics seems like a journey through familiar territory. You might even be able to impress your friends at parties (…or alienate them…).

We'll begin by laying a solid foundation in the concepts, theory and mathematics that underlie any real-time 3D rendering engine. Next, we'll have a look at the software and hardware pipelines used to turn this theoretical framework into reality. We'll discuss some of the most common optimization techniques and see how they drive the structure of the tools pipeline and the runtime rendering API in most engines. We'll end with a survey of some of the advanced rendering techniques and lighting models in use by game engines today. Throughout this chapter, I'll point you to some of my favorite books and

其他资源应该可以帮助您更深入地理解我们将在此介绍的主题。

11.1 深度缓冲三角形光栅化基础

归根结底，渲染三维场景涉及以下基本步骤：

- 虚拟场景通常以某种数学形式表示的 3D 表面来描述。
- 虚拟摄像机的定位和方向可生成所需的场景视图。通常，摄像机被建模为一个理想化的焦点，其前方一小段距离处有一个成像表面，该表面由与目标显示设备的图像元素（像素）对应的虚拟光传感器组成。
- 定义各种光源。这些光源提供所有与环境中的物体相互作用、反射的光线，最终到达虚拟相机的图像感应表面。
- 描述场景中表面的视觉属性。这定义了光线如何与每个表面相互作用。
- 对于成像矩形内的每个像素，渲染引擎计算通过该像素汇聚到虚拟相机焦点的光线的颜色和强度。这被称为求解渲染方程（也称为着色方程）。

图 11.1 描述了这个高级渲染过程。
可以使用多种不同的技术来执行上述基本渲染步骤。主要目标通常是照片级真实感，尽管有些游戏追求更具风格的外观（例如卡通、炭笔素描、水彩画等）。因此，渲染工程师和艺术家通常会尝试尽可能真实地描述场景的属性，并使用尽可能贴近物理现实的光传输模型。在此背景下，渲染技术的范围涵盖了从以牺牲视觉保真度为代价追求实时性能的技术，到以照片级真实感为设计目标但不打算实时运行的技术。

实时渲染引擎重复执行上述步骤，以每秒 30、50 或 60 帧的速率显示渲染图像，以提供

other resources that should help you to gain an even deeper understanding of the topics we'll cover here.

11.1 Foundations of Depth-Buffered Triangle Rasterization

When you boil it down to its essence, rendering a three-dimensional scene involves the following basic steps:

- A *virtual scene* is described, usually in terms of 3D surfaces represented in some mathematical form.
- A *virtual camera* is positioned and oriented to produce the desired view of the scene. Typically the camera is modeled as an idealized focal point, with an imaging surface hovering some small distance in front of it, composed of *virtual light sensors* corresponding to the picture elements (*pixels*) of the target display device.
- Various *light sources* are defined. These sources provide all the light rays that will interact with and reflect off the objects in the environment and eventually find their way onto the image-sensing surface of the virtual camera.
- The *visual properties* of the surfaces in the scene are described. This defines how light should interact with each surface.
- For each pixel within the imaging rectangle, the rendering engine calculates the color and intensity of the light ray(s) converging on the virtual camera's focal point through that pixel. This is known as *solving the rendering equation* (also called the *shading equation*).

This high-level rendering process is depicted in Figure 11.1.

Many different technologies can be used to perform the basic rendering steps described above. The primary goal is usually *photorealism*, although some games aim for a more stylized look (e.g., cartoon, charcoal sketch, watercolor and so on). As such, rendering engineers and artists usually attempt to describe the properties of their scenes as realistically as possible and to use light transport models that match physical reality as closely as possible. Within this context, the gamut of rendering technologies ranges from techniques designed for real-time performance at the expense of visual fidelity, to those designed for photorealism but which are not intended to operate in real time.

Real-time rendering engines perform the steps listed above repeatedly, displaying rendered images at a rate of 30, 50 or 60 frames per second to provide

图 11.1 几乎所有 3D 计算机图形技术使用的高级渲染方法。

运动的幻觉。这意味着实时渲染引擎最多有 33.3 毫秒来生成每幅图像（以达到 30 FPS 的帧率）。通常可用的时间要少得多，因为带宽也会被其他引擎系统（例如动画、AI、碰撞检测、物理模拟、音频、玩家机制和其他游戏逻辑）所消耗。考虑到电影渲染引擎渲染一帧通常需要几分钟甚至几小时的时间，如今实时计算机图形的质量确实令人惊叹。

11.1.1 描述场景

现实世界的场景由物体组成。有些物体是实体，例如砖块；有些物体是无定形的，例如烟雾，但每个物体都占据着三维空间的一定体积。物体可能是不透明的（在这种情况下，光线无法穿过其体积）；透明的（在这种情况下，光线可以穿过物体而不会发生散射，因此我们可以看到物体后方物体的清晰图像）；或者半透明的（这意味着光线可以穿过物体，但在此过程中会向各个方向散射，只产生模糊的颜色，从而暗示其后方物体）。

不透明物体的渲染只需考虑其表面即可。我们不需要知道不透明物体的内部结构，因为光线无法穿透其表面。渲染透明或半透明物体时，我们实际上应该模拟光线在穿过物体内部时如何反射、折射、散射和吸收。这需要了解物体的内部结构和属性。然而，

Figure 11.1. The high-level rendering approach used by virtually all 3D computer graphics technologies.

the illusion of motion. This means a real-time rendering engine has at most 33.3 ms to generate each image (to achieve a frame rate of 30 FPS). Usually much less time is available, because bandwidth is also consumed by other engine systems like animation, AI, collision detection, physics simulation, audio, player mechanics and other gameplay logic. Considering that film rendering engines often take anywhere from many minutes to many hours to render a single frame, the quality of real-time computer graphics these days is truly astounding.

11.1.1 Describing a Scene

A real-world scene is composed of objects. Some objects are solid, like a brick, and some are amorphous, like a cloud of smoke, but every object occupies a volume of 3D space. An object might be *opaque* (in which case light cannot pass through its volume), *transparent* (in which case light passes through it without being scattered, so that we can see a reasonably clear image of whatever is behind the object), or *translucent* (meaning that light can pass through the object but is scattered in all directions in the process, yielding only a blur of colors that hint at the objects behind it).

Opaque objects can be rendered by considering only their *surfaces*. We don't need to know what's inside an opaque object in order to render it, because light cannot penetrate its surface. When rendering a transparent or translucent object, we really should model how light is reflected, refracted, scattered and absorbed as it passes through the object's volume. This requires knowledge of the interior structure and properties of the object. However,

大多数游戏引擎不会费这么大劲。它们只是以几乎与渲染不透明物体相同的方式渲染透明和半透明物体的表面。一种称为 alpha 的简单数值不透明度度量用于描述表面的不透明度或透明度。这种方法可能会导致各种视觉异常（例如，物体远端的表面特征可能渲染不正确），但在许多情况下可以使近似值看起来相当逼真。即使是像烟雾云这样的无定形物体也经常使用粒子效果来表示，这些效果通常由大量半透明的矩形卡片组成。因此，可以肯定地说，大多数游戏渲染引擎主要关注的是渲染表面。

11.1.1.1 高端渲染包使用的表示

理论上，曲面是由三维空间中无限个点组成的二维平面。然而，这样的描述显然不切实际。为了让计算机能够处理和渲染任意曲面，我们需要一种简洁的数值表示方法。

某些曲面可以用参数曲面方程精确地以解析形式描述。例如，以原点为中心的球体可以用方程 $x^2 + y^2 + z^2 = r^2$ 表示。然而，参数方程对于任意形状的建模并不是特别有用。

在电影行业中，曲面通常由一系列矩形面片表示，每个面片由一条由少量控制点定义的二维样条曲线构成。各种样条曲线都有使用，包括贝塞尔曲面（例如，双三次面片，即三阶贝塞尔曲面——更多信息请参见 http://en.wikipedia.org/wiki/Bezier_surface）、非均匀有理 B 样条曲线（NURBS——更多信息请参见 http://en.wikipedia.org/wiki/Nurbs）、贝塞尔三角形和 N 面片（也称为法线面片——更多信息请参见 http://ubm.io/1iGnvJ5）。使用面片建模有点像用小块布料或纸糊覆盖雕像。

像皮克斯的RenderMan这样的高端电影渲染引擎使用细分曲面来定义几何形状。每个曲面都由控制多边形网格（类似于样条曲线）表示，但可以使用Catmull-Clark算法将这些多边形细分为越来越小的多边形。这种细分通常会持续到单个多边形的尺寸小于单个像素为止。这种方法最大的优点在于，无论摄像机距离曲面有多近，它都可以进一步细分，从而使其轮廓边缘看起来不会像多面体。要了解更多关于细分曲面的信息，请查看以下精彩文章：http://ubm.io/1lx6th5。

most game engines don't go to all that trouble. They just render the surfaces of transparent and translucent objects in almost the same way opaque objects are rendered. A simple numeric opacity measure known as *alpha* is used to describe how opaque or transparent a surface is. This approach can lead to various visual anomalies (for example, surface features on the far side of the object may be rendered incorrectly), but the approximation can be made to look reasonably realistic in many cases. Even amorphous objects like clouds of smoke are often represented using particle effects, which are typically composed of large numbers of semitransparent rectangular cards. Therefore, it's safe to say that most game rendering engines are primarily concerned with rendering *surfaces*.

11.1.1.1 Representations Used by High-End Rendering Packages

Theoretically, a surface is a two-dimensional sheet comprised of an infinite number of points in three-dimensional space. However, such a description is clearly not practical. In order for a computer to process and render arbitrary surfaces, we need a compact way to represent them numerically.

Some surfaces can be described exactly in analytical form using a *parametric surface equation*. For example, a sphere centered at the origin can be represented by the equation $x^2 + y^2 + z^2 = r^2$. However, parametric equations aren't particularly useful for modeling arbitrary shapes.

In the film industry, surfaces are often represented by a collection of rectangular *patches* each formed from a two-dimensional spline defined by a small number of control points. Various kinds of splines are used, including Bézier surfaces (e.g., bicubic patches, which are third-order Béziers—see http://en.wikipedia.org/wiki/Bezier_surface for more information), nonuniform rational B-splines (NURBS—see http://en.wikipedia.org/wiki/Nurbs), Bézier triangles and *N*-patches (also known as *normal patches*—see http://ubm.io/1iGnvJ5 for more details). Modeling with patches is a bit like covering a statue with little rectangles of cloth or paper maché.

High-end film rendering engines like Pixar's RenderMan use *subdivision surfaces* to define geometric shapes. Each surface is represented by a mesh of control polygons (much like a spline), but the polygons can be subdivided into smaller and smaller polygons using the Catmull-Clark algorithm. This subdivision typically proceeds until the individual polygons are smaller than a single pixel in size. The biggest benefit of this approach is that no matter how close the camera gets to the surface, it can always be subdivided further so that its silhouette edges won't look faceted. To learn more about subdivision surfaces, check out the following great article: http://ubm.io/1lx6th5.

11.1.1.2 三角形网格

游戏开发者传统上使用三角形网格来建模表面。三角形可以作为表面的分段线性近似，就像一串相互连接的线段可以作为函数或曲线的分段近似一样（参见图 11.2）。

图 11.2 三角形网格是曲面的线性近似，就像一系列连接的线段可以作为函数或曲线的线性近似一样。

三角形是实时渲染的首选多边形，因为它们具有以下理想属性：

- 三角形是最简单的多边形。如果顶点少于三个，就根本不会有曲面。
- 三角形始终是平面的。任何具有四个或更多顶点的多边形不一定具有此属性，因为虽然前三个顶点定义一个平面，但第四个顶点可能位于该平面的上方或下方。
- 三角形在大多数变换下仍然是三角形，包括仿射变换和透视投影。在最坏的情况下，从边缘观察的三角形会退化为线段。在其他所有方向上，它仍然是三角形。
- 几乎所有商用图形加速硬件都是围绕三角形光栅化设计的。从最早的 PC 3 D 图形加速器开始，渲染硬件几乎完全围绕三角形光栅化设计。这一决策可以追溯到最早的 3D 游戏（例如《德军总部 3D》和《毁灭战士》）中使用的首个软件光栅化器。无论你是否喜欢，基于三角形的技术已在我们的行业中根深蒂固，并且很可能在未来几年内仍将如此。

镶嵌

术语“镶嵌”描述了将表面分割成一系列离散多边形（通常是四边形，也称为四边形，或三角形）的过程。三角剖分就是将表面镶嵌成三角形。

游戏中使用的三角形网格的一个问题是，其镶嵌程度在艺术家创建时就已经固定了。固定的镶嵌程度会导致物体轮廓边缘看起来块状，如图 11.3 所示；当物体靠近相机时，这种情况尤其明显。

理想情况下，我们希望找到一种解决方案，能够随着物体靠近虚拟相机而任意增加曲面细分的程度。换句话说，我们希望无论物体距离多远，三角形与像素的密度都保持一致。细分曲面可以实现这一理想效果——曲面可以根据与相机的距离进行曲面细分，从而使每个三角形的尺寸都小于一个像素。

11.1.1.2 Triangle Meshes

Game developers have traditionally modeled their surfaces using triangle meshes. Triangles serve as a piecewise linear approximation to a surface, much as a chain of connected line segments acts as a piecewise approximation to a function or curve (see Figure 11.2).

Figure 11.2. A mesh of triangles is a linear approximation to a surface, just as a series of connected line segments can serve as a linear approximation to a function or curve.

Triangles are the polygon of choice for real-time rendering because they have the following desirable properties:

- *The triangle is the simplest type of polygon*. Any fewer than three vertices, and we wouldn't have a surface at all.
- *A triangle is always planar*. Any polygon with four or more vertices need not have this property because, while the first three vertices define a plane, the fourth vertex might lie above or below that plane.
- *Triangles remain triangles under most kinds of transformations, including affine transforms and perspective projections*. At worst, a triangle viewed edge-on will degenerate into a line segment. At every other orientation, it remains triangular.
- *Virtually all commercial graphics-acceleration hardware is designed around triangle rasterization*. Starting with the earliest 3D graphics accelerators for the PC, rendering hardware has been designed almost exclusively around triangle rasterization. This decision can be traced all the way back to the first software rasterizers used in the earliest 3D games like *Castle Wolfenstein 3D* and *Doom*. Like it or not, triangle-based technologies are entrenched in our industry and probably will be for years to come.

Tessellation

The term *tessellation* describes a process of dividing a surface up into a collection of discrete polygons (which are usually either quadrilaterals, also known as *quads*, or triangles). *Triangulation* is tessellation of a surface into triangles.

One problem with the kind of triangle mesh used in games is that its level of tessellation is fixed by the artist when he or she creates it. Fixed tessellation can cause an object's *silhouette edges* to look blocky, as shown in Figure 11.3; this is especially noticeable when the object is close to the camera.

Ideally, we'd like a solution that can arbitrarily increase tessellation as an object gets closer to the virtual camera. In other words, we'd like to have a uniform triangle-to-pixel density, no matter how close or far away the object is. Subdivision surfaces can achieve this ideal—surfaces can be tessellated based on distance from the camera, so that every triangle is less than one pixel in size.

图 11.3. 固定镶嵌会导致物体的轮廓边缘看起来块状，尤其是当物体靠近相机时。

游戏开发者经常尝试通过创建每个三角形网格的一系列替代版本（每个版本称为一个细节级别 (LOD)）来近似实现这种理想的均匀三角形像素密度。第一个 LOD 通常称为 LOD 0，代表最高级别的曲面细分；当物体非常靠近相机时使用。后续 LOD 的曲面细分分辨率会越来越低（参见图 11.4）。随着物体远离相机，引擎会从 LOD 0 切换到 LOD 1、LOD 2，依此类推。这使得渲染引擎可以将大部分时间用于变换和照亮最靠近相机的物体的顶点（因此在屏幕上占据的像素最多）。

一些游戏引擎会将动态曲面细分技术应用于水体或地形等广阔的网格。在这种技术中，网格通常用定义在某种规则网格图案上的高度场来表示。网格中距离摄像机最近的区域会被曲面细分到网格的全分辨率。距离摄像机较远的区域则使用越来越少的网格点进行曲面细分。

渐进式网格是另一种动态曲面细分和 LODing 技术。使用这种技术，当物体非常靠近相机时，会创建一个高分辨率网格用于显示。（这本质上是

图 11.4。一系列 LOD 网格，每个网格都具有固定的细分级别，可用于近似均匀的三角形像素密度。最左边的圆环由 5000 个三角形构成，中间的圆环由 450 个三角形构成，最右边的圆环由 200 个三角形构成。

Figure 11.3. Fixed tessellation can cause an object's silhouette edges to look blocky, especially when the object is close to the camera.

Game developers often attempt to approximate this ideal of uniform triangle-to-pixel density by creating a chain of alternate versions of each triangle mesh, each known as a *level of detail* (LOD). The first LOD, often called LOD 0, represents the highest level of tessellation; it is used when the object is very close to the camera. Subsequent LODs are tessellated at lower and lower resolutions (see Figure 11.4). As the object moves farther away from the camera, the engine switches from LOD 0 to LOD 1 to LOD 2 and so on. This allows the rendering engine to spend the majority of its time transforming and lighting the vertices of the objects that are closest to the camera (and therefore occupy the largest number of pixels on-screen).

Some game engines apply *dynamic tessellation* techniques to expansive meshes like water or terrain. In this technique, the mesh is usually represented by a height field defined on some kind of regular grid pattern. The region of the mesh that is closest to the camera is tessellated to the full resolution of the grid. Regions that are farther away from the camera are tessellated using fewer and fewer grid points.

Progressive meshes are another technique for dynamic tessellation and LODing. With this technique, a single high-resolution mesh is created for display when the object is very close to the camera. (This is essentially the

Figure 11.4. A chain of LOD meshes, each with a fixed level of tessellation, can be used to approximate uniform triangle-to-pixel density. The leftmost torus is constructed from 5000 triangles, the center torus from 450 triangles and the rightmost torus from 200 triangles.

LOD 0 网格。）随着物体距离的增加，该网格会通过折叠某些边缘自动进行去曲面细分。实际上，此过程会自动生成一个半连续的 LOD 链。有关渐进式网格技术的详细讨论，请参阅 http://research.microsoft. com/en-us/um/people/hoppe/pm.pdf。

11.1.1.3 构建三角形网格

现在我们了解了三角形网格是什么以及为什么使用它们，让我们简单了解一下它们是如何构造的。

缠绕顺序

三角形由其三个顶点的位置向量定义，我们将其分别记为 p 1 、 p 2 和 p 3 。三角形的边可以通过相邻顶点的位置向量相减来求得。例如：

$$\mathbf{e}_{12} = \mathbf{p}_2 - \mathbf{p}_1,$$
$$\mathbf{e}_{13} = \mathbf{p}_3 - \mathbf{p}_1,$$
$$\mathbf{e}_{23} = \mathbf{p}_3 - \mathbf{p}_2.$$

任意两条边的归一化叉积定义一个单位面法线 N：

$$\mathbf{N} = \frac{\mathbf{e}_{12} \times \mathbf{e}_{13}}{|\mathbf{e}_{12} \times \mathbf{e}_{13}|}.$$

这些推导如图 11.5 所示。为了确定面法线的方向（即边叉积的方向），我们需要定义三角形的哪一侧应被视为正面（即物体的外表面），哪一侧应被视为背面（即物体的内表面）。这可以通过指定环绕顺序来轻松定义——顺时针（CW）或逆时针 (CCW)。

大多数底层图形 API 都提供了根据缠绕顺序剔除背面三角形的方法。例如，如果我们在

图 11.5. 从三角形的顶点导出三角形的边和平面。

LOD 0 mesh.) This mesh is automatically detessellated as the object gets farther away by collapsing certain edges. In effect, this process automatically generates a semi-continuous chain of LODs. See http://research.microsoft.com/en-us/um/people/hoppe/pm.pdf for a detailed discussion of progressive mesh technology.

11.1.1.3 Constructing a Triangle Mesh

Now that we understand what triangle meshes are and why they're used, let's take a brief look at how they're constructed.

Winding Order

A triangle is defined by the position vectors of its three vertices, which we can denote $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. The edges of a triangle can be found by simply subtracting the position vectors of adjacent vertices. For example,

$$\mathbf{e}_{12} = \mathbf{p}_2 - \mathbf{p}_1,$$
$$\mathbf{e}_{13} = \mathbf{p}_3 - \mathbf{p}_1,$$
$$\mathbf{e}_{23} = \mathbf{p}_3 - \mathbf{p}_2.$$

The normalized cross product of any two edges defines a unit *face normal* **N**:

$$\mathbf{N} = \frac{\mathbf{e}_{12} \times \mathbf{e}_{13}}{|\mathbf{e}_{12} \times \mathbf{e}_{13}|}.$$

These derivations are illustrated in Figure 11.5. To know the *direction* of the face normal (i.e., the sense of the edge cross product), we need to define which side of the triangle should be considered the front (i.e., the outside surface of an object) and which should be the back (i.e., its inside surface). This can be defined easily by specifying a *winding order*—clockwise (CW) or counterclockwise (CCW).

Most low-level graphics APIs give us a way to *cull* back-facing triangles based on winding order. For example, if we set the cull mode parameter in

Figure 11.5. Deriving the edges and plane of a triangle from its vertices.

Direct3D（D3DRS_CULL）设置为 D3DCULLMODE_CW，则任何顶点在屏幕空间中以顺时针方式缠绕的三角形都将被视为背面三角形，并且不会被绘制。

背面剔除非常重要，因为我们通常不想浪费时间绘制那些无论如何都看不见的三角形。此外，渲染透明物体的背面实际上会导致视觉异常。缠绕顺序的选择是任意的，但当然，它必须在整个游戏的所有资源中保持一致。缠绕顺序不一致是初级 3D 建模人员常犯的一个错误。

三角列表

定义网格最简单的方法是将顶点以三个一组的形式列出，每个三元组对应一个三角形。这种数据结构称为三角形列表；如图 11.6 所示。

图 11.6。三角形列表。

索引三角形列表

你可能注意到，图 11.6 所示的三角形列表中的许多顶点都是重复的，而且经常是多次重复。正如我们将在 11.1.2.1 节中看到的那样，我们通常会为每个顶点存储大量元数据，因此在三角形列表中重复这些数据会浪费内存。这还会浪费 GPU 带宽，因为重复的顶点会被多次变换和点亮。

出于这些原因，大多数渲染引擎使用一种更高效的数据结构，称为索引三角形列表。其基本思想是一次性列出所有顶点，不重复，然后使用轻量级顶点索引（通常每个仅占用 16 位）来定义构成三角形的顶点三元组。

Direct3D (`D3DRS_CULL`) to `D3DCULLMODE_CW`, then any triangle whose vertices wind in a clockwise fashion in screen space will be treated as a back-facing triangle and will not be drawn.

Back-face culling is important because we generally don't want to waste time drawing triangles that aren't going to be visible anyway. Also, rendering the back faces of transparent objects can actually cause visual anomalies. The choice of winding order is an arbitrary one, but of course it must be consistent across all assets in the entire game. Inconsistent winding order is a common error among junior 3D modelers.

Triangle Lists

The easiest way to define a mesh is simply to list the vertices in groups of three, each triple corresponding to a single triangle. This data structure is known as a *triangle list*; it is illustrated in Figure 11.6.

Figure 11.6. A triangle list.

Indexed Triangle Lists

You probably noticed that many of the vertices in the triangle list shown in Figure 11.6 were duplicated, often multiple times. As we'll see in Section 11.1.2.1, we often store quite a lot of metadata with each vertex, so repeating this data in a triangle list wastes memory. It also wastes GPU bandwidth, because a duplicated vertex will be transformed and lit multiple times.

For these reasons, most rendering engines make use of a more efficient data structure known as an *indexed triangle list*. The basic idea is to list the vertices once with no duplication and then to use lightweight vertex *indices* (usually occupying only 16 bits each) to define the triples of vertices that constitute the

图 11.7. 索引三角形列表。

三角形。顶点存储在一个称为顶点缓冲区（DirectX）或顶点数组（OpenGL）的数组中。索引存储在一个称为索引缓冲区或索引数组的单独缓冲区中。该技术如图 11.7 所示。

条状和扇状

游戏渲染有时会使用称为三角形条带和三角形扇形的专用网格数据结构。这两种数据结构都消除了对索引缓冲区的需求，同时仍然在一定程度上减少了顶点重复。它们通过预定义顶点出现的顺序以及它们如何组合形成三角形来实现这一点。

在三角形带中，前三个顶点定义第一个三角形。每个后续顶点与其前两个相邻顶点一起构成一个全新的三角形。为了保持三角形带的缠绕顺序一致，每个新三角形之后，前两个相邻顶点会交换位置。三角形带如图 11.8 所示。

在扇形中，前三个顶点定义第一个三角形，之后的每个顶点由前一个顶点和扇形中的第一个顶点构成一个新的三角形。如图 11.9 所示。

顶点缓存优化

当 GPU 处理索引三角形列表时，每个三角形可以引用顶点缓冲区中的任意顶点。顶点必须按照它们在三角形中出现的顺序进行处理，因为在光栅化阶段必须维护每个三角形的完整性。顶点由顶点着色器处理时，会被缓存以供重复使用。如果后续图元引用了某个顶点

Figure 11.7. An indexed triangle list.

triangles. The vertices are stored in an array known as a *vertex buffer* (DirectX) or *vertex array* (OpenGL). The indices are stored in a separate buffer known as an *index buffer* or *index array*. This technique is shown in Figure 11.7.

Strips and Fans

Specialized mesh data structures known as *triangle strips* and *triangle fans* are sometimes used for game rendering. Both of these data structures eliminate the need for an index buffer, while still reducing vertex duplication to some degree. They accomplish this by predefining the order in which vertices must appear and how they are combined to form triangles.

In a strip, the first three vertices define the first triangle. Each subsequent vertex forms an entirely new triangle, along with its previous two neighbors. To keep the winding order of a triangle strip consistent, the previous two neighbor vertices swap places after each new triangle. A triangle strip is shown in Figure 11.8.

In a fan, the first three vertices define the first triangle and each subsequent vertex defines a new triangle with the previous vertex and the first vertex in the fan. This is illustrated in Figure 11.9.

Vertex Cache Optimization

When a GPU processes an indexed triangle list, each triangle can refer to any vertex within the vertex buffer. The vertices must be processed in the order they appear within the triangles, because the integrity of each triangle must be maintained for the rasterization stage. As vertices are processed by the vertex shader, they are cached for reuse. If a subsequent primitive refers to a vertex

图 11.8 三角形条带。

图 11.9 三角扇。

已经驻留在缓存中，其已处理的属性将被使用，而不是重新处理顶点。

使用条带和扇形结构，一方面是因为它们可以潜在地节省内存（无需索引缓冲区），另一方面是因为它们往往能提高 GPU 对显存的访问的缓存一致性。更妙的是，我们可以使用带索引的条带或带索引的扇形结构来几乎消除顶点重复（这通常比消除索引缓冲区更能节省内存），同时仍然能够获得条带或扇形顶点排序带来的缓存一致性优势。

索引三角形列表也可以进行缓存优化，而无需局限于剥离或扇形顶点排序。顶点缓存优化器是一种离线几何处理工具，它会尝试按优化缓存中顶点重用的顺序列出三角形。它通常会考虑多种因素，例如特定类型 GPU 上顶点缓存的大小，以及 GPU 用于决定何时缓存顶点和何时丢弃顶点的算法。例如，索尼 Edge 几何处理库中包含的顶点缓存优化器可以实现比三角形剥离最高 4% 的渲染吞吐量提升。

11.1.1.4 模型空间

三角形网格顶点的位置向量通常相对于一个方便的局部坐标系（称为模型空间、局部空间或对象空间）来指定。模型空间的原点通常位于对象的中心或

Figure 11.8. A triangle strip.

Figure 11.9. A triangle fan.

that already resides in the cache, its processed attributes are used instead of reprocessing the vertex.

Strips and fans are used in part because they can potentially save memory (no index buffer required) and in part because they tend to improve the cache coherency of the memory accesses made by the GPU to video RAM. Even better, we can use an *indexed strip* or *indexed fan* to virtually eliminate vertex duplication (which can often save more memory than eliminating the index buffer), while still reaping the cache coherency benefits of the strip or fan vertex ordering.

Indexed triangle lists can also be cache-optimized without restricting ourselves to strip or fan vertex ordering. A *vertex cache optimizer* is an offline geometry processing tool that attempts to list the triangles in an order that optimizes vertex reuse within the cache. It generally takes into account factors such as the size of the vertex cache(s) present on a particular type of GPU and the algorithms used by the GPU to decide when to cache vertices and when to discard them. For example, the vertex cache optimizer included in Sony's *Edge* geometry processing library can achieve rendering throughput that is up to 4% better than what is possible with triangle stripping.

11.1.1.4 Model Space

The position vectors of a triangle mesh's vertices are usually specified relative to a convenient local coordinate system called *model space*, *local space*, or *object space*. The origin of model space is usually either in the center of the object or at

图 11.10 模型空间轴的一种可能映射。

其他一些方便的位置，例如角色双脚之间的地板上或车辆车轮水平质心处的地面上。

正如我们在第 5.3.9.1 节中了解到的，模型空间轴的方向是任意的，但这些轴通常与模型上自然的“前”、“左”、“右”和“上”方向对齐。为了稍微严格一点的数学要求，我们可以定义三个单位向量 F 、 L （或 R ）和 U ，并根据需要将它们映射到模型空间中的单位基向量 i 、 j 和 k （从而分别映射到 x 、 y 和 z 轴）。例如，一种常见的映射是 L = i 、 U = j 和 F = k 。映射完全是任意的，但对于整个发动机的所有模型保持一致很重要。图 11.10 显示了飞机模型的模型空间轴的一种可能映射。

11.1.1.5 世界空间和网格实例

许多独立的网格通过在一个称为世界空间的通用坐标系中定位和定向，组成一个完整的场景。任何一个网格都可能在一个场景中出现多次——例如，街道两旁排列着相同的路灯柱，一群面目模糊的士兵，或是一群攻击玩家的蜘蛛。我们将每个这样的对象称为一个网格实例。

网格实例包含对其共享网格数据的引用，以及一个转换矩阵，该矩阵在该特定实例的上下文中将网格的顶点从模型空间转换到世界空间。该矩阵称为模型到世界矩阵，有时也简称为世界矩阵。使用 5.3.10.2 节中的符号，该矩阵可以写成如下形式：

$$\mathbf{M}_{M\to W} = \begin{bmatrix} (\mathbf{RS})_{M\to W} & 0 \\ \mathbf{t}_M & 1 \end{bmatrix},$$

Figure 11.10. One possible mapping of the model-space axes.

some other convenient location, like on the floor between the feet of a character or on the ground at the horizontal centroid of the wheels of a vehicle.

As we learned in Section 5.3.9.1, the sense of the model-space axes is arbitrary, but the axes typically align with the natural “front,” “left,” “right” and “up” directions on the model. For a little mathematical rigor, we can define three unit vectors **F**, **L** (or **R**) and **U** and map them as desired onto the unit basis vectors **i**, **j** and **k** (and hence to the *x*-, *y*- and *z*-axes, respectively) in model space. For example, a common mapping is $\mathbf{L} = \mathbf{i}$, $\mathbf{U} = \mathbf{j}$ and $\mathbf{F} = \mathbf{k}$. The mapping is completely arbitrary, but it’s important to be consistent for all models across the entire engine. Figure 11.10 shows one possible mapping of the model-space axes for an aircraft model.

11.1.1.5 World Space and Mesh Instancing

Many individual meshes are composed into a complete scene by positioning and orienting them within a common coordinate system known as *world space*. Any one mesh might appear many times in a scene—examples include a street lined with identical lamp posts, a faceless mob of soldiers or a swarm of spiders attacking the player. We call each such object a *mesh instance*.

A mesh instance contains a reference to its shared mesh data and also includes a transformation matrix that converts the mesh’s vertices from model space to world space, within the context of that particular instance. This matrix is called the *model-to-world* matrix, or sometimes just the *world matrix*. Using the notation from Section 5.3.10.2, this matrix can be written as follows:

$$\mathbf{M}_{M\to W} = \begin{bmatrix} (\mathbf{RS})_{M\to W} & 0 \\ \mathbf{t}_M & 1 \end{bmatrix},$$

其中，上层 3 × 3 矩阵（RS）M → W 将模型空间顶点旋转并缩放到世界空间，t M 是在世界空间中表示的模型空间轴的平移。如果我们有单位模型空间基向量 i M 、 j M 和 k M ，以世界空间坐标系表示，则该矩阵也可以写成如下形式：

$$\mathbf{M}_{M\to W} = \left[\begin{array}{c|c} \mathbf{i}_M & 0 \\ \mathbf{j}_M & 0 \\ \mathbf{k}_M & 0 \\ \hline \mathbf{t}_M & 1 \end{array}\right].$$

给定一个以模型空间坐标表示的顶点，渲染引擎按如下方式计算其世界空间等效值：

$$\mathbf{v}_W = \mathbf{v}_M \mathbf{M}_{M\to W}.$$

我们可以将矩阵 MM → W 视为模型空间轴本身的位置和方向的描述，以世界空间坐标系表示。或者，我们可以将其视为将顶点从模型空间转换到世界空间的矩阵。

渲染网格时，模型到世界矩阵也会应用于网格的表面法线（参见第 11.1.2.1 节）。回想一下第 5.3.11 节，为了正确地变换法线向量，我们必须将其乘以模型到世界矩阵的逆转置。如果我们的矩阵不包含任何缩放或剪切，我们可以通过在与模型到世界矩阵相乘之前将其 w 分量设置为零来正确地变换法线向量，如第 5.3.6.1 节所述。

有些网格，例如建筑物、地形和其他背景元素，是完全静态且唯一的。这些网格的顶点通常在世界空间中表示，因此它们的模型到世界矩阵是恒等的，可以忽略不计。

11.1.2 描述表面的视觉属性

为了正确地渲染和照亮表面，我们需要描述其视觉属性。表面属性包括几何信息，例如表面各个点的表面法线方向。它们还包含光线如何与表面相互作用的描述。这包括漫反射颜色、光泽度/反射率、粗糙度或纹理、不透明度或透明度、折射率和其他光学属性。表面属性还可能包括表面如何随时间变化的规范（例如，动画角色的皮肤如何跟踪其骨骼的关节，或者水体表面如何运动）。

where the upper 3×3 matrix $(\mathbf{RS})_{M\to W}$ rotates and scales model-space vertices into world space, and $\mathbf{t}_M$ is the translation of the model-space axes expressed in world space. If we have the unit model-space basis vectors $\mathbf{i}_M$, $\mathbf{j}_M$ and $\mathbf{k}_M$, expressed in world-space coordinates, this matrix can also be written as follows:

$$\mathbf{M}_{M\to W} = \left[\begin{array}{c|c} \mathbf{i}_M & 0 \\ \mathbf{j}_M & 0 \\ \mathbf{k}_M & 0 \\ \hline \mathbf{t}_M & 1 \end{array}\right].$$

Given a vertex expressed in model-space coordinates, the rendering engine calculates its world-space equivalent as follows:

$$\mathbf{v}_W = \mathbf{v}_M \mathbf{M}_{M\to W}.$$

We can think of the matrix $\mathbf{M}_{M\to W}$ as a description of the position and orientation of the model-space axes themselves, expressed in world-space coordinates. Or we can think of it as a matrix that transforms vertices from model space to world space.

When rendering a mesh, the model-to-world matrix is also applied to the surface normals of the mesh (see Section 11.1.2.1). Recall from Section 5.3.11, that in order to transform normal vectors properly, we must multiply them by the inverse transpose of the model-to-world matrix. If our matrix does not contain any scale or shear, we can transform our normal vectors correctly by simply setting their w components to zero prior to multiplication by the model-to-world matrix, as described in Section 5.3.6.1.

Some meshes like buildings, terrain and other background elements are entirely static and unique. The vertices of these meshes are often expressed in world space, so their model-to-world matrices are identity and can be ignored.

11.1.2 Describing the Visual Properties of a Surface

In order to properly render and light a surface, we need a description of its *visual properties*. Surface properties include geometric information, such as the direction of the surface normal at various points on the surface. They also encompass a description of how light should interact with the surface. This includes diffuse color, shininess/reflectivity, roughness or texture, degree of opacity or transparency, index of refraction and other optical properties. Surface properties might also include a specification of how the surface should change over time (e.g., how an animated character's skin should track the joints of its skeleton or how the surface of a body of water should move).

渲染逼真图像的关键在于正确考虑光线与场景中物体相互作用时的行为。因此，渲染工程师需要充分理解光线的工作原理、光线如何在环境中传播，以及虚拟相机如何“感知”光线并将其转化为屏幕像素中存储的色彩。

11.1.2.1 光与色彩简介

光是电磁辐射；在不同情况下，它既像波，又像粒子。光的颜色由其强度 I 和波长 λ（或频率 f，其中 f = 1/λ）决定。可见光色域范围从波长 740 nm（或频率 430 THz）到波长 380 nm（750 THz）。一束光可能包含单一纯波长（即彩虹的颜色，也称为光谱色），也可能包含各种波长的混合。我们可以绘制一个图表来显示给定光束包含多少每个频率，称为光谱图。白光包含所有波长的一点点，因此它的光谱图看起来大致像一个延伸到整个可见波段的盒子。纯绿光只包含一个波长，因此它的光谱图看起来像一个大约 570 THz 的无限窄尖峰。

光与物体的相互作用

光与物质之间可以发生许多复杂的相互作用。其行为一方面取决于其传播的介质，另一方面取决于不同介质（气固界面、气-水界面、水-玻璃界面等）之间界面的形状和性质。从技术角度来说，表面实际上只是两种不同介质之间的界面。

尽管光非常复杂，但它实际上只能做四件事：

- 它可以被吸收。
- 可以反映出来。
- 它可以通过物体传输，通常在此过程中发生折射（弯曲）。
- 当穿过非常狭窄的开口时，它可能会发生衍射。

大多数真实感渲染引擎都会考虑前三种行为；通常不会考虑衍射，因为在大多数场景中很少会注意到它的影响。

只有某些波长会被表面吸收，而其他波长会被反射。这就是我们对物体颜色感知的原因。例如，当白光照射到红色物体上时，除了

The key to rendering photorealistic images is properly accounting for light's behavior as it interacts with the objects in the scene. Hence rendering engineers need to have a good understanding of how light works, how it is transported through an environment and how the virtual camera "senses" it and translates it into the colors stored in the pixels on-screen.

11.1.2.1 Introduction to Light and Color

Light is electromagnetic radiation; it acts like both a wave and a particle in different situations. The color of light is determined by its *intensity* I and its *wavelength* λ (or its frequency f, where $f = 1/\lambda$). The visible gamut ranges from a wavelength of 740 nm (or a frequency of 430 THz) to a wavelength of 380 nm (750 THz). A beam of light may contain a single pure wavelength (i.e., the colors of the rainbow, also known as the *spectral colors*), or it may contain a mixture of various wavelengths. We can draw a graph showing how much of each frequency a given beam of light contains, called a *spectral plot*. White light contains a little bit of all wavelengths, so its spectral plot would look roughly like a box extending across the entire visible band. Pure green light contains only one wavelength, so its spectral plot would look like a single infinitesimally narrow spike at about 570 THz.

Light-Object Interactions

Light can have many complex interactions with matter. Its behavior is governed in part by the *medium* through which it is traveling and in part by the shape and properties of the *interfaces* between different types of media (air-solid, air-water, water-glass, etc.). Technically speaking, a surface is really just an interface between two different types of media.

Despite all of its complexity, light can really only do four things:

- It can be *absorbed*.
- It can be *reflected*.
- It can be *transmitted* through an object, usually being *refracted* (bent) in the process.
- It can be *diffracted* when passing through very narrow openings.

Most photorealistic rendering engines account for the first three of these behaviors; diffraction is not usually taken into account because its effects are rarely noticeable in most scenes.

Only certain wavelengths may be absorbed by a surface, while others are reflected. This is what gives rise to our perception of the color of an object. For example, when white light falls on a red object, all wavelengths except

红色光被吸收，因此物体呈现红色。当红光投射到白色物体上时，也会产生同样的感知效果——我们的眼睛无法感知其中的区别。

反射可以是漫反射，即入射光线均匀地散射到各个方向。反射也可以是镜面反射，即入射光线会直接反射或仅扩散到一个狭窄的锥体中。反射也可以是各向异性的，即光线从表面反射的方式会根据观察表面的角度而变化。

当光线穿过某个物体时，会发生散射（例如半透明物体）、部分吸收（例如彩色玻璃）或折射（例如光线穿过棱镜时）。不同波长的光的折射角度可能不同，从而导致光谱扩散。这就是为什么当光线穿过雨滴和玻璃棱镜时，我们会看到彩虹。光线也可以进入半固体表面，四处反射，然后从与进入表面不同的位置离开表面。我们称之为次表面散射，正是这种效应赋予了皮肤、蜡和大理石特有的温暖外观。

色彩空间和色彩模型

颜色模型是一种测量颜色的三维坐标系。颜色空间是一种特定的标准，用于规定特定颜色模型中的数值颜色应如何映射到人类在现实世界中感知的颜色。由于我们眼睛中有三种颜色传感器（视锥细胞），它们对不同波长的光敏感，因此颜色模型通常是三维的。

计算机图形学中最常用的颜色模型是 RGB 模型。在该模型中，颜色空间用一个单位立方体表示，红、绿、蓝光的相对强度沿立方体的轴测量。红、绿、蓝分量称为颜色通道。在标准的 RGB 颜色模型中，每个通道的值范围从 0 到 1。因此，颜色 $(0, 0, 0)$ 表示黑色，而 $(1, 1, 1)$ 表示白色。

当颜色存储在位图图像中时，可以采用各种颜色格式。颜色格式部分由其占用的每个像素的位数定义，更具体地说，由用于表示每个颜色通道的位数定义。RGB888 格式每个通道使用 8 位，每个像素总共 24 位。在此格式中，每个通道的范围从 0 到 255，而不是从 0 到 1。RGB565 使用 5 位表示红色和蓝色，6 位表示绿色，每个像素总共 16 位。调色板格式可能使用每个像素 8 位来存储 256 个元素调色板的索引，其中每个条目可能以 RGB888 或其他合适的格式存储。

red are absorbed, hence the object appears red. The same perceptual effect is achieved when red light is cast onto a white object—our eyes don't know the difference.

Reflections can be *diffuse*, meaning that an incoming ray is scattered equally in all directions. Reflections can also be *specular*, meaning that an incident light ray will reflect directly or be spread only into a narrow cone. Reflections can also be *anisotropic*, meaning that the way in which light reflects from a surface changes depending on the angle at which the surface is viewed.

When light is transmitted through a volume, it can be *scattered* (as is the case for translucent objects), partially *absorbed* (as with colored glass), or *refracted* (as happens when light travels through a prism). The refraction angles can be different for different wavelengths, leading to spectral spreading. This is why we see rainbows when light passes through raindrops and glass prisms. Light can also enter a semi-solid surface, bounce around and then exit the surface at a different point from the one at which it entered the surface. We call this *subsurface scattering*, and it is one of the effects that gives skin, wax and marble their characteristic warm appearance.

Color Spaces and Color Models

A *color model* is a three-dimensional coordinate system that measures colors. A *color space* is a specific standard for how numerical colors in a particular color model should be mapped onto the colors perceived by human beings in the real world. Color models are typically three-dimensional because of the three types of color sensors (cones) in our eyes, which are sensitive to different wavelengths of light.

The most commonly used color model in computer graphics is the RGB model. In this model, color space is represented by a unit cube, with the relative intensities of red, green and blue light measured along its axes. The red, green and blue components are called *color channels*. In the canonical RGB color model, each channel ranges from zero to one. So the color $(0, 0, 0)$ represents black, while $(1, 1, 1)$ represents white.

When colors are stored in a bitmapped image, various color formats can be employed. A color format is defined in part by the number of *bits per pixel* it occupies and, more specifically, the number of bits used to represent each color channel. The RGB888 format uses eight bits per channel, for a total of 24 bits per pixel. In this format, each channel ranges from 0 to 255 rather than from zero to one. RGB565 uses five bits for red and blue and six for green, for a total of 16 bits per pixel. A paletted format might use eight bits per pixel to store indices into a 256-element color palette, each entry of which might be stored in RGB888 or some other suitable format.

3D渲染中还使用了许多其他颜色模型。我们将在11.3.1.5节中看到log-LUV颜色模型如何用于高动态范围（HDR）照明。

不透明度和 Alpha 通道

第四个通道称为 alpha，通常会附加到 RGB 颜色向量上。如第 11.1.1 节所述，alpha 测量对象的不透明度。当 alpha 存储在图像像素中时，它表示该像素的不透明度。

RGB 颜色格式可以扩展为包含 Alpha 通道，在这种情况下，它们被称为 RGB A 或 ARGB 颜色格式。例如，RGBA8888 是一种 32 位/像素格式，其中红色、绿色、蓝色和 Alpha 通道各占 8 位。RGBA5551 是一种 16 位/像素格式，包含 1 位 Alpha 通道；在这种格式下，颜色可以完全不透明，也可以完全透明。

11.1.2.2 顶点属性

描述表面视觉属性最简单的方法是将其指定在表面上的离散点上。网格的顶点是存储表面属性的便捷位置，在这种情况下，它们被称为顶点属性。

典型的三角形网格在每个顶点上包含以下部分或全部属性。作为渲染工程师，我们当然可以自由定义任何可能需要的附加属性，以便在屏幕上实现所需的视觉效果。

- 位置矢量（pi = $p_{ix} \quad p_{iy} \quad p_{iz}$]）。这是第 i 个网格中的顶点。它通常在对象局部的坐标空间中指定，称为模型空间。]
- 法线顶点（is = [n ix n iy n out

  顶点 i 位置处的面法线。它用于每个顶点的动态光照计算。[t ix

  t iy t iz]）和双切线（bi = [b ix b iy b iz]）。
- 顶点切线（ti =

  这两个单位向量彼此垂直，并且与顶点法线 ni 垂直。这三个向量 ni 、 ti 和 bi 共同定义了一组坐标轴，称为切线空间。该空间用于各种逐像素光照计算，例如法线贴图和环境贴图。（双切线 bi 有时会被混淆地称为双法线，即使它并不垂直于表面。）

- 漫反射颜色（di = [d iR d iG d iB d iA] ）。这个四元素向量描述了表面的漫反射颜色，以 RGB 颜色空间表示。它通常还包括不透明度或 alpha (A) 的规范。

A number of other color models are also used in 3D rendering. We'll see how the log-LUV color model is used for *high dynamic range* (HDR) lighting in Section 11.3.1.5.

Opacity and the Alpha Channel

A fourth channel called *alpha* is often tacked on to RGB color vectors. As mentioned in Section 11.1.1, alpha measures the opacity of an object. When stored in an image pixel, alpha represents the opacity of the pixel.

RGB color formats can be extended to include an alpha channel, in which case they are referred to as RGBA or ARGB color formats. For example, RGBA8888 is a 32 bit-per-pixel format with eight bits each for red, green, blue and alpha. RGBA5551 is a 16 bit-per-pixel format with one-bit alpha; in this format, colors can either be fully opaque or fully transparent.

11.1.2.2 Vertex Attributes

The simplest way to describe the visual properties of a surface is to specify them at discrete points on the surface. The vertices of a mesh are a convenient place to store surface properties, in which case they are called *vertex attributes*.

A typical triangle mesh includes some or all of the following attributes at each vertex. As rendering engineers, we are of course free to define any additional attributes that may be required in order to achieve a desired visual effect on-screen.

- *Position vector* ($\mathbf{p}_i = \begin{bmatrix} p_{ix} & p_{iy} & p_{iz} \end{bmatrix}$). This is the 3D position of the *i*th vertex in the mesh. It is usually specified in a coordinate space local to the object, known as *model space*.
- *Vertex normal* ($\mathbf{n}_i = \begin{bmatrix} n_{ix} & n_{iy} & n_{iz} \end{bmatrix}$). This vector defines the unit surface normal at the position of vertex *i*. It is used in per-vertex dynamic lighting calculations.
- *Vertex tangent* ($\mathbf{t}_i = \begin{bmatrix} t_{ix} & t_{iy} & t_{iz} \end{bmatrix}$) and *bitangent* ($\mathbf{b}_i = \begin{bmatrix} b_{ix} & b_{iy} & b_{iz} \end{bmatrix}$). These two unit vectors lie perpendicular to one another and to the vertex normal $\mathbf{n}_i$. Together, the three vectors $\mathbf{n}_i$, $\mathbf{t}_i$ and $\mathbf{b}_i$ define a set of coordinate axes known as *tangent space*. This space is used for various per-pixel lighting calculations, such as normal mapping and environment mapping. (The bitangent $\mathbf{b}_i$ is sometimes confusingly called the *binormal*, even though it is *not* normal to the surface.)
- *Diffuse color* ($\mathbf{d}_i = \begin{bmatrix} d_{iR} & d_{iG} & d_{iB} & d_{iA} \end{bmatrix}$). This four-element vector describes the diffuse color of the surface, expressed in the RGB color space. It typically also includes a specification of the opacity or *alpha* (A) of the

顶点位置的表面。此颜色可以离线计算（静态照明），也可以在运行时

• 镜面反射颜色（si = [s iR s iG s iB s iA]）。该量描述了当光从闪亮表面直接反射到虚拟相机的成像平面时应该出现的镜面反射高光的颜色。

• 纹理坐标（uij = [uijvij]）。纹理坐标允许将二维（有时是三维）位图“收缩”到网格表面，此过程称为纹理映射。纹理坐标（u，v）描述特定顶点在纹理的二维标准化坐标空间中的位置。一个三角形可以映射到多个纹理上；因此它可以有多组纹理坐标。我们用上面的下标 j 表示不同的纹理坐标集。

• 蒙皮权重（k ij = [k ij w ij]）。在骨骼动画中，网格的顶点连接到铰接骨架中的各个关节。在这种情况下，每个顶点必须通过索引 k 指定它连接到哪个关节。一个顶点可以受多个关节的影响，在这种情况下，最终顶点位置成为这些影响的加权平均值。因此，每个关节影响的权重用加权因子 w 表示。通常，顶点 i 可以有多个关节影响 j，每个用数字对（k ij，w ij）表示。

11.1.2.3 顶点格式

顶点属性通常存储在诸如 C 结构体或 C++ 类之类的数据结构中。这种数据结构的布局称为顶点格式。不同的网格需要不同的属性组合，因此需要不同的顶点格式。以下是一些常见顶点格式的示例：

```
// Simplest possible vertex -- position only (useful for
// shadow volume extrusion, silhouette edge detection
// for cartoon rendering, z-prepass, etc.)
struct Vertex1P
{
    Vector3  m_p;  // position
};

// A typical vertex format with position, vertex normal
// and one set of texture coordinates.
struct Vertex1P1N1UV
{
    Vector3 m_p;     // position
```


surface at the position of the vertex. This color may be calculated offline (static lighting) or at runtime (dynamic lighting).

- *Specular color* ($\mathbf{s}_i = \begin{bmatrix} s_{iR} & s_{iG} & s_{iB} & s_{iA} \end{bmatrix}$). This quantity describes the color of the specular highlight that should appear when light reflects directly from a shiny surface onto the virtual camera's imaging plane.
- *Texture coordinates* ($\mathbf{u}_{ij} = \begin{bmatrix} u_{ij} & v_{ij} \end{bmatrix}$). Texture coordinates allow a two- (or sometimes three-) dimensional bitmap to be "shrink wrapped" onto the surface of a mesh—a process known as *texture mapping*. A texture coordinate (u, v) describes the location of a particular vertex within the two-dimensional normalized coordinate space of the texture. A triangle can be mapped with more than one texture; hence it can have more than one set of texture coordinates. We've denoted the distinct sets of texture coordinates via the subscript j above.
- *Skinning weights* ($\mathbf{k}_{ij} = \begin{bmatrix} k_{ij} & w_{ij} \end{bmatrix}$). In skeletal animation, the vertices of a mesh are attached to individual joints in an articulated skeleton. In this case, each vertex must specify to which joint it is attached via an index, k. A vertex can be influenced by multiple joints, in which case the final vertex position becomes a *weighted average* of these influences. Thus, the weight of each joint's influence is denoted by a weighting factor w. In general, a vertex i can have multiple joint influences j, each denoted by the pair of numbers (k_{ij}, w_{ij}).

II.1.2.3 Vertex Formats

Vertex attributes are typically stored within a data structure such as a C `struct` or a C++ class. The layout of such a data structure is known as a *vertex format*. Different meshes require different combinations of attributes and hence need different vertex formats. The following are some examples of common vertex formats:

```
// Simplest possible vertex -- position only (useful for
// shadow volume extrusion, silhouette edge detection
// for cartoon rendering, z-prepass, etc.)
struct Vertex1P
{
    Vector3  m_p;  // position
};

// A typical vertex format with position, vertex normal
// and one set of texture coordinates.
struct Vertex1P1N1UV
{
    Vector3 m_p;     // position
```

```
    Vector3 m_n;     // vertex normal
    F32     m_uv[2]; // (u, v) texture coordinate
};

// A skinned vertex with position, diffuse and specular
// colors and four weighted joint influences.
struct Vertex1P1D1S2UV4J
{
    Vector3 m_p;      // position
    Color4  m_d;      // diffuse color and translucency
    Color4  m_S;      // specular color
    F32     m_uv0[2]; // first set of tex coords
    F32     m_uv1[2]; // second set of tex coords
    U8      m_k[4];   // four joint indices, and...
    F32     m_w[3];   // three joint weights, for
                      // skinning (fourth is calc'd
                      // from the first three)
};
```

显然，顶点属性的可能排列组合数量——以及由此产生的不同顶点格式的数量——可能会变得非常庞大。（事实上，如果允许任意数量的纹理坐标和/或关节权重，那么格式的数量理论上是无限的。）管理所有这些顶点格式是任何图形程序员都经常遇到的难题。

可以采取一些步骤来减少引擎必须支持的顶点格式的数量。在实际的图形应用中，许多理论上可行的顶点格式根本没有用，或者图形硬件或游戏的着色器无法处理它们。一些游戏团队还会将自己限制在有用/可行的顶点格式的子集上，以使事情更易于管理。例如，他们可能只允许每个顶点使用零个、两个或四个关节权重，或者他们可能决定每个顶点支持不超过两组纹理坐标。一些 GPU 能够从顶点数据结构中提取属性子集，因此游戏团队还可以选择对所有网格使用单个“überformat”，并让硬件根据着色器的要求选择相关属性。

11.1.2.4 属性插值

三角形顶点的属性只是对整个表面视觉属性的粗略离散近似。渲染三角形时，真正重要的是屏幕上每个像素“看到”的三角形内部点的视觉属性。换句话说，我们需要知道每个像素的属性值，而不是每个顶点的属性值。


```
    Vector3 m_n;     // vertex normal
    F32     m_uv[2]; // (u, v) texture coordinate
};

// A skinned vertex with position, diffuse and specular
// colors and four weighted joint influences.
struct Vertex1P1D1S2UV4J
{
    Vector3 m_p;      // position
    Color4  m_d;      // diffuse color and translucency
    Color4  m_S;      // specular color
    F32     m_uv0[2]; // first set of tex coords
    F32     m_uv1[2]; // second set of tex coords
    U8      m_k[4];   // four joint indices, and...
    F32     m_w[3];   // three joint weights, for
                      // skinning (fourth is calc'd
                      // from the first three)
};
```

Clearly the number of possible permutations of vertex attributes—and hence the number of distinct vertex formats—can grow to be extremely large. (In fact the number of formats is theoretically unbounded, if one were to permit any number of texture coordinates and/or joint weights.) Management of all these vertex formats is a common source of headaches for any graphics programmer.

Some steps can be taken to reduce the number of vertex formats that an engine has to support. In practical graphics applications, many of the theoretically possible vertex formats are simply not useful, or they cannot be handled by the graphics hardware or the game's shaders. Some game teams also limit themselves to a subset of the useful/feasible vertex formats in order to keep things more manageable. For example, they might only allow zero, two or four joint weights per vertex, or they might decide to support no more than two sets of texture coordinates per vertex. Some GPUs are capable of extracting a subset of attributes from a vertex data structure, so game teams can also choose to use a single "überformat" for all meshes and let the hardware select the relevant attributes based on the requirements of the shader.

11.1.2.4 Attribute Interpolation

The attributes at a triangle's vertices are just a coarse, discretized approximation to the visual properties of the surface as a whole. When rendering a triangle, what really matters are the visual properties at the interior points of the triangle as "seen" through each pixel on-screen. In other words, we need to know the values of the attributes on a *per-pixel* basis, not a per-vertex basis.

图 11.11. 顶点具有不同灰度的 Gouraud 阴影三角形。

图 11.12。Gouraud 着色可以使多面物体看起来更光滑。

确定网格表面属性逐像素值的一种简单方法是对每个顶点的属性数据进行线性插值。当应用于顶点颜色时，属性插值被称为 Gouraud 着色。图 11.11 显示了应用于三角形的 Gouraud 着色示例，图 11.12 展示了其在简单三角形网格上的效果。插值通常也应用于其他类型的顶点属性信息，例如顶点法线、纹理坐标和深度。

顶点法线和平滑

正如我们将在 11.1.3 节中看到的，光照是根据物体表面的视觉属性和照射到物体上的光线属性，计算物体表面各个点颜色的过程。最简单的网格光照方法是逐个顶点计算表面颜色。换句话说，我们利用表面属性和入射光来计算每个顶点的漫反射颜色 (di)。然后，这些顶点颜色通过高氏着色法在网格的三角形之间进行插值。

为了确定光线如何从表面的某个点反射，大多数光照模型都会使用光线撞击点处表面的法线向量。由于我们是基于每个顶点进行光照计算的，因此可以使用顶点法线 ni 来实现此目的。因此，网格顶点法线的方向会对网格的最终外观产生重大影响。

Figure 11.11. A Gouraud-shaded triangle with different shades of gray at the vertices.

Figure 11.12. Gouraud shading can make faceted objects appear to be smooth.

One simple way to determine the per-pixel values of a mesh's surface attributes is to *linearly interpolate* the per-vertex attribute data. When applied to vertex colors, attribute interpolation is known as *Gouraud shading*. An example of Gouraud shading applied to a triangle is shown in Figure 11.11, and its effects on a simple triangle mesh are illustrated in Figure 11.12. Interpolation is routinely applied to other kinds of vertex attribute information as well, such as vertex normals, texture coordinates and depth.

Vertex Normals and Smoothing

As we'll see in Section 11.1.3, *lighting* is the process of calculating the color of an object at various points on its surface, based on the visual properties of the surface and the properties of the light impinging upon it. The simplest way to light a mesh is to calculate the color of the surface on a *per-vertex* basis. In other words, we use the properties of the surface and the incoming light to calculate the diffuse color of each vertex ($\mathbf{d}_i$). These vertex colors are then interpolated across the triangles of the mesh via Gouraud shading.

In order to determine how a ray of light will reflect from a point on a surface, most lighting models make use of a vector that is *normal* to the surface at the point of the light ray's impact. Since we're performing lighting calculations on a per-vertex basis, we can use the vertex normal $\mathbf{n}_i$ for this purpose. Therefore, the directions of a mesh's vertex normals can have a significant impact on the final appearance of a mesh.

图 11.13。网格顶点法线的方向会对每个顶点照明计算期间计算的颜色产生重大影响。

举个例子，假设有一个又高又细的四边形盒子。如果我们想让盒子看起来边缘清晰，可以指定顶点法线垂直于盒子的面。当我们照亮每个三角形时，我们会在三个顶点遇到相同的法向量，因此最终的光照看起来是平坦的，并且在盒子的角点处会像顶点法线一样发生突然变化。

我们还可以通过指定从盒子中心线向外径向延伸的顶点法线，使同一个盒子网格看起来有点像光滑的圆柱体。在这种情况下，每个三角形的顶点将具有不同的顶点法线，这导致我们需要在每个顶点计算不同的颜色。Gouraud 着色会平滑地插值这些顶点颜色，从而使光照在整个表面上看起来平滑变化。此效果如图 11.13 所示。

11.1.2.5 纹理

当三角形相对较大时，基于顶点指定表面属性可能过于粗粒度。线性属性插值并不总是我们想要的，而且它可能会导致不良的视觉异常。

举个例子，考虑一下渲染光线照射在光滑物体上时产生的明亮镜面高光的问题。如果网格是高度细分的，逐顶点光照结合高氏着色可以产生相当不错的效果。然而，当三角形过大时，镜面高光的线性插值误差会变得非常明显，如图 11.14 所示。

Figure 11.13. The directions of a mesh's vertex normals can have a profound effect on the colors calculated during per-vertex lighting calculations.

As an example, consider a tall, thin, four-sided box. If we want the box to appear to be sharp-edged, we can specify the vertex normals to be perpendicular to the faces of the box. As we light each triangle, we will encounter the same normal vector at all three vertices, so the resulting lighting will appear flat, and it will abruptly change at the corners of the box just as the vertex normals do.

We can also make the same box mesh look a bit like a smooth cylinder by specifying vertex normals that point radially outward from the box's center line. In this case, the vertices of each triangle will have different vertex normals, causing us to calculate different colors at each vertex. Gouraud shading will smoothly interpolate these vertex colors, resulting in lighting that appears to vary smoothly across the surface. This effect is illustrated in Figure 11.13.

11.1.2.5 Textures

When triangles are relatively large, specifying surface properties on a per-vertex basis can be too coarse-grained. Linear attribute interpolation isn't always what we want, and it can lead to undesirable visual anomalies.

As an example, consider the problem of rendering the bright *specular highlight* that can occur when light shines on a glossy object. If the mesh is highly tessellated, per-vertex lighting combined with Gouraud shading can yield reasonably good results. However, when the triangles are too large, the errors that arise from linearly interpolating the specular highlight can become jarringly obvious, as shown in Figure 11.14.

图 11.14。顶点属性的线性插值并不总能充分描述表面的视觉特性，尤其是在细分程度较低的情况下。

为了克服每个顶点表面属性的限制，渲染工程师使用称为纹理贴图的位图图像。纹理通常包含颜色信息，通常投影到网格的三角形上。在这种情况下，它的作用有点像我们小时候贴在手臂上的那些傻乎乎的假纹身。但纹理除了颜色之外，还可以包含其他类型的视觉表面属性。而且纹理不必投影到网格上——例如，纹理可以用作独立的数据表。纹理的各个图像元素称为纹素，以将它们与屏幕上的像素区分开来。

在某些图形硬件上，纹理位图的尺寸被限制为 2 的幂次方。典型的纹理尺寸包括 256 × 256 、 512 × 512 、 1024 × 1024 和 2048 × 2048 ，尽管在大多数硬件上，只要纹理能够装入显存，纹理可以是任意大小。某些图形硬件会施加额外的限制，例如要求纹理为正方形，或者放宽一些限制，例如不将纹理尺寸限制为 2 的幂次方。

纹理类型

最常见的纹理类型是漫反射贴图（或反照率贴图）。它描述了表面每个纹理像素的漫反射颜色，就像表面上的贴花或油漆一样。

计算机图形学中也使用其他类型的纹理，包括法线贴图（存储每个纹素的单位法线向量，并以 RGB 值编码）、光泽贴图（编码表面在每个纹素上的光泽度）、环境贴图（包含用于渲染反射的周围环境图像）等等。有关如何使用各种类型的纹理来实现基于图像的光照和其他效果的讨论，请参见第 11.3.1 节。

Figure 11.14. Linear interpolation of vertex attributes does not always yield an adequate description of the visual properties of a surface, especially when tessellation is low.

To overcome the limitations of per-vertex surface attributes, rendering engineers use bitmapped images known as *texture maps*. A texture often contains color information and is usually projected onto the triangles of a mesh. In this case, it acts a bit like those silly fake tattoos we used to apply to our arms when we were kids. But a texture can contain other kinds of visual surface properties as well as colors. And a texture needn't be projected onto a mesh—for example, a texture might be used as a stand-alone data table. The individual picture elements of a texture are called *texels* to differentiate them from the pixels on the screen.

The dimensions of a texture bitmap are constrained to be powers of two on some graphics hardware. Typical texture dimensions include 256 × 256, 512 × 512, 1024 × 1024 and 2048 × 2048, although textures can be any size on most hardware, provided the texture fits into video memory. Some graphics hardware imposes additional restrictions, such as requiring textures to be square, or lifts some restrictions, such as not constraining texture dimensions to be powers of two.

Types of Textures

The most common type of texture is known as a *diffuse map*, or *albedo map*. It describes the diffuse surface color at each texel on a surface and acts like a decal or paint job on the surface.

Other types of textures are used in computer graphics as well, including *normal maps* (which store unit normal vectors at each texel, encoded as RGB values), *gloss maps* (which encode how shiny a surface should be at each texel), environment maps (which contain a picture of the surrounding environment for rendering reflections) and many others. See Section 11.3.1 for a discussion of how various types of textures can be used for image-based lighting and other effects.

实际上，我们可以使用纹理贴图来存储光照计算中所需的任何信息。例如，一维纹理可以用来存储复杂数学函数的采样值、颜色到颜色的映射表，或任何其他类型的查找表 (LUT)。

纹理坐标

让我们考虑如何将二维纹理投影到网格上。为此，我们定义一个二维坐标系，称为纹理空间。纹理坐标通常用一对标准化的数字 (u , v) 表示。这些坐标的范围始终是从纹理左下角的 (0, 0) 到右上角的 (1, 1)。使用这样的标准化坐标，无论纹理的尺寸如何，都可以使用相同的坐标系。

要将三角形映射到二维纹理上，我们只需在每个顶点 i 处指定一对纹理坐标 (u i, vi)。这样就能有效地将三角形映射到纹理空间的图像平面上。图 11.15 展示了一个纹理映射的示例。

图 11.15 纹理映射示例。三角形在三维空间和纹理空间中均有显示。

纹理寻址模式

纹理坐标允许超出 [0, 1] 范围。图形硬件可以通过以下任一方式处理超出范围的纹理坐标。这些方式称为纹理寻址模式；具体使用哪种模式由用户控制。

- 包裹 。在此模式下，纹理会在各个方向上反复重复。所有形式为 (ju , kv) 的纹理坐标均等同于坐标 (u , v)，其中 j 和 k 为任意整数。

We can actually use texture maps to store any information that we happen to need in our lighting calculations. For example, a one-dimensional texture could be used to store sampled values of a complex math function, a color-to-color mapping table, or any other kind of look-up table (LUT).

Texture Coordinates

Let's consider how to project a two-dimensional texture onto a mesh. To do this, we define a two-dimensional coordinate system known as *texture space*. A texture coordinate is usually represented by a normalized pair of numbers denoted (u, v). These coordinates always range from $(0, 0)$ at the bottom left corner of the texture to $(1, 1)$ at the top right. Using normalized coordinates like this allows the same coordinate system to be used regardless of the dimensions of the texture.

To map a triangle onto a 2D texture, we simply specify a pair of texture coordinates (u_i, v_i) at each vertex i. This effectively maps the triangle onto the image plane in texture space. An example of texture mapping is depicted in Figure 11.15.

Figure 11.15. An example of texture mapping. The triangles are shown both in three-dimensional space and in texture space.

Texture Addressing Modes

Texture coordinates are permitted to extend beyond the $[0, 1]$ range. The graphics hardware can handle out-of-range texture coordinates in any one of the following ways. These are known as *texture addressing modes*; which mode is used is under the control of the user.

- *Wrap*. In this mode, the texture is repeated over and over in every direction. All texture coordinates of the form (ju, kv) are equivalent to the coordinate (u, v), where j and k are arbitrary integers.

图 11.16. 纹理寻址模式。

• 镜像 。此模式的作用类似于包裹模式，不同之处在于，对于 u 的奇数倍数，纹理会绕 v 轴镜像；对于 v 的奇数倍数，纹理会绕 u 轴镜像。

• 钳制。在此模式下，当纹理坐标超出正常范围时，纹理外边缘周围的纹素颜色会被简单地扩展。

• 边框颜色。在此模式下，对于 [0, 1] 纹理坐标范围之外的区域，使用任意用户指定的颜色。

这些纹理寻址模式如图 11.16 所示。

纹理格式

只要你的游戏引擎包含将其读入内存所需的代码，纹理位图几乎可以以任何图像格式存储在磁盘上。常见的格式包括 Targa (.tga)、便携式网络图形 (.png)、Windows 位图 (.bmp) 和标记图像文件格式 (.tif)。在内存中，纹理通常表示为二维（步长）像素数组，使用各种颜色格式，包括 RGB888、RGBA8888、RGB565、RGBA5551 等等。

大多数现代显卡和图形 API 都支持压缩纹理。DirectX 支持一系列压缩格式，称为 DXT 或 S3 纹理压缩 (S3TC)。我们不会在这里详细介绍，但其基本思想是将纹理分解成 4 × 4 像素块，并使用较小的

Figure II.16. Texture addressing modes.

- *Mirror*. This mode acts like wrap mode, except that the texture is mirrored about the v-axis for odd integer multiples of u, and about the u-axis for odd integer multiples of v.
- *Clamp*. In this mode, the colors of the texels around the outer edge of the texture are simply extended when texture coordinates fall outside the normal range.
- *Border color*. In this mode, an arbitrary user-specified color is used for the region outside the [0, 1] texture coordinate range.

These texture addressing modes are depicted in Figure 11.16.

Texture Formats

Texture bitmaps can be stored on disk in virtually any image format, provided your game engine includes the code necessary to read it into memory. Common formats include Targa (.tga), Portable Network Graphics (.png), Windows Bitmap (.bmp) and Tagged Image File Format (.tif). In memory, textures are usually represented as two-dimensional (strided) arrays of pixels using various color formats, including RGB888, RGBA8888, RGB565, RGBA5551 and so on.

Most modern graphics cards and graphics APIs support *compressed textures*. DirectX supports a family of compressed formats known as DXT or S3 Texture Compression (S3TC). We won't cover the details here, but the basic idea is to break the texture into 4 × 4 blocks of pixels and use a small

调色板用于存储每个块的颜色。您可以在 http://en.wikipedia.org/wiki/S3_Texture_Compression 上了解有关 S3 压缩纹理格式的更多信息。

压缩纹理比未压缩纹理具有明显的优势，即占用更少的内存。此外，压缩纹理还具有意想不到的优势，即渲染速度更快。S3 压缩纹理之所以能够实现这种加速，是因为它采用了更利于缓存的内存访问模式——将 4×4 个相邻像素块存储在单个 64 位或 128 位机器字中——并且一次可以将更多纹理内容放入缓存中。压缩纹理确实存在压缩伪影。虽然这些异常通常不易察觉，但在某些情况下必须使用未压缩纹理。

纹素密度和 Mipmapping

想象一下，渲染一个全屏四边形（由两个三角形组成的矩形），该四边形已映射了一张分辨率与屏幕分辨率完全匹配的纹理。在这种情况下，每个纹素都精确映射到屏幕上的单个像素，我们称纹素密度（纹素与像素的比率）为 1。当从远处观看同一个四边形时，其屏幕区域会变小。纹理的分辨率没有变化，因此四边形的纹素密度现在大于 1（这意味着每个像素由多个纹素构成）。

显然，纹素密度不是一个固定的量——它会随着纹理映射对象相对于相机的移动而变化。纹素密度影响内存消耗和三维场景的视觉质量。当纹素密度远小于 1 时，纹素会变得比屏幕上的像素大得多，您可以开始看到纹素的边缘。这会破坏幻觉。当纹素密度远大于 1 时，许多纹素会贡献给屏幕上的单个像素。这会导致莫尔条纹，如图 11.17 所示。更糟糕的是，像素的颜色可能会出现游动和闪烁，因为像素边界内的不同纹素会根据相机角度或位置的细微变化主导其颜色。如果玩家永远无法靠近远处的物体，那么使用非常高的纹素密度渲染该物体也会浪费内存。毕竟，如果没有人会看到所有细节，为什么要在内存中保留如此高分辨率的纹理呢？

理想情况下，我们希望始终保持纹理像素密度接近于 1，无论近处还是远处的物体。这不可能精确实现，但可以通过一种名为 mipmapping 的技术来近似实现。对于每张纹理，我们创建一系列较低分辨率的位图，每张位图的宽度和高度分别是其前一张的一半。我们将这些图像称为 mipmap 或 mip 级别。例如，一张 64×64 的纹理将具有

color palette to store the colors for each block. You can read more about S3 compressed texture formats at http://en.wikipedia.org/wiki/S3_Texture_Compression.

Compressed textures have the obvious benefit of using less memory than their uncompressed counterparts. An additional unexpected plus is that they are faster to render with as well. S3 compressed textures achieve this speed-up because of more cache-friendly memory access patterns—4×4 blocks of adjacent pixels are stored in a single 64- or 128-bit machine word—and because more of the texture can fit into the cache at once. Compressed textures do suffer from compression artifacts. While the anomalies are usually not noticeable, there are situations in which uncompressed textures must be used.

Texel Density and Mipmapping

Imagine rendering a full-screen quad (a rectangle composed of two triangles) that has been mapped with a texture whose resolution exactly matches that of the screen. In this case, each texel maps exactly to a single pixel on-screen, and we say that the *texel density* (ratio of texels to pixels) is one. When this same quad is viewed at a distance, its on-screen area becomes smaller. The resolution of the texture hasn't changed, so the quad's texel density is now greater than one (meaning that more than one texel is contributing to each pixel).

Clearly texel density is not a fixed quantity—it changes as a texture-mapped object moves relative to the camera. Texel density affects the memory consumption and the visual quality of a three-dimensional scene. When the texel density is much less than one, the texels become significantly larger than a pixel on-screen, and you can start to see the edges of the texels. This destroys the illusion. When texel density is much greater than one, many texels contribute to a single pixel on-screen. This can cause a *moiré banding pattern*, as shown in Figure 11.17. Worse, a pixel's color can appear to swim and flicker as different texels within the boundaries of the pixel dominate its color depending on subtle changes in camera angle or position. Rendering a distant object with a very high texel density can also be a waste of memory if the player can *never* get close to it. After all, why keep such a high-res texture in memory if no one will ever see all that detail?

Ideally we'd like to maintain a texel density that is close to one at all times, for both nearby and distant objects. This is impossible to achieve exactly, but it can be approximated via a technique called *mipmapping*. For each texture, we create a sequence of lower-resolution bitmaps, each of which is one-half the width and one-half the height of its predecessor. We call each of these images a *mipmap*, or *mip level*. For example, a 64×64 texture would have the

图 11.17. 大于 1 的纹素密度会导致莫尔条纹。

以下 mip 级别：64×64、32×32、16×16、8×8、4×4、2×2 和 1×1，如图 1 1.18 所示。对纹理进行 mip 映射后，图形硬件会根据三角形与相机的距离选择适当的 mip 级别，以试图保持接近于 1 的纹素密度。例如，如果纹理在屏幕上占据 40×40 的区域，则可能选择 64×64 mip 级别；如果同一纹理仅占据 10×10 的区域，则可能使用 16×16 mip 级别。正如我们将在下面看到的，三线性过滤允许硬件对两个相邻的 mip 级别进行采样并混合结果。在这种情况下，可以通过将 16 × 16 和 8 × 8 mip 级别混合在一起来映射 10 × 10 区域。

图 11.18。64 × 64 纹理的 Mip 级别。

世界空间纹理像素密度

“纹素密度”一词也可以用来描述纹理表面上纹素与世界空间面积的比率。例如，一个2米的立方体，如果映射了256×256的纹理，其纹素密度为 256 2 /2 2 = 16,384 。为了区别于我们之前讨论的屏幕空间纹素密度，我将称之为世界空间纹素密度。

Figure 11.17. A texel density greater than one can lead to a moiré pattern.

following mip levels: 64 × 64, 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2 and 1 × 1, as shown in Figure 11.18. Once we have mipmapped our textures, the graphics hardware selects the appropriate mip level based on a triangle's distance away from the camera, in an attempt to maintain a texel density that is close to one. For example, if a texture takes up an area of 40 × 40 on-screen, the 64 × 64 mip level might be selected; if that same texture takes up only a 10 × 10 area, the 16 × 16 mip level might be used. As we'll see below, *trilinear filtering* allows the hardware to sample two adjacent mip levels and blend the results. In this case, a 10 × 10 area might be mapped by blending the 16 × 16 and 8 × 8 mip levels together.

Figure 11.18. Mip levels for a 64 × 64 texture.

World-Space Texel Density

The term "texel density" can also be used to describe the ratio of texels to world-space area on a textured surface. For example, a 2 m cube mapped with a 256 × 256 texture would have a texel density of $256^2/2^2 = 16{,}384$. I will call this *world-space texel density* to differentiate it from the screen-space texel density we've been discussing thus far.

世界空间纹素密度不必接近 1，实际上，具体值通常会远大于 1，这完全取决于您选择的世界单位。尽管如此，对于纹理映射对象而言，保持合理一致的世界空间纹素密度仍然至关重要。例如，我们期望立方体的六个面都占据相同的纹理区域。如果不是这样，立方体一侧的纹理分辨率就会低于另一侧，玩家会注意到这一点。许多游戏工作室会为其美术团队提供指导方针和引擎内纹素密度可视化工具，以确保游戏中的所有对象都具有合理一致的世界空间纹素密度。

纹理过滤

在渲染带纹理三角形的像素时，图形硬件会根据像素中心在纹理空间中的位置来采样纹理贴图。纹素和像素之间通常不存在明确的一一映射，像素中心可能位于纹理空间中的任何位置，包括两个或多个纹素之间的边界。因此，图形硬件通常需要采样多个纹素，并将结果颜色混合，才能得出实际采样的纹素颜色。我们称之为纹理过滤。

大多数显卡支持以下类型的纹理过滤：

• 最近邻。在这种粗略的方法中，选择中心最接近像素中心的纹素。启用 mip 映射后，将选择分辨率最接近但大于实现屏幕空间纹素密度 1 所需的理想理论分辨率的 mip 级别。

• 双线性。在这种方法中，对像素中心周围的四个纹素进行采样，最终颜色是这些纹素颜色的加权平均值（其中权重基于纹素中心与像素中心的距离）。启用 mip 映射后，将选择最近的 mip 级别。

• 三线性。在这种方法中，对两个最接近的 mip 级别（一个分辨率高于理想值，另一个分辨率较低）分别使用双线性滤波，然后对这些结果进行线性插值。这消除了屏幕上 mip 级别之间突兀的视觉边界。

• 各向异性。双线性和三线性过滤都会采样 2×2 的正方形纹理像素块。当正面观看纹理表面时，这样做是正确的，但当表面相对于虚拟屏幕平面呈倾斜角度时，这样做是不正确的。各向异性过滤会采样与视图对应的梯形区域内的纹理像素。

World-space texel density need not be close to one, and in fact the specific value will usually be much greater than one and depends entirely upon your choice of world units. Nonetheless, it is important for objects to be texture mapped with a reasonably consistent world-space texel density. For example, we would expect all six sides of a cube to occupy the same texture area. If this were not the case, the texture on one side of the cube would have a lower-resolution appearance than another side, which can be noticeable to the player. Many game studios provide their art teams with guidelines and in-engine texel density visualization tools in an effort to ensure that all objects in the game have a reasonably consistent world-space texel density.

Texture Filtering

When rendering a pixel of a textured triangle, the graphics hardware samples the texture map by considering where the pixel center falls in texture space. There is usually not a clean one-to-one mapping between texels and pixels, and pixel centers can fall at any place in texture space, including directly on the boundary between two or more texels. Therefore, the graphics hardware usually has to sample more than one texel and blend the resulting colors to arrive at the actual sampled texel color. We call this *texture filtering*.

Most graphics cards support the following kinds of texture filtering:

- *Nearest neighbor*. In this crude approach, the texel whose center is closest to the pixel center is selected. When mipmapping is enabled, the mip level is selected whose resolution is nearest to but greater than the ideal theoretical resolution needed to achieve a screen-space texel density of one.
- *Bilinear*. In this approach, the four texels surrounding the pixel center are sampled, and the resulting color is a weighted average of their colors (where the weights are based on the distances of the texel centers from the pixel center). When mipmapping is enabled, the nearest mip level is selected.
- *Trilinear*. In this approach, bilinear filtering is used on each of the two nearest mip levels (one higher-res than the ideal and the other lower-res), and these results are then linearly interpolated. This eliminates abrupt visual boundaries between mip levels on-screen.
- *Anisotropic*. Both bilinear and trilinear filtering sample 2×2 square blocks of texels. This is the right thing to do when the textured surface is being viewed head-on, but it's incorrect when the surface is at an oblique angle relative to the virtual screen plane. Anisotropic filtering samples texels within a trapezoidal region corresponding to the view

角度，从而提高从角度观看时纹理表面的质量。

11.1.2.6 材料

材质是对网格视觉属性的完整描述。这包括映射到网格表面的纹理的规范，以及各种高级属性，例如渲染网格时要使用的着色器程序、这些着色器的输入参数，以及控制图形加速硬件本身功能的其他参数。

虽然从技术上讲，顶点属性是表面属性描述的一部分，但它们并不被视为材质的一部分。然而，它们会随着网格一起出现，因此网格-材质对包含了渲染对象所需的所有信息。网格-材质对有时被称为渲染包，而“几何图元”一词有时也会扩展为涵盖网格-材质对。

3D 模型通常使用多种材质。例如，人体模型的头发、皮肤、眼睛、牙齿以及各种衣物都会使用不同的材质。因此，网格通常会被划分为多个子网格，每个子网格会映射到一种材质。OGRE 渲染引擎通过其 Ogre::SubMesh 类实现了这种设计。

图 11.19。经典“康奈尔盒子”场景的变体，展示了逼真的光照如何使最简单的场景看起来栩栩如生。

angle, thereby increasing the quality of textured surfaces when viewed at an angle.

11.1.2.6 Materials

A *material* is a complete description of the visual properties of a mesh. This includes a specification of the textures that are mapped to its surface and also various higher-level properties, such as which shader programs to use when rendering the mesh, the input parameters to those shaders and other parameters that control the functionality of the graphics acceleration hardware itself.

While technically part of the surface properties description, vertex attributes are not considered to be part of the material. However, they come along for the ride with the mesh, so a mesh-material pair contains all the information we need to render the object. Mesh-material pairs are sometimes called *render packets*, and the term “geometric primitive” is sometimes extended to encompass mesh-material pairs as well.

A 3D model typically uses more than one material. For example, a model of a human would have separate materials for the hair, skin, eyes, teeth and various kinds of clothing. For this reason, a mesh is usually divided into *submeshes*, each mapped to a single material. The OGRE rendering engine implements this design via its `Ogre::SubMesh` class.

Figure 11.19. A variation on the classic “Cornell box” scene illustrating how realistic lighting can make even the simplest scene appear photorealistic.

图 11.20.《最后生还者：重制版》中的一个场景（© 2014/TM SIE。由顽皮狗（PlayStation 4）渲染时未使用纹理。（参见彩色图版 XV。）

11.1.3 照明基础知识

光照是所有 CG 渲染的核心。如果没有良好的光照，原本精美的场景模型也会显得平淡无奇。同样，即使是最简单的场景，只要光照准确，也能呈现出极其逼真的效果。图 11.19 所示的经典“康奈尔盒子”场景就是一个很好的例子。

顽皮狗的《最后生还者：重制版》中的一系列截图很好地说明了光照的重要性。图 11.20 中的场景渲染时没有使用纹理。图 11.21 展示了应用了漫反射纹理的同一场景。图 11.22 展示了完全光照的场景。请注意，当光照应用于场景时，真实感会有明显的提升。

“着色”一词通常被用作对光照及其他视觉效果的宽泛概括。因此，“着色”涵盖了顶点的程序化变形（用于模拟水面运动）、毛发曲线或毛皮外壳的生成、高阶曲面的曲面细分，以及渲染场景所需的几乎所有其他计算。

在接下来的章节中，我们将介绍光照基础知识，以便理解图形硬件和渲染管线。我们将在11.3节中继续讨论光照主题，并探讨一些高级光照和着色技术。

11.1.3.1 局部和全局光照模型

渲染引擎使用各种光-表面和光-体积相互作用的数学模型，称为光传输模型。最简单的模型只考虑直接光照，即光从单个物体发射出来，反射到

Figure 11.20. A scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) rendered without textures. (See Color Plate XV.)

11.1.3 Lighting Basics

Lighting is at the heart of all CG rendering. Without good lighting, an otherwise beautifully modeled scene will look flat and artificial. Likewise, even the simplest of scenes can be made to look extremely realistic when it is lit accurately. The classic “Cornell box” scene, shown in Figure 11.19, is an excellent example of this.

The sequence of screenshots from Naughty Dog’s *The Last of Us: Remastered* is another good illustration of the importance of lighting. In Figure 11.20, the scene is rendered without textures. Figure 11.21 shows the same scene with diffuse textures applied. The fully lit scene is shown in Figure 11.22. Notice the marked jump in realism when lighting is applied to the scene.

The term *shading* is often used as a loose generalization of lighting plus other visual effects. As such, “shading” encompasses procedural deformation of vertices to simulate the motion of a water surface, generation of hair curves or fur shells, tessellation of high-order surfaces, and pretty much any other calculation that’s required to render a scene.

In the following sections, we’ll lay the foundations of lighting that we’ll need in order to understand graphics hardware and the rendering pipeline. We’ll return to the topic of lighting in Section 11.3, where we’ll survey some advanced lighting and shading techniques.

11.1.3.1 Local and Global Illumination Models

Rendering engines use various mathematical models of light-surface and light-volume interactions called *light transport models*. The simplest models only account for *direct lighting* in which light is emitted, bounces off a single

图 11.21。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的同一场景，仅应用了漫反射纹理。（参见彩色图 XVI。）

图 11.22。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的场景，灯光充足。（参见彩色图版 XVII。）

场景中的物体，然后直接进入虚拟相机的成像平面。这种简单的模型被称为局部光照模型，因为只考虑光线对单个物体的局部影响；在局部光照模型中，物体彼此的外观互不□□影响。毫不奇怪，局部模型最早被用于游戏，并且至今仍在使用——在某些情况下，局部光照可以产生令人惊讶的逼真效果。

Figure 11.21. The same scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) with only diffuse textures applied. (See Color Plate XVI.)

Figure 11.22. Scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) with full lighting. (See Color Plate XVII.)

object in the scene, and then proceeds directly to the imaging plane of the virtual camera. Such simple models are called *local illumination models*, because only the local effects of light on a single object are considered; objects do not affect one another's appearance in a local lighting model. Not surprisingly, local models were the first to be used in games, and they are still in use today—local lighting can produce surprisingly realistic results in some circumstances.

真正的照片级真实感只有通过考虑间接光照才能实现，即光线在到达虚拟相机之前会从许多表面反射多次。考虑间接光照的光照模型称为全局光照模型。一些全局光照模型旨在模拟一种特定的视觉现象，例如产生真实的阴影、模拟反射表面、考虑物体之间的相互反射（一个物体的颜色会影响周围物体的颜色）以及模拟焦散效果（水或闪亮金属表面的强烈反射）。其他全局光照模型试图对各种光学现象提供整体说明。光线追踪和光能传递方法就是此类技术的例子。

全局光照可以用一个称为渲染方程或着色方程的数学公式完整描述。该方程由 JT Kajiya 于 1986 年在一篇具有开创性的 SIGGRAPH 论文中提出。从某种意义上说，每种渲染技术都可以被认为是渲染方程的完整或部分解，尽管它们在求解该方程的基本方法以及所做的假设、简化和近似处理方面有所不同。有关渲染方程的更多详细信息，请参阅 http://en.wikipedia.org/wiki/Rendering_equation、[10]、[2] 以及几乎所有其他关于高级渲染和光照的文献。

11.1.3.2 冯氏照明模型

游戏渲染引擎最常用的局部光照模型是 Phong 反射模型。该模型将从表面反射的光线建模为三个不同项的总和：

- 环境光项模拟了场景的整体光照水平。它是场景中间接反射光量的粗略近似值。间接反射是导致阴影区域不完全呈现黑色的原因。
- 漫反射项表示从每个直射光源向各个方向均匀反射的光线。这很好地近似了真实光线在无光泽表面（例如木块或布料）反射的方式。
- 镜面反射项模拟了我们在观察光滑表面时有时会看到的明亮高光。当视角与光源直接反射的路径紧密对齐时，就会出现镜面高光。

图 11.23 显示了环境光、漫反射和镜面反射项如何加在一起产生表面的最终强度和颜色。

True photorealism can only be achieved by accounting for *indirect lighting*, where light bounces multiple times off many surfaces before reaching the virtual camera. Lighting models that account for indirect lighting are called *global illumination models*. Some global illumination models are targeted at simulating one specific visual phenomenon, such as producing realistic shadows, modeling reflective surfaces, accounting for interreflection between objects (where the color of one object affects the colors of surrounding objects), and modeling caustic effects (the intense reflections from water or a shiny metal surface). Other global illumination models attempt to provide a holistic account of a wide range of optical phenomena. Ray tracing and radiosity methods are examples of such technologies.

Global illumination is described completely by a mathematical formulation known as the *rendering equation* or *shading equation*. It was introduced in 1986 by J. T. Kajiya as part of a seminal SIGGRAPH paper. In a sense, every rendering technique can be thought of as a full or partial solution to the rendering equation, although they differ in their fundamental approach to solving it and in the assumptions, simplifications and approximations they make. See http://en.wikipedia.org/wiki/Rendering_equation, [10], [2] and virtually any other text on advanced rendering and lighting for more details on the rendering equation.

11.1.3.2 The Phong Lighting Model

The most common local lighting model employed by game rendering engines is the *Phong* reflection model. It models the light reflected from a surface as a sum of three distinct terms:

- The *ambient* term models the overall lighting level of the scene. It is a gross approximation of the amount of indirect bounced light present in the scene. Indirect bounces are what cause regions in shadow not to appear totally black.
- The *diffuse* term accounts for light that is reflected uniformly in all directions from each direct light source. This is a good approximation to the way in which real light bounces off a matte surface, such as a block of wood or a piece of cloth.
- The *specular* term models the bright highlights we sometimes see when viewing a glossy surface. Specular highlights occur when the viewing angle is closely aligned with a path of direct reflection from a light source.

Figure 11.23 shows how the ambient, diffuse and specular terms add together to produce the final intensity and color of a surface.

图 11.23. 将环境光、漫反射和镜面反射相加来计算 Phong 反射。

为了计算表面上特定点的 Phong 反射，我们需要一些输入参数。Phong 模型通常独立应用于所有三个颜色通道（R、G 和 B），因此以下讨论中的所有颜色参数都是三元素向量。Phong 模型的输入包括：

- 观察方向向量 V = [V x V y V z]，从反射点延伸到虚拟相机的焦点（即相机的世界空间"前"向量的否定）；
- 三个颜色通道的环境光强度，A =
- 光线照射到表面时的表面法线 N = [N x N y N z];
- 表面反射特性，
  - 环境反射率 k A = [k AR k AG k AB] ，
  - 漫反射率 k D = [k DR k DG k DB] ，
  - 镜面反射率 k S = [k SR k SG k SB] ，
  - 镜面"光泽度"指数 α；
- 并且，对于每个光源 i ，
  - 光的颜色和强度 C i = [C iR C iG C iB] ，
  - 从反射点到光源的方向矢量 Li。

在Phong模型中，从某一点反射的光的强度I可以用以下矢量方程表示：

$$\mathbf{I} = (\mathbf{k}_A \otimes \mathbf{A}) + \sum_i \left[\mathbf{k}_D(\mathbf{N} \cdot \mathbf{L}_i) + \mathbf{k}_S(\mathbf{R}_i \cdot \mathbf{V})^\alpha\right] \otimes \mathbf{C}_i,$$

其中，求和是对影响该点的所有光线 i 进行求和。回想一下，运算符 □ 表示两个向量的分量乘法

Figure 11.23. Ambient, diffuse and specular terms are summed to calculate Phong reflection.

To calculate Phong reflection at a specific point on a surface, we require a number of input parameters. The Phong model is normally applied to all three color channels (R, G and B) independently, so all of the color parameters in the following discussion are three-element vectors. The inputs to the Phong model are:

- the viewing direction vector $\mathbf{V} = \begin{bmatrix} V_x & V_y & V_z \end{bmatrix}$, which extends from the reflection point to the virtual camera's focal point (i.e., the negation of the camera's world-space "front" vector);
- the ambient light intensity for the three color channels, $\mathbf{A} = \begin{bmatrix} A_R & A_G & A_B \end{bmatrix}$;
- the surface normal $\mathbf{N} = \begin{bmatrix} N_x & N_y & N_z \end{bmatrix}$ at the point the light ray impinges on the surface;
- the surface reflectance properties, which are
  - the ambient reflectivity $\mathbf{k}_A = \begin{bmatrix} k_{AR} & k_{AG} & k_{AB} \end{bmatrix}$,
  - the diffuse reflectivity $\mathbf{k}_D = \begin{bmatrix} k_{DR} & k_{DG} & k_{DB} \end{bmatrix}$,
  - the specular reflectivity $\mathbf{k}_S = \begin{bmatrix} k_{SR} & k_{SG} & k_{SB} \end{bmatrix}$,
  - a specular "glossiness" exponent α;
- and, for each light source i,
  - the light's color and intensity $\mathbf{C}_i = \begin{bmatrix} C_{iR} & C_{iG} & C_{iB} \end{bmatrix}$,
  - the direction vector $\mathbf{L}_i$ from the reflection point to the light source.

In the Phong model, the intensity $\mathbf{I}$ of light reflected from a point can be expressed with the following vector equation:

$$\mathbf{I} = (\mathbf{k}_A \otimes \mathbf{A}) + \sum_i \left[\mathbf{k}_D(\mathbf{N} \cdot \mathbf{L}_i) + \mathbf{k}_S(\mathbf{R}_i \cdot \mathbf{V})^\alpha\right] \otimes \mathbf{C}_i,$$

where the sum is taken over all lights i affecting the point in question. Recall that the operator $\otimes$ represents the *component-wise* multiplication of two vectors

图 11.24. 根据原始照明矢量 L 和表面法线 N 计算反射照明矢量 R。

（即所谓的 Hadamard 积）。该表达式可以分解为三个标量方程，每个颜色通道一个，如下所示：

$$I_R = k_{AR}A_R + \sum_i \left[k_{DR}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SR}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iR},$$
$$I_G = k_{AG}A_G + \sum_i \left[k_{DG}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SG}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iG},$$
$$I_B = k_{AB}A_B + \sum_i \left[k_{DB}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SB}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iB}.$$

在这些方程中，矢量 R i = 射线关于表面法线 N 的方 反射
向矢量 L i 。
向量 R i 可以通过一些向量数学运算轻松计算出来（参见图 11.24）。任何向量都可以表示为其法向分量和切向分量之和。例如，我们可以将光线方向向量 L 分解如下：

$$\mathbf{L} = \mathbf{L}_N + \mathbf{L}_T.$$

我们知道点积（N · L）表示 L 在曲面上的法线投影（一个标量）。因此法向量 LN 就是单位法向量 N 乘以点积的结果：

$$\mathbf{L}_N = (\mathbf{N}\cdot\mathbf{L})\mathbf{N}.$$

反射矢量 R 的法向分量与 L 相同，但切向分量相反（−LT）。因此，我们可以如下计算 R：

$$\begin{aligned}\mathbf{R} &= \mathbf{L}_N - \mathbf{L}_T\\ &= \mathbf{L}_N - (\mathbf{L} - \mathbf{L}_N)\\ &= 2\mathbf{L}_N - \mathbf{L};\\ \mathbf{R} &= 2(\mathbf{N}\cdot\mathbf{L})\mathbf{N} - \mathbf{L}_T\end{aligned}$$

Figure 11.24. Calculation of the reflected lighting vector **R** from the original lighting vector **L** and the surface normal **N**.

(the so-called Hadamard product). This expression can be broken into three scalar equations, one for each color channel, as follows:

$$I_R = k_{AR}A_R + \sum_i \left[k_{DR}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SR}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iR},$$
$$I_G = k_{AG}A_G + \sum_i \left[k_{DG}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SG}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iG},$$
$$I_B = k_{AB}A_B + \sum_i \left[k_{DB}(\mathbf{N}\cdot\mathbf{L}_i) + k_{SB}(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right] C_{iB}.$$

In these equations, the vector $\mathbf{R_i} = \begin{bmatrix}R_{ix} & R_{iy} & R_{iz}\end{bmatrix}$ is the *reflection* of the light ray's direction vector $\mathbf{L}_i$ about the surface normal **N**.

The vector $\mathbf{R_i}$ can be easily calculated via a bit of vector math (see Figure 11.24). Any vector can be expressed as a sum of its normal and tangential components. For example, we can break up the light direction vector **L** as follows:

$$\mathbf{L} = \mathbf{L}_N + \mathbf{L}_T.$$

We know that the dot product $(\mathbf{N}\cdot\mathbf{L})$ represents the projection of **L** normal to the surface (a scalar quantity). So the normal component $\mathbf{L}_N$ is just the unit normal vector **N** scaled by this dot product:

$$\mathbf{L}_N = (\mathbf{N}\cdot\mathbf{L})\mathbf{N}.$$

The reflected vector **R** has the same normal component as **L** but the *opposite* tangential component $(-\mathbf{L}_T)$. So we can find **R** as follows:

$$\begin{aligned}\mathbf{R} &= \mathbf{L}_N - \mathbf{L}_T\\ &= \mathbf{L}_N - (\mathbf{L} - \mathbf{L}_N)\\ &= 2\mathbf{L}_N - \mathbf{L};\\ \mathbf{R} &= 2(\mathbf{N}\cdot\mathbf{L})\mathbf{N} - \mathbf{L}_T\end{aligned}$$

该方程可用于找到与光方向 Li 对应的所有 R i 值。

布林-冯

Blinn-Phong 光照模型是 Phong 着色模型的一种变体，它以略微不同的方式计算镜面反射。我们将向量 H 定义为位于视线向量 V 和光线方向向量 L 中间的向量。Blinn-Phong 的镜面反射分量为（N · H）α，而不是 Phong 的（R · V）α。指数 α 与 Phong 指数 α 略有不同，但其值的选择是为了与等效的 Phong 镜面反射项紧密匹配。

Blinn-Phong 模型虽然提升了运行时效率，但牺牲了一定的精度，尽管对于某些类型的表面，它实际上比 Phong 模型更接近经验结果。Blinn-Phong 模型几乎只用于早期的电脑游戏，并且被硬编码到早期 GPU 的固定功能管线中。更多详情，请参阅 http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model。

BRDF 图

冯氏光照模型中的这三个术语是广义局部反射模型的特例，该模型被称为双向反射分布函数 (BRDF)。BRDF 计算沿给定观察方向 V 的出射（反射）辐射度与沿入射光线 L 的入射辐射度之比。

BRDF 可以可视化为半球形图，其中距原点的径向距离表示从该方向观察反射点时会看到的光的强度。漫反射 Phong 项为 kD(N · L)。该项仅考虑入射照明光线 L，而不考虑视角 V。因此，该项的值对于所有视角都是相同的。如果我们在三维空间中将该项绘制为视角的函数，它看起来就像一个以我们计算 Phong 反射的点为中心的半球。图 11.25 以二维形式显示了这一情况。

Phong 模型的镜面反射项为 kD(R · V)α。该项同时依赖于照明方向 L 和观察方向 V。当观察角度与照明方向 L 绕表面法线的反射 R 紧密对齐时，会产生镜面“热点”。然而，随着观察角度偏离反射照明方向，其贡献会迅速衰减。图 11.26 以二维形式展示了这一现象。

This equation can be used to find all of the $\mathbf{R}_i$ values corresponding to the light directions $\mathbf{L}_i$.

Blinn-Phong

The *Blinn-Phong* lighting model is a variation on Phong shading that calculates specular reflection in a slightly different way. We define the vector **H** to be the vector that lies halfway between the view vector **V** and the light direction vector **L**. The Blinn-Phong specular component is then $(\mathbf{N} \cdot \mathbf{H})^a$, as opposed to Phong's $(\mathbf{R} \cdot \mathbf{V})^\alpha$. The exponent a is slightly different than the Phong exponent α, but its value is chosen in order to closely match the equivalent Phong specular term.

The Blinn-Phong model offers increased runtime efficiency at the cost of some accuracy, although it actually matches empirical results more closely than Phong for some kinds of surfaces. The Blinn-Phong model was used almost exclusively in early computer games and was hard-wired into the fixed-function pipelines of early GPUs. See http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model for more details.

BRDF Plots

The three terms in the Phong lighting model are special cases of a general local reflection model known as a *bidirectional reflection distribution function* (BRDF). A BRDF calculates the ratio of the outgoing (reflected) radiance along a given viewing direction **V** to the incoming irradiance along the incident ray **L**.

A BRDF can be visualized as a hemispherical plot, where the radial distance from the origin represents the intensity of the light that would be seen if the reflection point were viewed from that direction. The *diffuse* Phong reflection term is $\mathbf{k}_D(\mathbf{N} \cdot \mathbf{L})$. This term only accounts for the incoming illumination ray **L**, not the viewing angle **V**. Hence the value of this term is the same for all viewing angles. If we were to plot this term as a function of the viewing angle in three dimensions, it would look like a hemisphere centered on the point at which we are calculating the Phong reflection. This is shown in two dimensions in Figure 11.25.

The specular term of the Phong model is $\mathbf{k}_D(\mathbf{R} \cdot \mathbf{V})^\alpha$. This term is dependent on both the illumination direction **L** and the viewing direction **V**. It produces a specular "hot spot" when the viewing angle aligns closely with the reflection **R** of the illumination direction **L** about the surface normal. However, its contribution falls off very quickly as the viewing angle diverges from the reflected illumination direction. This is shown in two dimensions in Figure 11.26.

图 11.25。Phong 反射模型的漫反射项取决于 N · L，但与视角 V 无关。

图 11.26 当视角 V 与反射光方向 R 重合时，Phong 反射模型的镜面反射项达到最大值，而当 V 偏离 R 时，镜面反射项迅速下降。

11.1.3.3 建模光源

除了建模光源与表面的相互作用之外，我们还需要描述场景中的光源。与实时渲染中的所有事物一样，我们使用各种简化模型来近似真实世界的光源。

静态照明

最快的光照计算是根本不需要进行的计算。因此，光照计算尽可能离线执行。我们可以预先计算网格顶点的 Phong 反射，并将结果存储为漫反射顶点颜色属性。我们还可以预先计算每个像素的光照，并将结果存储在一种称为光照贴图的纹理贴图中。在运行时，光照贴图纹理会被投影到场景中的对象上，以确定光照对它们的影响。

你可能会想，为什么我们不直接将光照信息烘焙到场景中的漫反射纹理中呢？原因有几个。首先，漫反射纹理贴图通常会在整个场景中平铺和/或重复出现，

Figure 11.25. The diffuse term of the Phong reflection model is dependent upon $\mathbf{N} \cdot \mathbf{L}$ but is independent of the viewing angle $\mathbf{V}$.

Figure 11.26. The specular term of the Phong reflection model is at its maximum when the viewing angle $\mathbf{V}$ coincides with the reflected light direction $\mathbf{R}$ and drops off quickly as $\mathbf{V}$ diverges from $\mathbf{R}$.

11.1.3.3 Modeling Light Sources

In addition to modeling the light's interactions with surfaces, we need to describe the sources of light in the scene. As with all things in real-time rendering, we approximate real-world light sources using various simplified models.

Static Lighting

The fastest lighting calculation is the one you don't do at all. Lighting is therefore performed offline whenever possible. We can precalculate Phong reflection at the vertices of a mesh and store the results as diffuse vertex color attributes. We can also precalculate lighting on a per-pixel basis and store the results in a kind of texture map known as a *light map*. At runtime, the light map texture is projected onto the objects in the scene in order to determine the light's effects on them.

You might wonder why we don't just bake lighting information directly into the diffuse textures in the scene. There are a few reasons for this. For one thing, diffuse texture maps are often tiled and/or repeated throughout a scene,

因此，将光照烘焙到其中并不实际。相反，通常会为每个光源生成一张光照贴图，并将其应用于该光源影响范围内的任何物体。这种方法允许动态物体经过光源并被其正确照亮。这也意味着我们的光照贴图的分辨率可以与漫反射纹理贴图不同（通常更低）。最后，“纯”光照贴图通常比包含漫反射颜色信息的光照贴图压缩效果更好。

环境光

环境光对应于冯氏光照模型中的环境光项。该项与视角无关，且没有特定的方向。因此，环境光用单一颜色表示，对应于冯氏方程中的 A 颜色项（运行时根据表面的环境光反射率 k A 进行缩放）。环境光的强度和颜色在游戏世界中可能因区域而异。

定向灯

图11.27。定向光源模型。

平行光模拟□□的是距离被照射表面无限远的光源，例如太阳。平行光发出的光线是平行的，并且光线本身在游戏世界中没有任何特定的位置。因此，平行光被建模为光色 C 和方向向量 L。平行光如图 11.27 所示。

点（全向）光源

图11.28 点光源模型。

点光源（全向光）在游戏世界中具有独特的位置，并向所有方向均匀辐射。光的强度通常被认为与距光源距离的平方成正比，并且超过预定义的最大半径后，其效果将被限制为零。点光源的建模由光源位置 P、光源颜色/强度 C 和最大半径 rmax 组成。渲染引擎仅将点光源的效果应用于其影响范围内的表面（这是一项重大优化）。图 11.28 展示了一个点光源。

聚光灯

聚光灯的作用类似于点光源，其光线被限制在一个锥形区域内，就像手电筒一样。通常指定两个锥体，分别具有内角和外角。在内锥内，光线被认为处于最大强度。

so baking lighting into them wouldn't be practical. Instead, a single light map is usually generated per light source and applied to any objects that fall within that light's area of influence. This approach permits dynamic objects to move past a light source and be properly illuminated by it. It also means that our light maps can be of a different (often lower) resolution than our diffuse texture maps. Finally, a "pure" light map usually compresses better than one that includes diffuse color information.

Ambient Lights

An *ambient light* corresponds to the ambient term in the Phong lighting model. This term is independent of the viewing angle and has no specific direction. An ambient light is therefore represented by a single color, corresponding to the **A** color term in the Phong equation (which is scaled by the surface's ambient reflectivity $\mathbf{k}_A$ at runtime). The intensity and color of ambient light may vary from region to region within the game world.

Directional Lights

Figure 11.27. Model of a directional light source.

A *directional light* models a light source that is effectively an infinite distance away from the surface being illuminated—like the sun. The rays emanating from a directional light are parallel, and the light itself does not have any particular location in the game world. A directional light is therefore modeled as a light color **C** and a direction vector **L**. A directional light is depicted in Figure 11.27.

Point (Omnidirectional) Lights

Figure 11.28. Model of a point light source.

A *point light* (*omnidirectional light*) has a distinct position in the game world and radiates uniformly in all directions. The intensity of the light is usually considered to fall off with the square of the distance from the light source, and beyond a predefined maximum radius its effects are simply clamped to zero. A point light is modeled as a light position **P**, a source color/intensity **C** and a maximum radius r_{max}. The rendering engine only applies the effects of a point light to those surfaces that fall within its sphere of influence (a significant optimization). Figure 11.28 illustrates a point light.

Spot Lights

A *spot light* acts like a point light whose rays are restricted to a cone-shaped region, like a flashlight. Usually two cones are specified with an inner and an outer angle. Within the inner cone, the light is considered to be at full intensity.

光强度随着角度从内锥到外锥的增加而衰减，超过外锥后，光强度被视为零。在两个锥体内部，光强度也随着径向距离的增加而衰减。聚光灯的模型由位置 P、光源颜色 C、中心方向矢量 L、最大半径 r max 以及内锥角 θ min 和外锥角 θ max 组成。图 11.29 展示了一个聚光光源。

图11.29。点光源的模型光。

区域光

到目前为止，我们讨论过的所有光源都是从一个理想化的点辐射出来的，要么在无穷远处，要么在局部。真实的光源几乎总是具有一个非零的面积——这就是它投射的阴影中产生本影和半影的原因。

CG 工程师通常不会尝试明确地模拟区域光源，而是使用各种“技巧”来解释其行为。例如，为了模拟半影，我们可能会投射多个阴影并混合结果，或者我们可能会以某种方式模糊锐利阴影的边缘。

发光物体

场景中的某些表面本身就是光源。例如，手电筒、发光的水晶球、火箭发动机的火焰等等。发光表面可以使用自发光纹理贴图来建模——这种纹理的颜色始终保持全亮度，不受周围光照环境的影响。这种纹理可以用来定义霓虹灯、汽车前灯等等。

某些类型的自发光物体需要结合多种技术进行渲染。例如，手电筒的渲染可以使用以下几种方式：一张自发光纹理，用于呈现正面直视光束的效果；一个共置聚光灯，用于将光线投射到场景中；一个黄色半透明网格，用于模拟光锥；一些面向摄像机的透明卡片，用于模拟镜头眩光（如果引擎支持高动态范围照明，则可模拟泛光效果）；以及一张投影纹理，用于产生手电筒照射表面的焦散效果。《路易吉洋楼》中的手电筒就是这种效果组合的绝佳示例，如图 11.30 所示。

11.1.4 虚拟相机

在计算机图形学中，虚拟相机比真实相机或人眼简单得多。我们将相机视为一个理想的焦点，并在其上浮动一小段距离，形成一个称为成像矩形的矩形虚拟传感表面。

The light intensity falls off as the angle increases from the inner to the outer angle, and beyond the outer cone it is considered to be zero. Within both cones, the light intensity also falls off with radial distance. A spot light is modeled as a position **P**, a source color **C**, a central direction vector **L**, a maximum radius $r_{\max}$ and inner and outer cone angles $\theta_{\min}$ and $\theta_{\max}$. Figure 11.29 illustrates a spot light source.

Figure 11.29. Model of a spot light source.

Area Lights

All of the light sources we've discussed thus far radiate from an idealized point, either at infinity or locally. A real light source almost always has a nonzero area—this is what gives rise to the umbra and penumbra in the shadows it casts.

Rather than trying to model area lights explicitly, CG engineers often use various "tricks" to account for their behavior. For example to simulate a penumbra, we might cast multiple shadows and blend the results, or we might blur the edges of a sharp shadow in some manner.

Emissive Objects

Some surfaces in a scene are themselves light sources. Examples include flashlights, glowing crystal balls, flames from a rocket engine and so on. Glowing surfaces can be modeled using an *emissive texture map*—a texture whose colors are always at full intensity, independent of the surrounding lighting environment. Such a texture could be used to define a neon sign, a car's headlights and so on.

Some kinds of emissive objects are rendered by combining multiple techniques. For example, a flashlight might be rendered using an emissive texture for when you're looking head-on into the beam, a colocated spot light that casts light into the scene, a yellow translucent mesh to simulate the light cone, some camera-facing transparent cards to simulate lens flare (or a *bloom* effect if high dynamic range lighting is supported by the engine), and a projected texture to produce the caustic effect that a flashlight has on the surfaces it illuminates. The flashlight in *Luigi's Mansion* is a great example of this kind of effect combination, as shown in Figure 11.30.

11.1.4 The Virtual Camera

In computer graphics, the virtual camera is much simpler than a real camera or the human eye. We treat the camera as an ideal focal point with a rectangular virtual sensing surface called the *imaging rectangle* floating some small distance

图 11.30。任天堂 Wii 游戏《路易吉洋楼》中的手电筒由多种视觉效果组成，包括用于光束的半透明几何锥形、用于将光线投射到场景中的动态聚光灯、镜头上的自发光纹理以及用于产生镜头光晕的面向相机的卡片。（参见彩色图版 XVIII。）

在它前面。成像矩形由正方形或矩形的虚拟光传感器网格组成，每个传感器对应屏幕上的一个像素。渲染可以被认为是确定每个虚拟传感器记录的光的颜色和强度的过程。

11.1.4.1 视图空间

虚拟摄像机的焦点是三维坐标系（称为“视图空间”或“摄像机空间”）的原点。摄像机通常“俯视”视图空间中的正 z 轴或负 z 轴，y 轴朝上，x 轴朝左或朝右。典型的左手和右手视图空间轴如图 11.31 所示。

相机的位置和方向可以用视图到世界矩阵来指定，就像网格实例通过其模型到世界矩阵在场景中定位一样。如果我们知道相机空间的位置向量和三个单位基向量（以世界空间坐标表示），则视图到世界矩阵可以写成如下形式，其方式类似于构建模型到世界矩阵的方式：

$$\mathbf{M}_{V\to W} = \left[\begin{array}{c|c} \mathbf{i}_V & 0 \\ \mathbf{j}_V & 0 \\ \mathbf{k}_V & 0 \\ \hline \mathbf{t}_V & 1 \end{array}\right].$$

渲染三角形网格时，其顶点首先从模型空间变换到世界空间，然后再从世界空间变换到观察空间。为了执行后者的变换，我们需要世界空间到观察空间的矩阵，它是观察空间到世界空间的矩阵的逆。这个矩阵有时也称为观察矩阵：

$$\mathbf{M}_{W\to V} = \mathbf{M}^{\mathrm{V\to W = M\ 视图。}}$$

Figure 11.30. The flashlight in *Luigi's Mansion* by Nintendo (Wii) is composed of numerous visual effects, including a cone of translucent geometry for the beam, a dynamic spot light to cast light into the scene, an emissive texture on the lens and camera-facing cards for the lens flare. (See Color Plate XVIII.)

in front of it. The imaging rectangle consists of a grid of square or rectangular virtual light sensors, each corresponding to a single pixel on-screen. Rendering can be thought of as the process of determining what color and intensity of light would be recorded by each of these virtual sensors.

11.1.4.1 View Space

The focal point of the virtual camera is the origin of a 3D coordinate system known as *view space* or *camera space*. The camera usually "looks" down the positive or negative z-axis in view space, with y up and x to the left or right. Typical left- and right-handed view-space axes are illustrated in Figure 11.31.

The camera's position and orientation can be specified using a view-to-world matrix, just as a mesh instance is located in the scene with its model-to-world matrix. If we know the position vector and three unit basis vectors of camera space, expressed in world-space coordinates, the view-to-world matrix can be written as follows, in a manner analogous to that used to construct a model-to-world matrix:

$$\mathbf{M}_{V\to W} = \left[\begin{array}{c|c} \mathbf{i}_V & 0 \\ \mathbf{j}_V & 0 \\ \mathbf{k}_V & 0 \\ \hline \mathbf{t}_V & 1 \end{array}\right].$$

When rendering a triangle mesh, its vertices are transformed first from model space to world space, and then from world space to view space. To perform this latter transformation, we need the world-to-view matrix, which is the inverse of the view-to-world matrix. This matrix is sometimes called the *view matrix*:

$$\mathbf{M}_{W\to V} = \mathbf{M}_{V\to W}^{-1} = \mathbf{M}_{\text{view}}.$$

图 11.31. 左手和右手相机空间轴。

这里要小心。相机的矩阵相对于场景中物体的矩阵是反转的，这是新游戏开发者容易混淆和犯错的地方。

在渲染特定网格实例之前，世界空间到视图矩阵通常会与模型空间到世界空间矩阵连接起来。这个组合矩阵在 OpenGL 中被称为模型-视图矩阵。我们预先计算了这个矩阵，以便渲染引擎在将顶点从模型空间转换到视图空间时只需进行一次矩阵乘法：

MM → V = MM → WMW → V = M 模型视图。

11.1.4.2 预测

为了将 3D 场景渲染到 2D 图像平面上，我们使用一种称为投影的特殊变换。透视投影是计算机图形学中最常见的投影，因为它模拟了典型相机生成的图像。在这种投影中，物体距离相机越远，看起来就越小——这种效果称为透视缩短。

一些游戏也使用长度保持正交投影，主要用于渲染 3D 模型或游戏关卡的平面视图（例如正面、侧面和俯视图），以便进行编辑，以及将 2D 图形叠加到屏幕上，用于平视显示器等。图 11.32 展示了使用这两种投影类型渲染立方体时的外观。

Figure 11.31. Left- and right-handed camera-space axes.

Be careful here. The fact that the camera's matrix is inverted relative to the matrices of the objects in the scene is a common point of confusion and bugs among new game developers.

The world-to-view matrix is often concatenated to the model-to-world matrix prior to rendering a particular mesh instance. This combined matrix is called the *model-view matrix* in OpenGL. We precalculate this matrix so that the rendering engine only needs to do a single matrix multiply when transforming vertices from model space into view space:

$$\mathbf{M}_{M\to V} = \mathbf{M}_{M\to W}\mathbf{M}_{W\to V} = \mathbf{M}_{\text{modelview}}.$$

11.1.4.2 Projections

In order to render a 3D scene onto a 2D image plane, we use a special kind of transformation known as a *projection*. The *perspective* projection is the most common projection in computer graphics, because it mimics the kinds of images produced by a typical camera. With this projection, objects appear smaller the farther away they are from the camera—an effect known as *perspective foreshortening*.

The length-preserving *orthographic* projection is also used by some games, primarily for rendering *plan views* (e.g., front, side and top) of 3D models or game levels for editing purposes, and for overlaying 2D graphics onto the screen for heads-up displays and the like. Figure 11.32 illustrates how a cube would look when rendered with these two types of projections.

图 11.32.使用透视投影（左）和正交投影（右）渲染的立方体。

11.1.4.3 视景体和视锥体

相机能够“看到”的空间区域称为视景体。视景体由六个平面定义。近平面对应于虚拟图像传感表面。四个侧平面对应于虚拟屏幕的边缘。远平面用于优化渲染，以确保不会绘制非常远的物体。它还规定了深度缓冲区中存储的深度上限（参见第 11.1.4.8 节）。

使用透视投影渲染场景时，视景体的形状是一个截头金字塔，称为平截头体。使用正交投影渲染场景时，视景体是一个长方体。透视投影和正交投影的视景体分别如图 11.33 和图 11.34 所示。

视场体的六个平面可以用六个四元素向量 (nix , n iy , n iz , di) 简洁地表示，其中 n = (nx , ny , nz) 是平面法线，d 是其与原点的垂直距离。如果我们更喜欢

图 11.33.体积（切片）的透视图。

Figure 11.32. A cube rendered using a perspective projection (on the left) and an orthographic projection (on the right).

11.1.4.3 The View Volume and the Frustum

The region of space that the camera can “see” is known as the view volume. A view volume is defined by six planes. The *near plane* corresponds to the virtual image-sensing surface. The four side planes correspond to the edges of the virtual screen. The *far plane* is used as a rendering optimization to ensure that extremely distant objects are not drawn. It also provides an upper limit for the depths that will be stored in the depth buffer (see Section 11.1.4.8).

When rendering the scene with a *perspective* projection, the shape of the view volume is a truncated pyramid known as a *frustum*. When using an orthographic projection, the view volume is a rectangular prism. Perspective and orthographic view volumes are illustrated in Figure 11.33 and Figure 11.34, respectively.

The six planes of the view volume can be represented compactly using six four-element vectors $(n_{ix}, n_{iy}, n_{iz}, d_i)$, where $\mathbf{n} = (n_x, n_y, n_z)$ is the plane normal and d is its perpendicular distance from the origin. If we prefer the

Figure 11.33. A perspective view volume (frustum).

图 11.34.正交视图体积。

除了点法平面表示法之外，我们还可以用六对向量（Q i , ni）来描述平面，其中 Q 是平面上的任意点，n 是平面法线。（在这两种情况下，i 都是代表六个平面的索引。）

11.1.4.4 投影和齐次裁剪空间

透视投影和正交投影都会将视图空间中的点变换到称为齐次裁剪空间的坐标空间中。这个三维空间实际上只是视图空间的扭曲版本。裁剪空间的目的是将相机空间的视图体转换为一个规范的视图体，该视图体既独立于用于将三维场景转换为二维屏幕空间的投影类型，也独立于渲染场景的屏幕的分辨率和宽高比。

在裁剪空间中，标准视体积是一个长方体，沿 x 轴和 y 轴从 −1 延伸到 +1。沿 z 轴，视体积从 −1 延伸到 +1（OpenGL）或从 0 延伸到 1（DirectX）。我们将此坐标系称为“裁剪空间”，因为视体积平面与轴对齐，从而可以方便地将三角形裁剪到此空间中的视体积（即使使用透视投影）。OpenGL 的标准裁剪空间视体积如图 11.35 所示。请注意，裁剪空间的 z 轴向上，y 轴向右。换句话说，齐次裁剪空间通常是左手系的。这里使用左手系约定，因为它会使 z 值的增加对应于屏幕深度的增加，照常 y 向上增加，x 向右增加。

Figure 11.34. An orthographic view volume.

point-normal plane representation, we can also describe the planes with six pairs of vectors $(\mathbf{Q}_i, \mathbf{n}_i)$, where $\mathbf{Q}$ is the arbitrary point on the plane and $\mathbf{n}$ is the plane normal. (In both cases, i is an index representing the six planes.)

11.1.4.4 Projection and Homogeneous Clip Space

Both perspective and orthographic projections transform points in view space into a coordinate space called *homogeneous clip space*. This three-dimensional space is really just a warped version of view space. The purpose of clip space is to convert the camera-space view volume into a canonical view volume that is independent both of the kind of *projection* used to convert the 3D scene into 2D screen space, and of the *resolution* and *aspect ratio* of the screen onto which the scene is going to be rendered.

In clip space, the canonical view volume is a rectangular prism extending from -1 to $+1$ along the x- and y-axes. Along the z-axis, the view volume extends either from -1 to $+1$ (OpenGL) or from 0 to 1 (DirectX). We call this coordinate system “clip space” because the view volume planes are axis-aligned, making it convenient to *clip* triangles to the view volume in this space (even when a perspective projection is being used). The canonical clip-space view volume for OpenGL is depicted in Figure 11.35. Notice that the z-axis of clip space goes into the screen, with y up and x to the right. In other words, homogeneous clip space is usually *left-handed*. A left-handed convention is used here because it causes increasing z values to correspond to increasing *depth* into the screen, with y increasing up and x increasing to the right as usual.

图 11.35. 齐次裁剪空间中的规范视图体积。

透视投影

[32] 的 4.5.1 节对透视投影进行了详尽的解释，因此我们在此不再赘述。下面我们将简单介绍透视投影矩阵 MV → H。（下标 V → H 表示该矩阵将顶点从视图空间变换到齐次裁剪空间。）如果我们将视图空间视为右手坐标系，则近平面与 z 轴的交点为 z = − n，远平面与 z = − f。虚拟屏幕的左边缘、右边缘、下边缘和上边缘分别位于近平面的 x = l、x = r、y = b 和 y = t 处。（通常，虚拟屏幕以相机空间的 z 轴为中心，在这种情况下 l = − r 和 b = − t，但并非总是如此。）使用这些定义，OpenGL 的透视投影矩阵如下所示：

$$\mathbf{M}_{V\to H} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}.$$

DirectX 定义了剪辑空间视图体积的 z 轴范围

Figure II.35. The canonical view volume in homogeneous clip space.

Perspective Projection

An excellent explanation of perspective projection is given in Section 4.5.1 of [32], so we won't repeat it here. Instead, we'll simply present the perspective projection matrix $\mathbf{M}_{V\to H}$ below. (The subscript $V \to H$ indicates that this matrix transforms vertices from view space into homogeneous clip space.) If we take view space to be right-handed, then the near plane intersects the z-axis at $z = -n$, and the far plane intersects it at $z = -f$. The virtual screen's left, right, bottom, and top edges lie at $x = l$, $x = r$, $y = b$ and $y = t$ on the near plane, respectively. (Typically the virtual screen is centered on the camera-space z-axis, in which case $l = -r$ and $b = -t$, but this isn't always the case.) Using these definitions, the perspective projection matrix for OpenGL is as follows:

$$\mathbf{M}_{V\to H} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}.$$

DirectX defines the z-axis extents of the clip-space view volume to lie in

范围是 [0, 1]，而不是像 OpenGL 那样在范围 [− 1, 1] 内。我们可以轻松地调整透视投影矩阵以符合 DirectX 的约定，如下所示：

$$(\mathbf{M}_{V\to H})_{\text{DirectX}} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{nf}{f-n}\right) & 0 \end{bmatrix}.$$

除以 z

透视投影会导致每个顶点的 x 和 y 坐标除以其 z 坐标。这就是透视缩短的原因。为了理解为什么会发生这种情况，请考虑将以四元素齐次坐标表示的视图空间点 p V 乘以 OpenGL 透视投影矩阵：

$$\begin{aligned} \mathbf{p}_H &= \mathbf{p}_V \mathbf{M}_{V\to H} \\ &= \begin{bmatrix} p_{Vx} & p_{Vy} & p_{Vz} & 1 \end{bmatrix} \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}. \end{aligned}$$

乘法的结果形式为

$$\mathbf{p}_H = \begin{matrix} a & b & c & -p_{Vz} \end{matrix}. \tag{11.1}$$

当我们将任何齐次向量转换为三维坐标时，x、y 和 z 分量都会除以 w 分量：

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \equiv \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}.$$

因此，将方程 (11.1) 除以齐次 w 分量（实际上就是负视图空间 z 坐标 − p Vz ），我们得到：

$$\begin{aligned} \mathbf{p}_H &= \begin{bmatrix} \frac{a}{-p_{Vz}} & \frac{b}{-p_{Vz}} & \frac{c}{-p_{Vz}} \end{bmatrix} \\ &= \begin{bmatrix} p_{Hx} & p_{Hy} & p_{Hz} \end{bmatrix}. \end{aligned}$$

the range $[0, 1]$ rather than in the range $[-1, 1]$ as OpenGL does. We can easily adjust the perspective projection matrix to account for DirectX's conventions as follows:

$$(\mathbf{M}_{V\to H})_{\text{DirectX}} = \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{nf}{f-n}\right) & 0 \end{bmatrix}.$$

Division by z

Perspective projection results in each vertex's x- and y-coordinates being divided by its z-coordinate. This is what produces perspective foreshortening. To understand why this happens, consider multiplying a view-space point $\mathbf{p}_V$ expressed in four-element homogeneous coordinates by the OpenGL perspective projection matrix:

$$\begin{aligned} \mathbf{p}_H &= \mathbf{p}_V \mathbf{M}_{V\to H} \\ &= \begin{bmatrix} p_{Vx} & p_{Vy} & p_{Vz} & 1 \end{bmatrix} \begin{bmatrix} \left(\frac{2n}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2n}{t-b}\right) & 0 & 0 \\ \left(\frac{r+l}{r-l}\right) & \left(\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & -1 \\ 0 & 0 & \left(-\frac{2nf}{f-n}\right) & 0 \end{bmatrix}. \end{aligned}$$

The result of this multiplication takes the form

$$\mathbf{p}_H = \begin{bmatrix} a & b & c & -p_{Vz} \end{bmatrix}. \tag{11.1}$$

When we convert any homogeneous vector into three-dimensional coordinates, the x-, y- and z-components are divided by the w-component:

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \equiv \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}.$$

So, after dividing Equation (11.1) by the homogeneous w-component, which is really just the negative view-space z-coordinate $-p_{Vz}$, we have:

$$\begin{aligned} \mathbf{p}_H &= \begin{bmatrix} \frac{a}{-p_{Vz}} & \frac{b}{-p_{Vz}} & \frac{c}{-p_{Vz}} \end{bmatrix} \\ &= \begin{bmatrix} p_{Hx} & p_{Hy} & p_{Hz} \end{bmatrix}. \end{aligned}$$

因此，齐次剪辑空间坐标已被视图空间 z 坐标除，这就是导致透视缩短的原因。

透视校正顶点属性插值

在 11.1.2.4 节中，我们学习了顶点属性的插值，以便在三角形内部确定合适的值。属性插值在屏幕空间中执行。我们遍历屏幕的每个像素，并尝试确定每个属性在三角形表面相应位置的值。在使用透视投影渲染场景时，我们必须非常谨慎地执行此操作，以考虑透视缩短效应。这称为透视校正属性插值。

透视校正插值的推导超出了我们的范围，但可以说，我们必须将插值的属性值除以每个顶点对应的 z 坐标（深度）。对于任意一对顶点属性 A 1 和 A 2，我们可以将插值属性值写成它们之间距离的百分比 t，如下所示：

$$\frac{A}{p_z} = (1-t)\left(\frac{A_1}{p_{1z}}\right) + t\left(\frac{A_2}{p_{2z}}\right) = \mathrm{LERP}\left(\frac{A_1}{p_{1z}}, \frac{A_2}{p_{2z}}, t\right).$$

请参阅[32]，了解透视校正属性插值背后的数学推导。

正交投影

正交投影由以下矩阵执行：

$$(\mathbf{M}_{V\to H})_{\mathrm{ortho}} = \begin{bmatrix} \left(\frac{2}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2}{t-b}\right) & 0 & 0 \\ 0 & 0 & \left(-\frac{2}{f-n}\right) & 0 \\ \left(-\frac{r+l}{r-l}\right) & \left(-\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & 1 \end{bmatrix}.$$

这只是一个常见的缩放和平移矩阵。（左上角的 3×3 矩阵包含一个对角线非均匀缩放矩阵，下一行包含平移矩阵。）由于视空间和裁剪空间中的视空间体都是长方体，我们只需要缩放和平移顶点就可以从一个空间转换到另一个空间。

Thus, the homogeneous clip-space coordinates have been divided by the view-space z-coordinate, which is what causes perspective foreshortening.

Perspective-Correct Vertex Attribute Interpolation

In Section 11.1.2.4, we learned that vertex attributes are interpolated in order to determine appropriate values for them within the interior of a triangle. Attribute interpolation is performed in *screen space*. We iterate over each pixel of the screen and attempt to determine the value of each attribute at the corresponding location *on the surface of the triangle*. When rendering a scene with a perspective projection, we must do this very carefully so as to account for perspective foreshortening. This is known as *perspective-correct* attribute interpolation.

A derivation of perspective-correct interpolation is beyond our scope, but suffice it to say that we must divide our interpolated attribute values by the corresponding z-coordinates (depths) at each vertex. For any pair of vertex attributes A_1 and A_2, we can write the interpolated attribute at a percentage t of the distance between them as follows:

$$\frac{A}{p_z} = (1-t)\left(\frac{A_1}{p_{1z}}\right) + t\left(\frac{A_2}{p_{2z}}\right) = \mathrm{LERP}\left(\frac{A_1}{p_{1z}}, \frac{A_2}{p_{2z}}, t\right).$$

Refer to [32] for an excellent derivation of the math behind perspective-correct attribute interpolation.

Orthographic Projection

An orthographic projection is performed by the following matrix:

$$(\mathbf{M}_{V\to H})_{\mathrm{ortho}} = \begin{bmatrix} \left(\frac{2}{r-l}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{2}{t-b}\right) & 0 & 0 \\ 0 & 0 & \left(-\frac{2}{f-n}\right) & 0 \\ \left(-\frac{r+l}{r-l}\right) & \left(-\frac{t+b}{t-b}\right) & \left(-\frac{f+n}{f-n}\right) & 1 \end{bmatrix}.$$

This is just an everyday scale-and-translate matrix. (The upper-left 3×3 contains a diagonal nonuniform scaling matrix, and the lower row contains the translation.) Since the view volume is a rectangular prism in both view space and clip space, we need only scale and translate our vertices to convert from one space to the other.

图 11.36 最常见的两种屏幕空间宽高比是 4:3 和 16:9。

11.1.4.5 屏幕空间和宽高比

屏幕空间是一个二维坐标系，其坐标轴以屏幕像素为单位。x 轴通常指向右方，原点位于屏幕左上角，y 轴指向下方。（y 轴反转的原因是 CRT 显示器从上到下扫描屏幕。）屏幕宽度与屏幕高度之比称为宽高比。最常见的宽高比是 4:3（传统电视屏幕的宽高比）和 16:9（电影屏幕或高清电视的宽高比）。这些宽高比如图 11.36 所示。

我们可以通过简单地绘制三角形的 (x, y) 坐标并忽略 z 来渲染在齐次裁剪空间中表示的三角形。但在此之前，我们需要缩放和平移裁剪空间坐标，使它们位于屏幕空间中，而不是标准化的单位正方形内。这种缩放和平移操作称为屏幕映射。

11.1.4.6 帧缓冲区

最终渲染的图像存储在位图颜色缓冲区（称为帧缓冲区）中。像素颜色通常以 RGBA8888 格式存储，但大多数显卡也支持其他帧缓冲区格式。一些常见的格式包括 RGB565、RGB5551 以及一种或多种调色板模式。

显示硬件（CRT、平板显示器、高清电视等）以 60 Hz（北美和日本使用的 NTSC 电视）或 50 Hz（欧洲和世界其他许多地方使用的 PAL/SECAM 电视）的周期速率读取帧缓冲区的内容。渲染引擎通常维护至少两个帧缓冲区。当显示硬件扫描其中一个帧缓冲区时，渲染引擎可以更新另一个帧缓冲区。这称为双缓冲。通过在垂直消隐间隔（CRT 的电子枪重置到屏幕左上角的时间段）期间交换或“翻转”两个缓冲区，双缓冲可确保显示硬件始终扫描完整的帧缓冲区。这避免了不和谐的效果

Figure 11.36. The two most prevalent screen-space aspect ratios are 4:3 and 16:9.

11.1.4.5 Screen Space and Aspect Ratios

Screen space is a two-dimensional coordinate system whose axes are measured in terms of screen pixels. The x-axis typically points to the right, with the origin at the top-left corner of the screen and y pointing down. (The reason for the inverted y-axis is that CRT monitors scan the screen from top to bottom.) The ratio of screen width to screen height is known as the *aspect ratio*. The most common aspect ratios are 4:3 (the aspect ratio of a traditional television screen) and 16:9 (the aspect ratio of a movie screen or HDTV). These aspect ratios are illustrated in Figure 11.36.

We can render triangles expressed in homogeneous clip space by simply drawing their (x, y) coordinates and ignoring z. But before we do, we scale and shift the clip-space coordinates so that they lie in screen space rather than within the normalized unit square. This scale-and-shift operation is known as *screen mapping*.

11.1.4.6 The Frame Buffer

The final rendered image is stored in a bitmapped color buffer known as the *frame buffer*. Pixel colors are usually stored in RGBA8888 format, although other frame buffer formats are supported by most graphics cards as well. Some common formats include RGB565, RGB5551, and one or more paletted modes.

The display hardware (CRT, flat-screen monitor, HDTV, etc.) reads the contents of the frame buffer at a periodic rate of 60 Hz for NTSC televisions used in North America and Japan, or 50 Hz for PAL/SECAM televisions used in Europe and many other places in the world. Rendering engines typically maintain at least two frame buffers. While one is being scanned by the display hardware, the other one can be updated by the rendering engine. This is known as *double buffering*. By swapping or “flipping” the two buffers during the *vertical blanking interval* (the period during which the CRT’s electron gun is being reset to the top-left corner of the screen), double buffering ensures that the display hardware always scans the complete frame buffer. This avoids a jarring effect

这种现象称为撕裂，屏幕上方显示新渲染的图像，而下方显示前一帧图像的残余。

有些引擎会使用三个帧缓冲区——这种技术被巧妙地称为三重缓冲。这样做的目的是，即使前一帧仍在由显示硬件扫描，渲染引擎也可以开始处理下一帧。例如，当引擎完成缓冲区 B 的绘制时，硬件可能仍在扫描缓冲区 A。有了三重缓冲，渲染引擎就可以继续将新帧渲染到缓冲区 C 中，而不是在等待显示硬件完成扫描缓冲区 A 时处于空闲状态。

渲染目标

渲染引擎用来绘制图形的任何缓冲区都称为渲染目标 (render target)。正如本章后面将要介绍的，除了帧缓冲区之外，渲染引擎还会使用各种其他屏幕外渲染目标。这些目标包括深度缓冲区、模板缓冲区以及用于存储中间渲染结果的各种其他缓冲区。

11.1.4.7 三角形光栅化和碎片

为了在屏幕上生成三角形的图像，我们需要填充与其重叠的像素。这个过程称为光栅化。在光栅化过程中，三角形的表面被分解成称为碎片的碎片，每个碎片代表三角形表面的一小块区域，对应屏幕上的单个像素。（在多重采样抗锯齿的情况下，一个碎片对应一个像素的一部分——见下文。）

片段就像训练中的像素。在写入帧缓冲区之前，它必须通过一系列测试（下文将更详细地描述）。如果任何测试失败，它都会被丢弃。通过测试的片段会被着色（即确定其颜色），并且片段颜色要么写入帧缓冲区，要么与已有的像素颜色混合。图 11.37 展示了片段如何变成像素。

11.1.4.8 遮挡和深度缓冲区

当渲染两个在屏幕空间中相互重叠的三角形时，我们需要某种方式来确保靠近相机的三角形显示在最上面。我们可以通过始终按从后到前的顺序渲染三角形来实现这一点（即所谓的画家算法）。然而，如图 11.38 所示，如果三角形彼此相交，这种方法就行不通了。

known as *tearing*, in which the upper portion of the screen displays the newly rendered image while the bottom shows the remnants of the previous frame's image.

Some engines make use of three frame buffers—a technique aptly known as *triple buffering*. This is done so that the rendering engine can start work on the next frame, even while the previous frame is still being scanned by the display hardware. For example, the hardware might still be scanning buffer A when the engine finishes drawing buffer B. With triple buffering, it can proceed to render a new frame into buffer C, rather than idling while it waits for the display hardware to finish scanning buffer A.

Render Targets

Any buffer into which the rendering engine draws graphics is known as a *render target*. As we'll see later in this chapter, rendering engines make use of all sorts of other off-screen render targets in addition to the frame buffers. These include the depth buffer, the stencil buffer and various other buffers used for storing intermediate rendering results.

11.1.4.7 Triangle Rasterization and Fragments

To produce an image of a triangle on-screen, we need to fill in the pixels it overlaps. This process is known as rasterization. During rasterization, the triangle's surface is broken into pieces called *fragments*, each one representing a small region of the triangle's surface that corresponds to a single pixel on the screen. (In the case of multisample antialiasing, a fragment corresponds to a *portion* of a pixel—see below.)

A fragment is like a pixel in training. Before it is written into the frame buffer, it must pass a number of tests (described in more depth below). If it fails any of these tests, it will be discarded. Fragments that pass the tests are shaded (i.e., their colors are determined), and the fragment color is either written into the frame buffer or blended with the pixel color that's already there. Figure 11.37 illustrates how a fragment becomes a pixel.

11.1.4.8 Occlusion and the Depth Buffer

When rendering two triangles that overlap each other in screen space, we need some way of ensuring that the triangle that is closer to the camera will appear on top. We could accomplish this by always rendering our triangles in back-to-front order (the so-called painter's algorithm). However, as shown in Figure 11.38, this doesn't work if the triangles are intersecting one another.

图 11.37。片段是对应于屏幕上像素的三角形小区域。它穿过渲染管线，要么被丢弃，要么将其颜色写入帧缓冲区。

为了正确实现三角形遮挡，无论三角形的渲染顺序如何，渲染引擎都会使用一种称为深度缓冲或 z 缓冲的技术。深度缓冲区是一个全屏缓冲区，通常包含帧缓冲区中每个像素的 24 位整数或（较少见）浮点深度信息。（深度缓冲区通常以每像素 32 位的格式存储，每个像素的 32 位四字中包含一个 24 位深度值和一个 8 位模板值。）每个片段都有一个 z 坐标，用于测量其“进入”屏幕的深度。（片段的深度是通过对三角形顶点的深度进行插值得到的。）当片段的颜色写入帧缓冲区时，其深度会存储在深度缓冲区的相应像素中。当另一个片段（来自另一个三角形）绘制到同一像素时，引擎会将新片段的深度与深度缓冲区中已有的深度进行比较。如果片段距离摄像机较近（即

图 11.38。画家算法按从后到前的顺序渲染三角形，以产生正确的三角形遮挡。然而，当三角形相互交叉时，该算法就会失效。

Figure 11.37. A fragment is a small region of a triangle corresponding to a pixel on the screen. It passes through the rendering pipeline and is either discarded or its color is written into the frame buffer.

To implement triangle occlusion properly, independent of the order in which the triangles are rendered, rendering engines use a technique known as *depth buffering* or *z-buffering*. The depth buffer is a full-screen buffer that typically contains 24-bit integer or (more rarely) floating-point depth information for each pixel in the frame buffer. (The depth buffer is usually stored in a 32-bits-per-pixel format, with a 24-bit depth value and an 8-bit stencil value packed into each pixel's 32-bit quadword.) Every fragment has a z-coordinate that measures its depth "into" the screen. (The depth of a fragment is found by interpolating the depths of the triangle's vertices.) When a fragment's color is written into the frame buffer, its depth is stored into the corresponding pixel of the depth buffer. When another fragment (from another triangle) is drawn into the same pixel, the engine compares the new fragment's depth to the depth already present in the depth buffer. If the fragment is closer to the camera (i.e.,

Figure 11.38. The painter's algorithm renders triangles in a back-to-front order to produce proper triangle occlusion. However, the algorithm breaks down when triangles intersect one another.

如果深度较小，则覆盖帧缓冲区中的像素。否则，该片段将被丢弃。

z-Fighting 和 w-Buffer

当渲染彼此非常接近的平行表面时，渲染引擎能够区分两个平面的深度至关重要。如果我们的深度缓冲区具有无限精度，这绝不会成为问题。然而，真正的深度缓冲区精度有限，因此当两个平面足够接近时，它们的深度值可能会合并为一个离散值。当这种情况发生时，较远平面的像素会开始“穿透”较近平面，从而导致一种称为“z 冲突”的噪声效应。

为了将整个场景的 z 冲突降至最低，我们希望无论渲染靠近相机的表面还是远离相机的表面，都能获得相同的精度。然而，z 缓冲并非如此。由于被视空间 z 坐标除以 1/z ，裁剪空间 z 深度（p H z）的精度并非均匀分布在从近平面到远平面的整个范围内。由于 1/z 曲线的形状，深度缓冲区的大部分精度集中在相机附近。

图 11.39 中函数 p H z = 1/ p V z 的绘图演示了这种效果。在靠近相机的地方，观察空间中两个平面之间的距离 Δ p V z 会转换为裁剪空间中相当大的增量 Δ p H z 。但远离相机的地方，同样的距离会转换为裁剪空间中一个微小的增量。结果就是 z 轴冲突，并且随着物体距离相机越来越远，这种冲突会变得越来越普遍。

为了解决这个问题，我们希望在深度缓冲区中存储视图空间 z 坐标（pVz），而不是裁剪空间 z 坐标（pHz）。视图空间 z 坐标随与相机的距离线性变化，因此使用它们作为深度测量可以在整个深度范围内实现均匀的精度。

图 11.39 函数 1/ p V z 的图，显示了大部分精度如何靠近相机。

if it has a smaller depth), it overwrites the pixel in the frame buffer. Otherwise the fragment is discarded.

z-Fighting and the w-Buffer

When rendering parallel surfaces that are very close to one another, it's important that the rendering engine can distinguish between the depths of the two planes. If our depth buffer had infinite precision, this would never be a problem. Unfortunately, a real depth buffer only has limited precision, so the depth values of two planes can collapse into a single discrete value when the planes are close enough together. When this happens, the more-distant plane's pixels start to "poke through" the nearer plane, resulting in a noisy effect known as *z-fighting*.

To reduce z-fighting to a minimum across the entire scene, we would like to have equal precision whether we're rendering surfaces that are close to the camera or far away. However, with z-buffering this is not the case. The precision of clip-space z-depths (p_{H_z}) are not evenly distributed across the entire range from the near plane to the far plane, because of the division by the view-space z-coordinate. Because of the shape of the $1/z$ curve, most of the depth buffer's precision is concentrated near the camera.

The plot of the function $p_{H_z} = 1/p_{V_z}$ shown in Figure 11.39 demonstrates this effect. Near the camera, the distance between two planes in view space Δp_{V_z} gets transformed into a reasonably large delta in clip space Δp_{H_z}. But far from the camera, this same separation gets transformed into a tiny delta in clip space. The result is z-fighting, and it becomes rapidly more prevalent as objects get farther away from the camera.

To circumvent this problem, we would like to store *view-space* z-coordinates (p_{V_z}) in the depth buffer instead of clip-space z-coordinates (p_{H_z}). View-space z-coordinates vary linearly with the distance from the camera, so using them as our depth measure achieves uniform precision across the entire depth range.

Figure 11.39. A plot of the function $1/p_{V_z}$, showing how most of the precision lies close to the camera.

这种技术被称为 w 缓冲，因为观察空间的 z 坐标恰好出现在齐次裁剪空间坐标的 w 分量中。（回想一下公式 (11.1)，p H w = − p V z。）这里的术语可能非常令人困惑。z 和 w 缓冲区存储的是裁剪空间中的坐标。但就观察空间坐标而言，z 缓冲区存储的是 1/ z（即 1/ p V z），而 w 缓冲区存储的是 z（即 p V z）！

这里需要注意的是，w 缓冲方法比基于 z 缓冲的方法开销更大一些。这是因为使用 w 缓冲时，我们无法直接线性插值深度。深度必须在插值之前进行反转，然后在存储到 w 缓冲区之前再次反转。

11.2 渲染管线

我们已经完成了对三角形光栅化主要理论和实践基础的快速了解，现在让我们来看看它的典型实现方式。在实时游戏渲染引擎中，11.1 节中描述的高级渲染步骤是使用一种称为“流水线”的软硬件架构实现的。流水线只是一系列有序的计算阶段，每个阶段都有特定的用途，对输入数据流进行操作并生成输出数据流。

流水线的每个阶段通常可以独立于其他阶段运行。因此，流水线架构的最大优势之一在于它非常适合并行化。当第一阶段处理一个数据元素时，第二阶段可以处理第一阶段先前生成的结果，依此类推。

并行化也可以在流水线的单个阶段内实现。例如，如果某个阶段的计算硬件在芯片上复制了 N 个，那么该阶段就可以并行处理 N 个数据元素。图 11.40 展示了一个并行化的流水线。理想情况下，各个阶段并行运行（大多数情况下），并且某些阶段还能够同时处理多个数据项。

流水线的吞吐量衡量的是每秒处理的数据项数量。流水线的延迟衡量的是单个数据元素流经整个流水线所需的时间。单个阶段的延迟衡量的是该阶段处理单个数据项所需的时间。流水线中最慢的阶段决定了整个流水线的吞吐量。它也会影响整个流水线的平均延迟。因此，在设计渲染流水线时，我们会尝试最小化和平衡整个流水线的延迟，并消除瓶颈。在一个良好的

This technique is called *w-buffering*, because the view-space z-coordinate conveniently appears in the w-component of our homogeneous clip-space coordinates. (Recall from Equation (11.1) that $p_{H_w} = -p_{V_z}$.)

The terminology can be very confusing here. The z- and w-buffers store coordinates that are expressed in *clip space*. But in terms of *view-space* coordinates, the z-buffer stores $1/z$ (i.e., $1/p_{V_z}$) while the w-buffer stores z (i.e., p_{V_z})!

We should note here that the w-buffering approach is a bit more expensive than its z-based counterpart. This is because with w-buffering, we cannot linearly interpolate depths directly. Depths must be inverted prior to interpolation and then re-inverted prior to being stored in the w-buffer.

11.2 The Rendering Pipeline

Now that we've completed our whirlwind tour of the major theoretical and practical underpinnings of triangle rasterization, let's turn our attention to how it is typically implemented. In real-time game rendering engines, the high-level rendering steps described in Section 11.1 are implemented using a software/hardware architecture known as a *pipeline*. A pipeline is just an ordered chain of computational stages, each with a specific purpose, operating on a stream of input data items and producing a stream of output data.

Each stage of a pipeline can typically operate independently of the other stages. Hence, one of the biggest advantages of a pipelined architecture is that it lends itself extremely well to parallelization. While the first stage is chewing on one data element, the second stage can be processing the results previously produced by the first stage, and so on down the chain.

Parallelization can also be achieved within an individual stage of the pipeline. For example, if the computing hardware for a particular stage is duplicated N times on the die, N data elements can be processed in parallel by that stage. A parallelized pipeline is shown in Figure 11.40. Ideally the stages operate in parallel (most of the time), and certain stages are capable of operating on multiple data items simultaneously as well.

The *throughput* of a pipeline measures how many data items are processed per second overall. The pipeline's *latency* measures the amount of time it takes for a single data element to make it through the entire pipeline. The latency of an individual stage measures how long that stage takes to process a single item. The slowest stage of a pipeline dictates the throughput of the entire pipeline. It also has an impact on the average latency of the pipeline as a whole. Therefore, when designing a rendering pipeline, we attempt to minimize and balance latency across the entire pipeline and eliminate bottlenecks. In a well-

图 11.40 并行化流水线。所有阶段均并行运行，部分阶段甚至能够同时处理多个数据项。

设计的管道，所有阶段同时运行，并且没有一个阶段会长时间空闲以等待另一个阶段空闲。

11.2.1 渲染管线概述

一些图形学教材将渲染管线粗略地划分为三个阶段。本书将进一步延伸这一管线，涵盖用于创建最终由游戏引擎渲染的场景的离线工具。管线中的高级阶段包括：

- 工具阶段（离线）。定义几何形状和表面特性（材料）。
- 资产调节阶段（离线）。几何和材料数据由资产调节管道（ACP）处理为引擎就绪格式。
- 应用阶段（CPU）。识别潜在可见的网格实例，并将其连同材质一起提交给图形硬件进行渲染。
- 几何处理阶段（GPU）。顶点经过变换、点亮并投影到均匀裁剪空间。三角形由可选的几何着色器处理，然后裁剪到截锥体上。
- 光栅化阶段（GPU）。三角形被转换成经过着色的碎片，经过各种测试（z 测试、alpha 测试、模板测试等），最后混合到帧缓冲区中。

Figure 11.40. A parallelized pipeline. The stages all operate in parallel, and some stages are capable of operating on multiple data items simultaneously as well.

designed pipeline, all the stages operate simultaneously, and no stage is ever idle for very long waiting for another stage to become free.

11.2.1 Overview of the Rendering Pipeline

Some graphics texts divide the rendering pipeline into three coarse-grained stages. In this book, we'll extend this pipeline back even further, to encompass the offline tools used to create the scenes that are ultimately rendered by the game engine. The high-level stages in our pipeline are:

- *Tools stage (offline)*. Geometry and surface properties (materials) are defined.
- *Asset conditioning stage (offline)*. The geometry and material data are processed by the asset conditioning pipeline (ACP) into an engine-ready format.
- *Application stage (CPU)*. Potentially visible mesh instances are identified and submitted to the graphics hardware along with their materials for rendering.
- *Geometry processing stage (GPU)*. Vertices are transformed and lit and projected into homogeneous clip space. Triangles are processed by the optional geometry shader and then clipped to the frustum.
- *Rasterization stage (GPU)*. Triangles are converted into fragments that are shaded, passed through various tests (*z*-test, alpha test, stencil test, etc.) and finally blended into the frame buffer.

图 11.41。几何数据的格式在经过渲染管道的各个阶段时会发生根本性的变化。

11.2.1.1 渲染管道如何转换数据

值得注意的是，几何数据的格式在通过渲染管线时会发生变化。工具和资源调节阶段处理网格和材质。应用阶段处理网格实例和子网格，每个实例都与一种材质相关联。在几何阶段，每个子网格被分解成单独的顶点，这些顶点大部分是并行处理的。在此阶段结束时，三角形会根据完全变换和着色后的顶点重建。在光栅化阶段，每个三角形被分解成碎片，这些碎片要么被丢弃，要么最终作为颜色写入帧缓冲区。该过程如图 11.41 所示。

11.2.1.2 管道的实施

渲染管线的前两个阶段是离线实现的，通常由 Windows 或 Linux 系统执行。应用程序阶段通常在一个或多个 CPU 核心上运行，而几何和光栅化阶段通常由图形处理单元 (GPU) 执行。在接下来的章节中，我们将探讨每个阶段的具体实现细节。

11.2.2 工具阶段

在工具阶段，网格由 3D 建模人员在数字内容创建 (DCC) 应用程序（如 Maya、3ds Max、Lightwave、Softimage/XSI）中创建，

Figure 11.41. The format of geometric data changes radically as it passes through the various stages of the rendering pipeline.

11.2.1.1 How the Rendering Pipeline Transforms Data

It's interesting to note how the format of geometry data changes as it passes through the rendering pipeline. The tools and asset conditioning stages deal with meshes and materials. The application stage deals in terms of mesh instances and submeshes, each of which is associated with a single material. During the geometry stage, each submesh is broken down into individual vertices, which are processed largely in parallel. At the conclusion of this stage, the triangles are reconstructed from the fully transformed and shaded vertices. In the rasterization stage, each triangle is broken into fragments, and these fragments are either discarded, or they are eventually written into the frame buffer as colors. This process is illustrated in Figure 11.41.

11.2.1.2 Implementation of the Pipeline

The first two stages of the rendering pipeline are implemented offline, usually executed by a Windows or Linux machine. The application stage is typically run on one or more CPU cores, whereas the geometry and rasterization stages are usually executed by the graphics processing unit (GPU). In the following sections, we'll explore some of the details of how each of these stages is implemented.

11.2.2 The Tools Stage

In the tools stage, meshes are authored by 3D modelers in a *digital content creation* (DCC) application like Maya, 3ds Max, Lightwave, Softimage/XSI,

SketchUp 等。可以使用任何方便的表面描述（NURBS、四边形、三角形等）来定义模型。但是，在由管道的运行时部分渲染之前，它们总是被细分为三角形。

网格的顶点也可以进行蒙皮。这涉及将每个顶点与铰接式骨架结构中的一个或多个关节关联，并设置权重来描述每个关节对顶点的相对影响。动画系统使用蒙皮信息和骨架来驱动模型的运动——更多详情请参阅第 12 章。

材质也由艺术家在工具阶段定义。这包括为每种材质选择一个着色器，根据着色器的需求选择纹理，以及指定每个着色器的配置参数和选项。纹理被映射到表面，其他顶点属性也需要定义，通常是通过使用 DCC 应用程序中某种直观的工具来“绘制”它们。

材质通常使用商业或定制的内部材质编辑器进行创作。材质编辑器有时作为插件直接集成到 DCC 应用程序中，也可能是一个独立的程序。一些材质编辑器与游戏实时链接，以便材质创作者可以看到材质在真实游戏中的效果。其他编辑器提供离线 3D 可视化视图。一些编辑器甚至允许美术师或着色器工程师编写和调试着色器程序。这类工具允许通过使用鼠标连接各种节点来快速创建视觉效果的原型。这些工具通常以所见即所得的方式显示生成的材质。NVIDIA 的 Fx Composer 就是这样一个工具。遗憾的是，NVIDIA 不再更新 Fx Compose r，它仅支持最高 DirectX 10 的着色器模型。但他们确实提供了一个名为 NVI DIA® NsightTM Visual Studio Edition 的新 Visual Studio 插件。如图 11.42 所示，Nsight 提供了强大的着色器创作和调试功能。虚幻引擎还提供了一个名为“材质编辑器”的图形着色器编辑器；如图 11.43 所示。

材质可以与各个网格一起存储和管理。然而，这可能会导致数据和工作量的重复。在许多游戏中，可以使用相对较少的材质来定义游戏中的各种对象。例如，我们可能会定义一些标准的、可重复使用的材质，例如木材、岩石、金属、塑料、布料、皮肤等等。没有必要在每个网格中重复使用这些材质。相反，许多游戏团队会建立一个材质库供用户选择，然后各个网格以松散耦合的方式引用这些材质。

SketchUp, etc. The models may be defined using any convenient surface description—NURBS, quads, triangles, etc. However, they are invariably tessellated into triangles prior to rendering by the runtime portion of the pipeline.

The vertices of a mesh may also be skinned. This involves associating each vertex with one or more joints in an articulated skeletal structure, along with weights describing each joint's relative influence over the vertex. Skinning information and the skeleton are used by the animation system to drive the movements of a model—see Chapter 12 for more details.

Materials are also defined by the artists during the tools stage. This involves selecting a shader for each material, selecting textures as required by the shader, and specifying the configuration parameters and options of each shader. Textures are mapped onto the surfaces, and other vertex attributes are also defined, often by "painting" them with some kind of intuitive tool within the DCC application.

Materials are usually authored using a commercial or custom in-house material editor. The material editor is sometimes integrated directly into the DCC application as a plug-in, or it may be a stand-alone program. Some material editors are live-linked to the game, so that material authors can see what the materials will look like in the real game. Other editors provide an offline 3D visualization view. Some editors even allow shader programs to be written and debugged by the artist or a shader engineer. Such tools allow rapid prototyping of visual effects by connecting various kinds of nodes together with a mouse. These tools generally provide a WYSIWYG display of the resulting material. NVIDIA's Fx Composer is an example of such a tool. Sadly, NVIDIA is no longer updating Fx Composer, and it only supports shader models up to DirectX 10. But they do offer a new Visual Studio plugin called NVIDIA® Nsight™ Visual Studio Edition. Depicted in Figure 11.42, Nsight provides powerful shader authoring and debugging facilities. The Unreal Engine also provides a graphical shader editor called Material Editor; it is shown in Figure 11.43.

Materials may be stored and managed with the individual meshes. However, this can lead to duplication of data—and effort. In many games, a relatively small number of materials can be used to define a wide range of objects in the game. For example, we might define some standard, reusable materials like wood, rock, metal, plastic, cloth, skin and so on. There's no reason to duplicate these materials inside every mesh. Instead, many game teams build up a library of materials from which to choose, and the individual meshes refer to the materials in a loosely coupled manner.

图 11.42。NVIDIA® NsightTM Visual Studio Edition 允许轻松编写、预览和调试着色器程序。

11.2.3 资产调节阶段

资源调节阶段本身就是一个管线，有时也称为资源调节管线 (ACP) 或工具管线。正如我们在 7.2.1.4 节中看到的，它的作用是导出、处理并将多种类型的资源链接成一个整体。例如，一个 3D 模型由几何体（顶点和索引缓冲区）、材质、纹理和一个可选的骨架组成。ACP 确保 3D 模型引用的所有单个资源都可用，并准备好被引擎加载。

几何和材质数据从 DCC 应用程序中提取，通常以独立于平台的中间格式存储。然后，根据引擎支持的目标平台数量，这些数据会被进一步处理成一种或多种平台特定的格式。理想情况下，此阶段生成的平台特定资源可以直接加载到内存中，运行时几乎无需后期处理即可使用。例如，针对 Xbox One 或 PS4 的网格数据可能会输出为索引和顶点缓冲区，可供 GPU 使用；在 PS3 上，几何体可能会以压缩数据流的形式生成，以便通过 DMA 传输到 SPU 进行解压。ACP 通常会考虑材质/着色器的需求。

Figure 11.42. NVIDIA® Nsight™ Visual Studio Edition allows shader programs to be written, previsualized and debugged easily.

11.2.3 The Asset Conditioning Stage

The asset conditioning stage is itself a pipeline, sometimes called the *asset conditioning pipeline* (ACP) or the *tools pipeline*. As we saw in Section 7.2.1.4, its job is to export, process and link together multiple types of assets into a cohesive whole. For example, a 3D model is comprised of geometry (vertex and index buffers), materials, textures and an optional skeleton. The ACP ensures that all of the individual assets referenced by a 3D model are available and ready to be loaded by the engine.

Geometric and material data is extracted from the DCC application and is usually stored in a platform-independent intermediate format. The data is then further processed into one or more platform-specific formats, depending on how many target platforms the engine supports. Ideally the platform-specific assets produced by this stage are ready to load into memory and use with little or no postprocessing at runtime. For example, mesh data targeted for the Xbox One or PS4 might be output as index and vertex buffers that are ready to be consumed by the GPU; on the PS3, geometry might be produced in compressed data streams that are ready to be DMA'd to the SPUs for decompression. The ACP often takes the needs of the material/shader into account

图 11.43.虚幻引擎 4 材质编辑器。

在构建资产时。例如，特定着色器可能需要切线和副切线向量以及顶点法线；ACP 可以自动生成这些向量。

在资源调节阶段，还可以计算高级场景图数据结构。例如，可以处理静态几何体以构建 BSP 树。（我们将在 11.2.7.4 节中探讨，场景图数据结构可以帮助渲染引擎在给定特定摄像机位置和方向的情况下快速确定需要渲染的对象。）

昂贵的光照计算通常作为资源调节阶段的一部分离线完成。这被称为静态光照；它可能包括计算网格顶点的光色（这被称为“烘焙”顶点光照）、构建编码每像素光照信息的纹理贴图（称为光照贴图）、计算预计算辐射传输 (PRT) 系数（通常用球谐函数表示）等等。

11.2.4 GPU管道

图形硬件是围绕一种称为图形处理单元（GPU）的专用微处理器发展起来的。正如我们在 4.11 节中讨论的那样，

Figure II.43. The Unreal Engine 4 Material Editor.

when building assets. For example, a particular shader might require tangent and bitangent vectors as well as a vertex normal; the ACP could generate these vectors automatically.

High-level *scene graph* data structures may also be computed during the asset conditioning stage. For example, static-level geometry may be processed in order to build a BSP tree. (As we'll investigate in Section 11.2.7.4, scene graph data structures help the rendering engine to very quickly determine which objects should be rendered, given a particular camera position and orientation.)

Expensive lighting calculations are often done offline as part of the asset conditioning stage. This is called *static lighting*; it may include calculation of light colors at the vertices of a mesh (this is called "baked" vertex lighting), construction of texture maps that encode per-pixel lighting information known as *light maps*, calculation of *precomputed radiance transfer* (PRT) coefficients (usually represented by spherical harmonic functions) and so on.

II.2.4 The GPU Pipeline

Graphics hardware has evolved around a specialized type of microprocessor known as a *graphics processing unit* or GPU. As we discussed in Section 4.11,

图 11.44 典型 GPU 实现的渲染管线的几何处理和光栅化阶段。白色阶段表示可编程，浅灰色阶段表示可配置，深灰色框表示固定功能。

GPU 旨在最大限度地提高图形流水线的吞吐量，它通过大规模并行化顶点处理和逐像素着色计算等任务来实现这一目标。例如，像 AMD RadeonTM 7970 这样的现代 GPU 可以实现 4 TFLOPS 的峰值性能，这是通过在 32 个计算单元上并行执行工作负载来实现的，每个计算单元包含四个 16 通道 SIMD VPU，而每个计算单元又执行由 64 个线程组成的流水线波前。GPU 可用于渲染图形，但如今的 GPU 也是完全可编程的，允许程序员利用 GPU 强大的计算能力来执行计算着色器。这被称为通用 GPU 计算 (GPGPU)。

几乎所有 GPU 都会将图形流水线划分为如下所述的子阶段，如图 11.44 所示。每个阶段都用阴影表示其功能是可编程的、固定但可配置的，还是固定且不可配置的。

11.2.4.1 顶点着色器

此阶段完全可编程。它负责对各个顶点进行变换和着色/光照处理。此阶段的输入是单个顶点（尽管实际上会并行处理多个顶点）。其位置和法线通常在模型空间或世界空间中表示。顶点着色器通过模型视图变换处理从模型空间到视图空间的变换。此外，它还会应用透视投影、逐顶点光照和纹理计算以及动画角色的蒙皮。顶点着色器还可以通过修改顶点的位置来执行程序化动画。例如，在微风中摇曳的树叶或起伏的水面。此阶段的输出是一个完全变换并经过光照处理的顶点，其位置和法线以均匀裁剪表示。

Figure 11.44. The geometry processing and rasterization stages of the rendering pipeline, as implemented by a typical GPU. The white stages are programmable, the light grey stages are configurable, and the dark grey boxes are fixed-function.

a GPU is designed to maximize throughput of the graphics pipeline, which it achieves through massive parallelization of tasks like vertex processing and per-pixel shading calculations. For example, a modern GPU like the AMD Radeon™ 7970 can achieve a peak performance of 4 TFLOPS, which it does by executing workloads in parallel across 32 compute units, each of which contains four 16-lane SIMD VPUs, which in turn execute pipelined wavefronts consisting of 64 threads each. A GPU can be used to render graphics, but today's GPUs are also fully programmable, allowing programmers to leverage the awesome computing power of a GPU to execute *compute shaders*. This is known as *general-purpose GPU* computing (GPGPU).

Virtually all GPUs break the graphics pipeline into the substages described below and depicted in Figure 11.44. Each stage is shaded to indicate whether its functionality is programmable, fixed but configurable, or fixed and non-configurable.

11.2.4.1 Vertex Shader

This stage is fully programmable. It is responsible for transformation and shading/lighting of individual vertices. The input to this stage is a single vertex (although in practice many vertices are processed in parallel). Its position and normal are typically expressed in model space or world space. The vertex shader handles transformation from model space to view space via the model-view transform. Perspective projection is also applied, as well as per-vertex lighting and texturing calculations, and skinning for animated characters. The vertex shader can also perform procedural animation by modifying the position of the vertex. Examples of this include foliage that sways in the breeze or an undulating water surface. The output of this stage is a fully transformed and lit vertex, whose position and normal are expressed in homogeneous clip

空间（参见第 11.1.4.4 节）。

在现代 GPU 上，顶点着色器可以完全访问纹理数据——这一功能过去仅供像素着色器使用。当纹理用作独立的数据结构（例如高度图或查找表）时，这一点尤为有用。

11.2.4.2 几何着色器

这个可选阶段也是完全可编程的。几何着色器在齐次裁剪空间中对整个图元（三角形、线和点）进行操作。它能够剔除或修改输入图元，还可以生成新的图元。典型的用途包括：阴影体积挤压（参见第 11.3.3.1 节）、渲染立方体贴图的六个面（参见第 11.3.1.4 节）、网格轮廓边缘的毛鳍挤压、基于点数据创建粒子四边形（参见第 11.4.1 节）、动态曲面细分、线段的分形细分以实现闪电效果、布料模拟等等。

11.2.4.3 流输出

一些 GPU 允许将流水线中已处理的数据写回到内存。然后，这些数据可以循环回到流水线顶部进行进一步处理。此功能称为流输出。

流输出允许在没有 CPU 帮助的情况下实现许多引人入胜的视觉效果。一个很好的例子就是头发渲染。头发通常表示为三次样条曲线的集合。过去，头发的物理模拟是在 CPU 上完成的。CPU 还会将样条曲线细分为线段。最后，GPU 会渲染这些线段。

通过流输出，GPU 可以在顶点着色器中对毛发样条线的控制点进行物理模拟。几何着色器对样条线进行曲面细分，然后使用流输出功能将曲面细分后的顶点数据写入内存。之后，线段会被传输回流水线顶部，以便进行渲染。

11.2.4.4 剪辑

裁剪阶段会切掉三角形中横跨截头体的部分。裁剪过程如下：识别位于截头体外部的顶点，然后找到三角形边与截头体平面的交点。这些交点将成为定义一个或多个裁剪三角形的新顶点。

此阶段的功能是固定的，但在一定程度上是可配置的。例如，除了视锥体之外，还可以添加用户定义的裁剪平面

space (see Section 11.1.4.4).

On modern GPUs, the vertex shader has full access to texture data—a capability that used to be available only to the pixel shader. This is particularly useful when textures are used as stand-alone data structures like heightmaps or look-up tables.

11.2.4.2 Geometry Shader

This optional stage is also fully programmable. The geometry shader operates on entire primitives (triangles, lines and points) in homogeneous clip space. It is capable of culling or modifying input primitives, and it can also generate new primitives. Typical uses include shadow volume extrusion (see Section 11.3.3.1), rendering the six faces of a cube map (see Section 11.3.1.4), fur fin extrusion around silhouette edges of meshes, creation of particle quads from point data (see Section 11.4.1), dynamic tessellation, fractal subdivision of line segments for lightning effects, cloth simulations, and the list goes on.

11.2.4.3 Stream Output

Some GPUs permit the data that has been processed up to this point in the pipeline to be written back to memory. From there, it can then be looped back to the top of the pipeline for further processing. This feature is called *stream output*.

Stream output permits a number of intriguing visual effects to be achieved without the aid of the CPU. An excellent example is hair rendering. Hair is often represented as a collection of cubic spline curves. It used to be that hair physics simulation would be done on the CPU. The CPU would also tessellate the splines into line segments. Finally the GPU would render the segments.

With stream output, the GPU can do the physics simulation on the control points of the hair splines within the vertex shader. The geometry shader tessellates the splines, and the stream output feature is used to write the tessellated vertex data to memory. The line segments are then piped back into the top of the pipeline so they can be rendered.

11.2.4.4 Clipping

The clipping stage chops off those portions of the triangles that straddle the frustum. Clipping is done by identifying vertices that lie outside the frustum and then finding the intersection of the triangle's edges with the planes of the frustum. These intersection points become new vertices that define one or more clipped triangles.

This stage is fixed in function, but it is somewhat configurable. For example, user-defined clipping planes can be added in addition to the frustum

平面。此阶段还可以配置为剔除完全位于视锥体外的三角形。

11.2.4.5 屏幕映射

屏幕映射只是将顶点从均匀裁剪空间缩放并移动到屏幕空间。此阶段完全固定且不可配置。

11.2.4.6 三角形设置

在三角形设置期间，光栅化硬件会被初始化，以便高效地将三角形转换为片段。此阶段不可配置。

11.2.4.7 三角形遍历

每个三角形在三角形遍历阶段被分解成碎片（即光栅化）。通常每个像素生成一个碎片，但某些抗锯齿技术可以为每个像素创建多个碎片（参见第 11.1.4.7 节）。三角形遍历阶段还会对顶点属性进行插值，以便生成每个碎片的属性，供像素着色器处理。在适当的情况下，会使用透视校正插值。此阶段的功能是固定的，不可配置。

11.2.4.8 早期 z 检验

许多显卡能够在管线的这个阶段检查片段的深度，如果该片段被帧缓冲区中已有的像素遮挡，则丢弃该片段。这样，对于被遮挡的片段，就可以完全跳过（可能非常昂贵的）像素着色器阶段。

令人惊讶的是，并非所有图形硬件都支持在流水线的这个阶段进行深度测试。在较旧的 GPU 设计中，z 测试是在像素着色器运行后与 alpha 测试一起进行的。因此，此阶段被称为早期 z 测试或早期深度测试阶段。

11.2.4.9 像素着色器

此阶段完全可编程。它的任务是着色（即，光照和其他处理）每个片段。像素着色器也可以丢弃某些片段，例如，因为它们被视为完全透明。像素着色器可以处理一个或多个纹理贴图，运行逐像素光照计算，并执行任何其他必要的操作来确定片段的颜色。

此阶段的输入是每个片段属性的集合（这些属性由三角形遍历阶段根据顶点属性插值而来）。输出是一个描述片段所需颜色的单一颜色向量。

planes. This stage can also be configured to cull triangles that lie entirely outside the frustum.

11.2.4.5 Screen Mapping

Screen mapping simply scales and shifts the vertices from homogeneous clip space into screen space. This stage is entirely fixed and non-configurable.

11.2.4.6 Triangle Set-up

During triangle set-up, the rasterization hardware is initialized for efficient conversion of the triangle into fragments. This stage is not configurable.

11.2.4.7 Triangle Traversal

Each triangle is broken into fragments (i.e., rasterized) by the triangle traversal stage. Usually one fragment is generated for each pixel, although with certain antialiasing techniques, multiple fragments may be created per pixel (see Section 11.1.4.7). The triangle traversal stage also interpolates vertex attributes in order to generate per-fragment attributes for processing by the pixel shader. Perspective-correct interpolation is used where appropriate. This stage's functionality is fixed and not configurable.

11.2.4.8 Early z-Test

Many graphics cards are capable of checking the depth of the fragment at this point in the pipeline, discarding it if it is being occluded by the pixel already in the frame buffer. This allows the (potentially very expensive) pixel shader stage to be skipped entirely for occluded fragments.

Surprisingly, not all graphics hardware supports depth testing at this stage of the pipeline. In older GPU designs, the z-test was done along with alpha testing, after the pixel shader had run. For this reason, this stage is called the *early z-test* or *early depth test* stage.

11.2.4.9 Pixel Shader

This stage is fully programmable. Its job is to shade (i.e., light and otherwise process) each fragment. The pixel shader can also discard fragments, for example because they are deemed to be entirely transparent. The pixel shader can address one or more texture maps, run per-pixel lighting calculations, and do whatever else is necessary to determine the fragment's color.

The input to this stage is a collection of per-fragment attributes (which have been interpolated from the vertex attributes by the triangle traversal stage). The output is a single color vector describing the desired color of the fragment.

11.2.4.10 合并/光栅操作阶段

管线的最后一个阶段称为合并阶段或混合阶段，在 NVIDIA 术语中也称为光栅操作阶段或 ROP。此阶段不可编程，但高度可配置。它负责运行各种片段测试，包括深度测试（参见第 11.1.4.8 节）、Alpha 测试（其中片段和像素的 Alpha 通道值可用于剔除某些片段）和模板测试（参见第 11.3.3.1 节）。

如果片段通过了所有测试，它的颜色就会与帧缓冲区中已有的颜色混合（合并）。混合的方式由 Alpha 混合函数控制——该函数的基本结构是固定的，但其操作符和参数可以配置，以产生各种各样的混合操作。

Alpha 混合最常用于渲染半透明几何体。在这种情况下，使用以下混合函数：

$$\mathbf{C}'_D = A_S\mathbf{C}_S + (1 - A_S)\mathbf{C}_D.$$

下标 S 和 D 分别代表“源”（传入的片段）和“目标”（帧缓冲区中的像素）。因此，写入帧缓冲区的颜色（C ′ D）是现有帧缓冲区内容（CD）和正在绘制的片段颜色（CS）的加权平均值。混合权重（AS）就是传入片段的源 Alpha 值。

为了使 Alpha 混合效果正确，在将不透明几何体渲染到帧缓冲区后，必须按从后到前的顺序对场景中的半透明和半透明表面进行排序和渲染。这是因为执行 Alpha 混合后，新片段的深度会覆盖与其混合的像素的深度。换句话说，深度缓冲区会忽略透明度（当然，除非已关闭深度写入）。如果我们在不透明背景上渲染一堆半透明物体，则最终的像素颜色理想情况下应该是不透明表面颜色与堆栈中所有半透明表面颜色的混合。如果我们尝试以从后到前以外的任何顺序渲染堆栈，深度测试失败将导致部分半透明片段被丢弃，从而导致混合不完整（以及看起来相当奇怪的图像）。

除了透明度混合之外，还可以定义其他 Alpha 混合函数，用于其他用途。通用混合方程的形式为 C ′ D = (w S □ CS) + (w D □ CD)，其中加权因子 w S 和 w D 可由程序员从一组预定义的值中选择，包括零。

11.2.4.10 Merging / Raster Operations Stage

The final stage of the pipeline is known as the *merging stage* or *blending stage*, also known as the *raster operations* stage or ROP in NVIDIA parlance. This stage is not programmable, but it is highly configurable. It is responsible for running various fragment tests including the depth test (see Section 11.1.4.8), alpha test (in which the values of the fragment's and pixel's alpha channels can be used to reject certain fragments) and stencil test (see Section 11.3.3.1).

If the fragment passes all of the tests, its color is blended (merged) with the color that is already present in the frame buffer. The way in which blending occurs is controlled by the *alpha blending function*—a function whose basic structure is hard-wired, but whose operators and parameters can be configured in order to produce a wide variety of blending operations.

Alpha blending is most commonly used to render semitransparent geometry. In this case, the following blending function is used:

$$\mathbf{C}'_D = A_S\mathbf{C}_S + (1 - A_S)\mathbf{C}_D.$$

The subscripts S and D stand for "source" (the incoming fragment) and "destination" (the pixel in the frame buffer), respectively. Therefore, the color that is written into the frame buffer ($\mathbf{C}'_D$) is a *weighted average* of the existing frame buffer contents ($\mathbf{C}_D$) and the color of the fragment being drawn ($\mathbf{C}_S$). The blend weight (A_S) is just the source alpha of the incoming fragment.

For alpha blending to look right, the semitransparent and translucent surfaces in the scene must be sorted and rendered in back-to-front order, *after* the opaque geometry has been rendered to the frame buffer. This is because after alpha blending has been performed, the depth of the new fragment *overwrites* the depth of the pixel with which it was blended. In other words, the depth buffer ignores transparency (unless depth writes have been turned off, of course). If we are rendering a stack of translucent objects on top of an opaque backdrop, the resulting pixel color should ideally be a blend between the opaque surface's color and the colors of *all* of the translucent surfaces in the stack. If we try to render the stack in any order other than back-to-front, depth-test failures will cause some of the translucent fragments to be discarded, resulting in an incomplete blend (and a rather odd-looking image).

Other alpha blending functions can be defined as well, for purposes other than transparency blending. The general blending equation takes the form $\mathbf{C}'_D = (\mathbf{w}_S \otimes \mathbf{C}_S) + (\mathbf{w}_D \otimes \mathbf{C}_D)$, where the weighting factors $\mathbf{w}_S$ and $\mathbf{w}_D$ can be selected by the programmer from a predefined set of values including zero,

一个是源或目标颜色值，一个是源或目标 Alpha 值，另一个是源或目标颜色值或 Alpha 值减一。运算符 □ 可以是常规的标量-向量乘法，也可以是按分量的向量-向量乘法（Hadamard 积——参见 5.2.4.1 节），具体取决于 w S 和 w D 的数据类型。

11.2.5 可编程着色器

现在我们已经对 GPU 流水线有了端到端的了解，接下来让我们更深入地了解流水线中最有趣的部分——可编程着色器。自 DirectX 8 推出以来，着色器架构已经发生了显著的变化。早期的着色器模型仅支持低级汇编语言编程，像素着色器的指令集和寄存器集与顶点着色器的指令集和寄存器集存在显著差异。DirectX 9 引入了对类似 C 语言的高级着色器语言的支持，例如 Cg（图形 C 语言）、HLSL（高级着色语言——微软对 Cg 语言的实现）和 GLSL（OpenGL 着色语言）。DirectX 10 引入了几何着色器，随之而来的是统一的着色器架构，在 DirectX 术语中称为着色器模型 4.0。在统一着色器模型中，所有三种类型的着色器都支持大致相同的指令集，并具有大致相同的功能集，包括读取纹理内存的能力。

着色器获取输入数据的单个元素并将其转换为零个或多个输出数据元素。

- 对于顶点着色器，输入是一个顶点，其位置和法线在模型空间或世界空间中表示。顶点着色器的输出是一个完全变换和点亮的顶点，在齐次裁剪空间中表示。

- 几何着色器的输入是一个包含 n 个顶点的图元——一个点（n = 1）、线段（n = 2）或三角形（n = 3）——以及最多 n 个用作控制点的附加顶点。输出是零个或多个图元，其类型可能与输入不同。例如，几何着色器可以将点转换为包含两个三角形的四边形，也可以将三角形转换为三角形，但可以选择丢弃一些三角形，等等。

- 像素着色器的输入是一个片段，其属性已从其来源三角形的三个顶点进行插值。像素着色器的输出是将写入帧缓冲区的颜色（假设该片段通过了深度测试和其他可选

one, source or destination color, source or destination alpha and one minus the source or destination color or alpha. The operator $\otimes$ is either a regular scalar-vector multiplication or a component-wise vector-vector multiplication (a Hadamard product—see Section 5.2.4.1) depending on the data types of $\mathbf{w}_S$ and $\mathbf{w}_D$.

11.2.5 Programmable Shaders

Now that we have an end-to-end picture of the GPU pipeline in mind, let's take a deeper look at the most interesting part of the pipeline—the programmable shaders. Shader architectures have evolved significantly since their introduction with DirectX 8. Early *shader models* supported only low-level assembly language programming, and the instruction set and register set of the pixel shader differed significantly from those of the vertex shader. DirectX 9 brought with it support for high-level C-like shader languages such as Cg (C for graphics), HLSL (High-Level Shading Language—Microsoft's implementation of the Cg language) and GLSL (OpenGL shading language). With DirectX 10, the geometry shader was introduced, and with it came a unified shader architecture called *shader model 4.0* in DirectX parlance. In the unified shader model, all three types of shaders support roughly the same instruction set and have roughly the same set of capabilities, including the ability to read texture memory.

A shader takes a single element of input data and transforms it into zero or more elements of output data.

- In the case of the vertex shader, the input is a vertex whose position and normal are expressed in model space or world space. The output of the vertex shader is a fully transformed and lit vertex, expressed in homogeneous clip space.
- The input to the geometry shader is a single n-vertex primitive—a point ($n = 1$), line segment ($n = 2$) or triangle ($n = 3$)—with up to n additional vertices that act as control points. The output is zero or more primitives, possibly of a different type than the input. For example, the geometry shader could convert points into two-triangle quads, or it could transform triangles into triangles but optionally discard some triangles and so on.
- The pixel shader's input is a fragment whose attributes have been interpolated from the three vertices of the triangle from which it came. The output of the pixel shader is the color that will be written into the frame buffer (presuming the fragment passes the depth test and other optional

测试）。像素着色器还能够明确丢弃片段，在这种情况下它不会产生任何输出。

11.2.5.1 访问内存

由于 GPU 实现了数据处理流水线，因此对 RAM 的访问受到严格控制。着色器程序通常无法直接读取或写入内存。相反，它的内存访问仅限于两种方式：

寄存器和纹理贴图。

然而，我们应该注意到，在 GPU 和 CPU 直接共享内存的系统上，这些限制会被取消。例如，PlayStation 4 核心的 AMD Jaguar 片上系统 (SoC) 就是异构系统架构 (HSA) 的一个例子。在非 HSA 系统中，CPU 和 GPU 通常是独立的设备，各自拥有私有内存，并且通常分别位于独立的电路板上。在两个处理器之间传输数据需要通过专用总线（例如 AGP 或 PCIe）进行繁琐且高延迟的通信。在 HSA 中，CPU 和 GPU 共享一个统一的内存存储，称为异构统一内存架构 (hUMA)。因此，在具有 hUMA 的系统（例如 PS4）上运行的着色器可以接收着色器资源表 □□(SRT) 作为输入。这只是一个指向内存中 C/C++ 结构体的指针，CPU 和运行在 GPU 上的着色器都可以读取或写入该结构体。在 PS4 上，SRT 取代了以下章节中描述的常量寄存器。

着色器寄存器

着色器可以通过寄存器间接访问 RAM。所有 GPU 寄存器均为 128 位 SIMD 格式。每个寄存器能够保存四个 32 位浮点数或整数值（在 Cg 语言中用 float4 数据类型表示）。这样的寄存器可以包含齐次坐标中的四元素向量或 RGBA 格式的颜色，其中每个分量都是 32 位浮点格式。矩阵可以用三到四个寄存器组表示（在 Cg 中用内置矩阵类型（如 float4x4）表示）。GPU 寄存器也可用于保存单个 32 位标量，在这种情况下，该值通常会复制到所有四个 32 位字段中。某些 GPU 可以对 16 位字段（称为 halfs）进行操作。（为此，Cg 提供了各种内置类型，如 half4 和 half4x4。）

寄存器有以下四种类型：

- 输入寄存器。这些寄存器是着色器输入数据的主要来源。在顶点着色器中，输入寄存器包含属性数据

tests). The pixel shader is also capable of discarding fragments explicitly, in which case it produces no output.

11.2.5.1 Accessing Memory

Because the GPU implements a data processing pipeline, access to RAM is carefully controlled. A shader program usually cannot read from or write to memory directly. Instead, its memory accesses are limited to two methods: registers and texture maps.

However, we should note that these restrictions are lifted on systems in which the GPU and CPU share memory directly. For example, the AMD Jaguar system on a chip (SoC) that sits at the heart of the PlayStation 4 is an example of a *heterogeneous system architecture* (HSA). On a non-HSA system, the CPU and GPU are typically separate devices, each with its own private memory, and each usually residing on a separate circuit board. Transferring data between the two processors requires cumbersome, high-latency communication over a specialized bus such as AGP or PCIe. With HSA, the CPU and GPU share a single unified memory store called a *heterogeneous unified memory architecture* (hUMA). Shaders running on a system with hUMA, like the PS4, can therefore be passed a *shader resource table* (SRT) as input. This is just a pointer to a C/C++ `struct` in memory that can be read from or written to by both the CPU and the shader running on the GPU. On the PS4, SRTs take the place of the constant registers described in the following sections.

Shader Registers

A shader can access RAM indirectly via *registers*. All GPU registers are in 128-bit SIMD format. Each register is capable of holding four 32-bit floating-point or integer values (represented by the `float4` data type in the Cg language). Such a register can contain a four-element vector in homogeneous coordinates or a color in RGBA format, with each component in 32-bit floating-point format. Matrices can be represented by groups of three or four registers (represented by built-in matrix types like `float4x4` in Cg). A GPU register can also be used to hold a single 32-bit scalar, in which case the value is usually replicated across all four 32-bit fields. Some GPUs can operate on 16-bit fields, known as *halfs*. (Cg provides various built-in types like `half4` and `half4x4` for this purpose.)

Registers come in four flavors, as follows:

- *Input registers*. These registers are the shader's primary source of input data. In a vertex shader, the input registers contain attribute data

直接从顶点获取。在像素着色器中，输入寄存器包含与单个片段对应的插值顶点属性数据。所有输入寄存器的值均由 GPU 在调用着色器之前自动设置。

• 常量寄存器。常量寄存器的值由应用程序设置，并且会随着图元的不同而变化。只有从着色器程序的角度来看，它们的值才是恒定的。它们为着色器提供了第二种输入形式。典型的内容包括模型视图矩阵、投影矩阵、光照参数以及着色器所需的任何其他顶点属性中未提供的参数。

• 临时寄存器。这些寄存器供着色器程序内部使用，通常用于存储计算的中间结果。

• 输出寄存器。这些寄存器的内容由着色器填充，并作为其唯一的输出形式。在顶点着色器中，输出寄存器包含顶点属性，例如齐次裁剪空间中的变换位置和法向量、可选的顶点颜色、纹理坐标等等。在像素着色器中，输出寄存器包含被着色片段的最终颜色。

应用程序在提交图元进行渲染时，会提供常量寄存器的值。GPU 在调用着色器程序之前，会自动将顶点或片段属性数据从视频 RAM 复制到相应的输入寄存器中，并在程序执行结束时将输出寄存器的内容写回 RAM，以便将数据传递到流水线的下一阶段。

GPU 通常会缓存输出数据，以便无需重新计算即可重复使用。例如，后变换顶点缓存存储了顶点着色器发出的最近处理的顶点。如果遇到引用先前处理过的顶点的三角形，则会尽可能从后变换顶点缓存中读取该顶点——只有当相关顶点已从缓存中移除以便为新处理的顶点腾出空间时，才需要再次调用顶点着色器。

纹理

着色器还可以直接读取纹理贴图。纹理数据通过纹理坐标进行寻址，而不是通过绝对内存地址。GPU 的纹理采样器会自动过滤纹理数据，并根据需要在相邻的纹素或相邻的 mipmap 级别之间混合值。纹理

obtained directly from the vertices. In a pixel shader, the input registers contain interpolated vertex attribute data corresponding to a single fragment. The values of all input registers are set automatically by the GPU prior to invoking the shader.

- *Constant registers*. The values of constant registers are set by the application and can change from primitive to primitive. Their values are constant only from the point of view of the shader program. They provide a secondary form of input to the shader. Typical contents include the model-view matrix, the projection matrix, light parameters and any other parameters required by the shader that are not available as vertex attributes.
- *Temporary registers*. These registers are for use by the shader program internally and are typically used to store intermediate results of calculations.
- *Output registers*. The contents of these registers are filled in by the shader and serve as its only form of output. In a vertex shader, the output registers contain vertex attributes such as the transformed position and normal vectors in homogeneous clip space, optional vertex colors, texture coordinates and so on. In a pixel shader, the output register contains the final color of the fragment being shaded.

The application provides the values of the constant registers when it submits primitives for rendering. The GPU automatically copies vertex or fragment attribute data from video RAM into the appropriate input registers prior to calling the shader program, and it also writes the contents of the output registers back into RAM at the conclusion of the program's execution so that the data can be passed to the next stage of the pipeline.

GPUs typically cache output data so that it can be reused without being recalculated. For example, the *post-transform vertex cache* stores the most-recently processed vertices emitted by the vertex shader. If a triangle is encountered that refers to a previously processed vertex, it will be read from the post-transform vertex cache if possible—the vertex shader need only be called again if the vertex in question has since been ejected from the cache to make room for newly processed vertices.

Textures

A shader also has direct read-only access to *texture maps*. Texture data is addressed via texture coordinates, rather than via absolute memory addresses. The GPU's texture samplers automatically *filter* the texture data, blending values between adjacent texels or adjacent mipmap levels as appropriate. Texture

可以禁用过滤，以便直接访问特定纹素的值。例如，当纹理贴图用作数据表时，此功能非常有用。

着色器只能以间接的方式写入纹理贴图——通过将场景渲染到屏幕外的帧缓冲区，该缓冲区在后续渲染过程中被解释为纹理贴图。此功能称为“渲染到纹理”。

11.2.5.2 高级着色器语言语法简介

像 Cg 和 GLSL 这样的高级着色器语言模仿了 C 编程语言。程序员可以声明函数、定义简单的结构体并执行算术运算。然而，正如我们上面所说，着色器程序只能访问寄存器和纹理。因此，我们在 Cg 或 GLSL 中声明的结构体和变量会被着色器编译器直接映射到寄存器上。我们通过以下方式定义这些映射：

- 语义。变量和结构体成员可以以冒号加后缀，后跟一个称为语义的关键字。语义会告诉着色器编译器将变量或数据成员绑定到特定的顶点或片段属性。例如，在顶点着色器中，我们可以声明一个输入结构体，其成员映射到顶点的位置和颜色属性，如下所示：

```
struct VtxOut
{
    float4 pos : POSITION; // map to position attribute
    float4 color : COLOR;  // map to color attribute
};
```

- 输入与输出。编译器根据特定变量或结构体所使用的上下文，确定其应映射到输入寄存器还是输出寄存器。如果变量作为参数传递给着色器程序的主函数，则假定其为输入；如果变量是主函数的返回值，则假定其为输出。

```
VtxOut vshaderMain(VtxIn in) // maps to input registers
{
    VtxOut out;
    // ...
    return out;  // maps to output registers
}
```

- 统一声明。为了通过常量寄存器访问应用程序提供的数据，我们可以使用关键字

filtering can be disabled in order to gain direct access to the values of particular texels. This can be useful when a texture map is used as a data table, for example.

Shaders can only *write* to texture maps in an indirect manner—by rendering the scene to an off-screen frame buffer that is interpreted as a texture map by subsequent rendering passes. This feature is known as *render to texture*.

11.2.5.2 Introduction to High-Level Shader Language Syntax

High-level shader languages like Cg and GLSL are modeled after the C programming language. The programmer can declare functions, define a simple `struct`, and perform arithmetic. However, as we said above, a shader program only has access to registers and textures. As such, the `struct` and variable we declare in Cg or GLSL is mapped directly onto registers by the shader compiler. We define these mappings in the following ways:

- *Semantics*. Variables and `struct` members can be suffixed with a colon followed by a keyword known as a *semantic*. The semantic tells the shader compiler to bind the variable or data member to a particular vertex or fragment attribute. For example, in a vertex shader we might declare an input `struct` whose members map to the *position* and *color* attributes of a vertex as follows:

```
struct VtxOut
{
    float4 pos : POSITION; // map to position attribute
    float4 color : COLOR;  // map to color attribute
};
```

- *Input versus output*. The compiler determines whether a particular variable or `struct` should map to input or output registers from the context in which it is used. If a variable is passed as an argument to the shader program's main function, it is assumed to be an input; if it is the return value of the main function, it is taken to be an output.

```
VtxOut vshaderMain(VtxIn in) // maps to input registers
{
    VtxOut out;
    // ...
    return out;  // maps to output registers
}
```

- *Uniform declaration*. To gain access to the data supplied by the application via the constant registers, we can declare a variable with the keyword

uniform 。例如，模型视图矩阵可以按如下方式传递给顶点着色器：

```
VtxOut vshaderMain(
    VtxIn in,
    uniform float4x4 modelViewMatrix)
{
    VtxOut out;
    // ...
    return out;
}
```

算术运算可以通过调用 C 语言风格的运算符或调用适当的内部函数来执行。例如，要将输入顶点位置乘以模型视图矩阵，我们可以这样写：

```
VtxOut vshaderMain(VtxIn in,
                   uniform float4x4 modelViewMatrix)
{
    VtxOut out;

    out.pos = mul(modelViewMatrix, in.pos);
    out.color = float4(0, 1, 0, 1); // RGBA green

    return out;
}
```

通过调用特殊的内部函数从纹理中获取数据，这些函数读取指定纹理坐标处的纹素值。有多种变体可用于读取各种格式的一维、二维和三维纹理，包括带或不带滤波的纹理。特殊的纹理寻址模式也可用于访问立方体贴图和阴影贴图。对纹理贴图本身的引用使用一种称为纹理采样器声明的特殊数据类型来声明。例如，数据类型 sampler2D 表示对典型二维纹理的引用。以下简单的 Cg 像素着色器将漫反射纹理应用于三角形：

```
struct FragmentOut
{
    float4 color : COLOR;
};

FragmentOut pshaderMain(float2 uv : TEXCOORD0,
                        uniform sampler2D texture)
{
    FragmentOut out;
```


uniform. For example, the model-view matrix could be passed to a vertex shader as follows:

```
VtxOut vshaderMain(
    VtxIn in,
    uniform float4x4 modelViewMatrix)
{
    VtxOut out;
    // ...
    return out;
}
```

Arithmetic operations can be performed by invoking C-style operators, or by calling intrinsic functions as appropriate. For example, to multiply the input vertex position by the model-view matrix, we could write:

```
VtxOut vshaderMain(VtxIn in,
                   uniform float4x4 modelViewMatrix)
{
    VtxOut out;

    out.pos = mul(modelViewMatrix, in.pos);
    out.color = float4(0, 1, 0, 1); // RGBA green

    return out;
}
```

Data is obtained from textures by calling special intrinsic functions that read the value of the texels at a specified texture coordinate. A number of variants are available for reading one-, two- and three-dimensional textures in various formats, with and without filtering. Special texture addressing modes are also available for accessing cube maps and shadow maps. References to the texture maps themselves are declared using a special data type known as a *texture sampler* declaration. For example, the data type sampler2D represents a reference to a typical two-dimensional texture. The following simple Cg pixel shader applies a diffuse texture to a triangle:

```
struct FragmentOut
{
    float4 color : COLOR;
};

FragmentOut pshaderMain(float2 uv : TEXCOORD0,
                        uniform sampler2D texture)
{
    FragmentOut out;
```

```
    // look up texel at (u,v)
    out.color = tex2D(texture, uv);

    return out;
}
```

11.2.5.3 效果文件

着色器程序本身并没有特别的用处。GPU 流水线需要额外的信息才能通过有意义的输入调用着色器程序。例如，我们需要指定应用程序指定的参数（如模型视图矩阵、光照参数等）如何映射到着色器程序中声明的统一变量。此外，某些视觉效果需要两次或多次渲染，但着色器程序仅描述在一次渲染过程中要应用的操作。如果我们为 PC 平台编写游戏，我们需要定义一些更高级渲染效果的“后备”版本，以便它们即使在较旧的显卡上也能工作。为了将着色器程序组合成完整的视觉效果，我们需要使用一种称为效果文件的文件格式。

不同的渲染引擎实现效果的方式略有不同。在 Cg 中，效果文件格式称为 CgFX 。OGRE 使用与 CgFX 非常相似的文件格式，称为材质文件 。GLSL 效果可以使用基于 XML 的 COLLADA 格式描述。尽管存在差异，但效果通常采用以下分层格式：

- 在全局范围内，定义结构、着色器程序（作为各种“主要”功能实现）和全局变量（映射到应用程序指定的常量参数）。
- 定义了一种或多种技术。一种技术代表渲染特定视觉效果的一种方法。一种效果通常提供一种主要技术来实现其最高质量的实现，并可能提供多种后备技术，以便在性能较低的图形硬件上使用。
- 每种技术都定义了一个或多个渲染通道。每个渲染通道描述了如何渲染单个全帧图像。它通常包含对顶点、几何和/或像素着色器程序“主”函数的引用、各种参数绑定以及可选的渲染状态设置。

11.2.5.4 进一步阅读

在本节中，我们只是对高级着色器编程有了一点了解——完整的教程超出了我们的范围。


```
    // look up texel at (u,v)
    out.color = tex2D(texture, uv);

    return out;
}
```

11.2.5.3 Effect Files

By itself, a shader program isn't particularly useful. Additional information is required by the GPU pipeline in order to call the shader program with meaningful inputs. For example, we need to specify how the application-specified parameters, like the model-view matrix, light parameters and so on, map to the `uniform` variables declared in the shader program. In addition, some visual effects require two or more rendering passes, but a shader program only describes the operations to be applied during a single rendering pass. If we are writing a game for the PC platform, we will need to define "fallback" versions of some of our more-advanced rendering effects, so that they will work even on older graphics cards. To tie our shader program(s) together into a complete visual effect, we turn to a file format known as an *effect file*.

Different rendering engines implement effects in slightly different ways. In Cg, the effect file format is known as *CgFX*. OGRE uses a file format very similar to CgFX known as a *material file*. GLSL effects can be described using the COLLADA format, which is based on XML. Despite the differences, effects generally take on the following hierarchical format:

- At global scope, structs, shader programs (implemented as various "main" functions) and global variables (which map to application-specified constant parameters) are defined.
- One or more *techniques* are defined. A technique represents one way to render a particular visual effect. An effect typically provides a primary technique for its highest-quality implementation and possibly a number of fallback techniques for use on lower-powered graphics hardware.
- Within each technique, one or more *passes* are defined. A pass describes how a single full-frame image should be rendered. It typically includes a reference to a vertex, geometry and/or pixel shader program's "main" function, various parameter bindings and optional render state settings.

11.2.5.4 Further Reading

In this section, we've only had a small taste of what high-level shader programming is like—a complete tutorial is beyond our scope here. For a much

图 11.45. 无抗锯齿（左）、4 × MSAA（中）以及 Nvidia FXAA 预设 3（右）。图片来自 Timothy Lottes 撰写的 Nvidia FXAA 白皮书 (http://bit.ly/1mIzCTv)。（参见彩色图版 XIX。）

关于Cg着色器编程的更详细介绍，请参考NVIDIA网站上的Cg教程，网址为https://developer.nvidia.com/content/hello-cg-introductory-tutorial。

11.2.6 抗锯齿

当三角形被光栅化时，它的边缘看起来会参差不齐——这就是我们熟悉且喜爱（或讨厌）的“阶梯状”效果。从技术上讲，产生混叠的原因是我们使用一组离散的像素来采样图像，而该图像实际上是一个平滑、连续的二维信号。（有关采样和混叠的详细讨论，请参见第 14.3.2.1 节。）术语“抗锯齿”描述了任何可以减少由混叠引起的视觉伪影的技术。对渲染场景进行抗锯齿处理的方法有很多种。几乎所有方法的最终效果都是通过将渲染三角形的边缘与周围像素混合来“柔化”渲染三角形的边缘。每种技术都有独特的性能、内存使用情况和质量特性。图 11.45 显示了首先未使用抗锯齿、然后使用 4 × MSAA 以及最后使用 Nvidia 的 FXAA 技术渲染的场景。

Figure 11.45. No antialiasing (left), 4× MSAA (center) and Nvidia's FXAA, preset 3 (right). Image from Nvidia's FXAA white paper by Timothy Lottes (http://bit.ly/1mIzCTv). (See Color Plate XIX.)

more detailed introduction to Cg shader programming, refer to the Cg tutorial available on NVIDIA's website at https://developer.nvidia.com/content/hello-cg-introductory-tutorial.

11.2.6 Antialiasing

When a triangle is rasterized, its edges can look jagged—the familiar "stair step" effect we have all come to know and love (or hate). Technically speaking, aliasing arises because we are using a discrete set of pixels to *sample* an image that is really a smooth, continuous two-dimensional signal. (See Section 14.3.2.1 for a detailed discussion of sampling and aliasing.)

The term *antialiasing* describes any technique that reduces the visual artifacts caused by aliasing. There are many different ways to antialias a rendered scene. The net effect of pretty much all of them is to "soften" the edges of rendered triangles by blending them with surrounding pixels. Each technique has unique performance, memory-usage and quality characteristics. Figure 11.45 shows a scene rendered first without antialiasing, then with 4× MSAA and finally with Nvidia's FXAA technique.

11.2.6.1 全屏抗锯齿 (FSAA)

这项技术也称为超级采样抗锯齿 (SSAA)，场景被渲染到比实际屏幕更大的帧缓冲区中。帧渲染完成后，生成的超大图像会被下采样至所需分辨率。在 4 倍超级采样中，渲染图像的宽度和高度分别是屏幕的两倍，因此帧缓冲区占用的内存是屏幕的四倍。由于像素着色器必须为每个屏幕像素运行四次，因此它需要四倍的 GPU 处理能力。由此可见，FSAA 是一项极其昂贵的技术，无论是在内存消耗方面还是在 GPU 周期方面。因此，它在实际应用中很少使用。

11.2.6.2 多重采样抗锯齿 (MSAA)

多重采样抗锯齿 (MSAA) 技术能够提供与 FSAA 相当的视觉质量，同时消耗更少的 GPU 带宽（以及相同的显存容量）。MSAA 方法基于以下观察：由于纹理 mipmapping 的自然抗锯齿效果，锯齿问题通常主要出现在三角形的边缘，而不是内部。

要理解 MSAA 的工作原理，请回想一下，光栅化三角形的过程实际上可以归结为三个不同的操作：（1）确定三角形重叠的像素（覆盖），（2）确定每个像素是否被其他三角形遮挡（深度测试）和（3）确定每个像素的颜色，假设覆盖和深度测试告诉我们该像素实际上应该被绘制（像素着色）。

在未启用抗锯齿的情况下光栅化三角形时，覆盖测试、深度测试和像素着色操作均在每个屏幕像素内的单个理想点（通常位于其中心）运行。在 MSAA 中，覆盖测试和深度测试针对每个屏幕像素内的 N 个点（称为子样本）运行。N 通常选择为 2、4、5、8 或 16。但是，无论我们使用多少个子样本，像素着色器每个屏幕像素仅运行一次。这使得 MSAA 在 GPU 带宽方面比 FSAA 具有巨大优势，因为着色通常比覆盖测试和深度测试开销大得多。

在 N × MSAA 中，深度、模板和颜色缓冲区的分配大小均为其原本大小的 N 倍。对于每个屏幕像素，这些缓冲区包含 N 个“槽位”，每个子样本对应一个槽位。光栅化三角形时，将对三角形每个片段内的 N 个子样本运行 N 次覆盖和深度测试。如果 N 次测试中至少有一次指示应该绘制该片段，则运行一次像素着色器。然后，从像素着色器获取的颜色仅存储在与以下情况对应的槽位中：

11.2.6.1 Full-Screen Antialiasing (FSAA)

In this technique, also known as *super-sampled antialiasing* (SSAA), the scene is rendered into a frame buffer that is larger than the actual screen. Once rendering of the frame is complete, the resulting oversized image is *downsampled* to the desired resolution. In 4× supersampling, the rendered image is twice as wide and twice as tall as the screen, resulting in a frame buffer that occupies four times the memory. It also requires four times the GPU processing power because the pixel shader must be run four times for each screen pixel. As you can see, FSAA is an incredibly expensive technique both in terms of memory consumption and GPU cycles. As such, it is rarely used in practice.

11.2.6.2 Multisampled Antialiasing (MSAA)

Multisampled antialiasing is a technique that provides visual quality comparable to that of FSAA, while consuming a great deal less GPU bandwidth (and the same amount of video RAM). The MSAA approach is based on the observation that, thanks to the natural antialiasing effect of texture *mipmapping*, aliasing tends to be a problem primarily at the *edges* of triangles, not in their interiors.

To understand how MSAA works, recall that the process of rasterizing a triangle really boils down to three distinct operations: (1) Determining which pixels the triangle overlaps (coverage), (2) determining whether or not each pixel is occluded by some other triangle (depth testing) and (3) determining the color of each pixel, presuming that the coverage and depth tests tell us that the pixel should in fact be drawn (pixel shading).

When rasterizing a triangle without antialiasing, the coverage test, depth test and pixel shading operations are all run at a single idealized point within each screen pixel, usually located at its center. In MSAA, the coverage and depth tests are run for N points known as *subsamples* within each screen pixel. N is typically chosen to be 2, 4, 5, 8 or 16. However, the pixel shader is only run *once* per screen pixel, no matter how many subsamples we use. This gives MSAA a big advantage over FSAA in terms of GPU bandwidth, because shading is typically a great deal more expensive than coverage and depth testing.

In $N\times$ MSAA, the depth, stencil and color buffers are each allocated to be N times as large as they would otherwise be. For each screen pixel, these buffers contain N “slots,” one slot for each subsample. When rasterizing a triangle, the coverage and depth tests are run N times for the N subsamples within each fragment of the triangle. If at least one of the N tests indicates that the fragment should be drawn, the pixel shader is run once. The color obtained from the pixel shader is then stored *only* into those slots that correspond to

图 11.46. 未进行抗锯齿处理的三角形栅格化。

落在三角形内的子样本。渲染完整个场景后，对超大颜色缓冲区进行下采样，以生成最终的屏幕分辨率图像。此过程涉及对每个屏幕像素的 N 个子采样槽中找到的颜色值进行平均。最终结果是经过抗锯齿处理的图像，其着色成本与未经过抗锯齿处理的图像相同。

在图 11.46 中，我们看到一个未使用抗锯齿进行光栅化的三角形。图 11.47 展示了 4 × MSAA 技术。有关 MSAA 的更多信息，请参阅 http://mynameismjp.wordpress.com/2012/10/24/msaa

-概述。

11.2.6.3 覆盖样本抗锯齿 (CSAA)

这项技术是对 Nvidia 首创的 MSAA 技术的优化。对于 4 × CSAA，像素着色器仅运行一次，深度测试和颜色存储针对每个片段的四个子采样点进行，但像素覆盖测试针对每个片段的 16 个“覆盖子采样”进行。这可以在三角形边缘产生更细粒度的颜色混合，类似于 8 × 或 16 × MSAA 的效果，但内存和 GPU 成本与 4 × MSAA 相同。

11.2.6.4 形态抗锯齿 (MLAA)

形态抗锯齿专注于校正场景中受锯齿影响最严重的区域。在 MLAA 中，场景以正常大小渲染，然后进行扫描以识别阶梯状图案。找到这些图案后，会对其进行模糊处理以降低锯齿的影响。快速近似抗锯齿 (FXAA) 是一种优化技术

Figure 11.46. Rasterizing a triangle without antialiasing.

the subsamples that fell *inside* the triangle. Once the entire scene has been rendered, the oversized color buffer is downsampled to yield the final screen-resolution image. This process involves averaging the color values found in the N subsample slots for each screen pixel. The net result is an antialiased image with a shading cost equal to that of a non-antialiased image.

In Figure 11.46 we see a triangle that has been rasterized without antialiasing. Figure 11.47 illustrates the 4× MSAA technique. For more information on MSAA, see http://mynameismjp.wordpress.com/2012/10/24/msaa-overview.

11.2.6.3 Coverage Sample Antialiasing (CSAA)

This technique is an optimization of the MSAA technique pioneered by Nvidia. For 4× CSAA, the pixel shader is run once, the depth test and color storage is done for four subsample points per fragment, but the pixel coverage test is performed for 16 “coverage subsamples” per fragment. This produces finer-grained color blending at the edges of triangles, similar to what you’d see with 8× or 16× MSAA, but at the memory and GPU cost of 4× MSAA.

11.2.6.4 Morphological Antialiasing (MLAA)

Morphological antialiasing focuses its efforts on correcting only those regions of a scene that suffer the most from the effects of aliasing. In MLAA, the scene is rendered at normal size, and then scanned in order to identify stair-stepped patterns. When these patterns are found, they are blurred to reduce the effects of aliasing. Fast approximate antialiasing (FXAA) is an optimized technique

图 11.47.多重采样抗锯齿 (MSAA)。

Figure II.47. Multisampled antialiasing (MSAA).

由 Nvidia 开发，其方法与 MLAA 类似。

有关 MLAA 的详细讨论，请参阅 https://intel.ly/2HhrQWX。FXAA 的详细说明请参见：https://bit.ly/1mIzCTv。

11.2.6.5 子像素形态抗锯齿 (SMAA)

子像素形态抗锯齿 (SMAA) 将形态抗锯齿（MLAA 和 FXAA）技术与多重采样/超级采样策略（MSAA、SSAA）相结合，以产生更精确的子像素特征。与 FXAA 类似，SMAA 是一种成本低廉的技术，但最终图像的模糊程度低于 FXAA。因此，SMAA 可以说是目前最好的抗锯齿解决方案。本书不会详细介绍该主题，但您可以在 http://www.iryoku.com/smaa/ 上了解更多关于 SMAA 的信息。

11.2.7 申请阶段

现在我们了解了 GPU 的工作原理，接下来可以讨论负责驱动 GPU 的流水线阶段——应用程序阶段。此阶段有三个角色：

1. 可见性判定。只有可见（或至少潜在可见）的对象才应该提交给 GPU，以免我们浪费宝贵的资源去处理那些永远不会被看到的三角形。

2. 将几何体提交给 GPU 进行渲染。子网格-材质对通过渲染调用（例如 DrawIndexedPrimitive()（DirectX）或 glDrawArrays()（OpenGL））或直接构建 GPU 命令列表）发送到 GPU。几何体可能会进行排序以获得最佳渲染性能。如果场景需要多次渲染，则可能需要多次提交几何体。

3. 控制着色器参数和渲染状态。通过常量寄存器传递给着色器的统一参数由应用程序阶段根据每个图元进行配置。此外，应用程序阶段必须设置所有非可编程流水线阶段的可配置参数，以确保每个图元都能得到正确的渲染。

在接下来的章节中，我们将简要探讨应用程序阶段如何执行这些任务。

11.2.7.1 能见度测定

最便宜的三角形是你永远不会绘制的三角形。因此，从场景中剔除那些对最终渲染没有贡献的物体非常重要。

developed by Nvidia that is similar to MLAA in its approach.

For a detailed discussion of MLAA, see https://intel.ly/2HhrQWX. FXAA is described in detail here: https://bit.ly/1mIzCTv.

11.2.6.5 Subpixel Morphological Antialiasing (SMAA)

Subpixel Morphological Antialiasing (SMAA) combines morphological antialiasing (MLAA and FXAA) techniques with multisampling/supersampling strategies (MSAA, SSAA) to produce more accurate subpixel features. Like FXAA, it's an inexpensive technique, but it blurs the final image less than FXAA. For these reasons, it's arguably the best AA solution available today. A detailed coverage of this topic is beyond our scope in this book, but you can read more about SMAA at http://www.iryoku.com/smaa/.

11.2.7 The Application Stage

Now that we understand how the GPU works, we can discuss the pipeline stage that is responsible for driving it—the application stage. This stage has three roles:

1. *Visibility determination*. Only objects that are visible (or at least *potentially* visible) should be submitted to the GPU, lest we waste valuable resources processing triangles that will never be seen.

2. *Submitting geometry to the GPU for rendering*. Submesh-material pairs are sent to the GPU via a rendering call like `DrawIndexedPrimitive()` (DirectX) or `glDrawArrays()` (OpenGL), or via direct construction of the GPU command list. The geometry may be sorted for optimal rendering performance. Geometry might be submitted more than once if the scene needs to be rendered in multiple passes.

3. *Controlling shader parameters and render state*. The uniform parameters passed to the shader via constant registers are configured by the application stage on a per-primitive basis. In addition, the application stage must set all of the configurable parameters of the non-programmable pipeline stages to ensure that each primitive is rendered appropriately.

In the following sections, we'll briefly explore how the application stage performs these tasks.

11.2.7.1 Visibility Determination

The cheapest triangles are the ones you never draw. So it's incredibly important to *cull* objects from the scene that do not contribute to the final rendered

在将图像提交给 GPU 之前，先构建可见网格实例列表。构建可见网格实例列表的过程称为可见性确定。

剔除切片

在视锥体剔除中，所有完全位于视锥体外部的对象都会被排除在渲染列表之外。给定一个候选网格实例，我们可以通过在对象的边界体和六个视锥体平面之间执行一些简单的测试来确定它是否位于视锥体内部。边界体通常是一个球体，因为球体特别容易剔除。对于每个视锥体平面，我们将该平面向外移动一个等于球体半径的距离，然后确定球体的中心点位于每个修改平面的哪一侧。如果发现球体位于所有六个修改平面的正面，则该球体位于视锥体内部。

实际上，我们不需要真正移动截头平面。回想一下公式 (5.13)，从一个点到一个平面的垂直距离 h 可以通过将该点直接代入平面方程来计算，如下所示：h = αx + by + cz + d = n · P − n · P 0（参见 5.6.3 节）。因此，我们需要做的就是将边界球的中心点代入每个截头平面的平面方程中，得到每个平面 i 的 hi 值，然后我们可以将 hi 值与边界球的半径进行比较，以确定它是否位于每个平面内。

场景图数据结构（如第 11.2.7.4 节中所述）可以让我们忽略边界球远离视锥体内部的物体，从而帮助优化视锥体剔除。

遮挡和潜在可见集

即使物体完全位于视锥体内，它们也可能相互遮挡。将完全被其他物体遮挡的物体从可见列表中移除的过程称为遮挡剔除。在从地面观看的拥挤环境中，物体间可能会存在大量遮挡，因此遮挡剔除至关重要。在不那么拥挤的场景中，或者从上方观看场景时，遮挡可能要少得多，遮挡剔除的成本可能超过其收益。

通过预先计算潜在可见集 (PVS)，可以对大规模环境进行粗略遮挡剔除。对于任何给定的摄像机有利位置，PVS 都会列出可能可见的场景对象。PVS 倾向于包含实际上不可见的对象，而不是排除那些实际上会对渲染场景有所贡献的对象。

实现 PVS 系统的一种方法是将关卡划分成各种区域。每个区域都可以提供一个其他区域的列表

image prior to submitting them to the GPU. The process of constructing the list of visible mesh instances is known as *visibility determination*.

Frustum Culling

In frustum culling, all objects that lie entirely outside the frustum are excluded from our render list. Given a candidate mesh instance, we can determine whether or not it lies inside the frustum by performing some simple tests between the object's *bounding volume* and the six frustum planes. The bounding volume is usually a sphere, because spheres are particularly easy to cull. For each frustum plane, we move the plane outward a distance equal to the radius of the sphere, then we determine on which side of each modified plane the center point of the sphere lies. If the sphere is found to be on the front side of all six modified planes, the sphere is inside the frustum.

In practice, we don't need to actually move the frustum planes. Recall from Equation (5.13) that the perpendicular distance h from a point to a plane can be calculated by plugging the point directly into the plane equation as follows: $h = ax + by + cz + d = \mathbf{n} \cdot \mathbf{P} - \mathbf{n} \cdot \mathbf{P}_0$ (see Section 5.6.3). So all we need to do is plug the center point of our bounding sphere into the plane equations for each frustum plane, giving us a value of h_i for each plane i, and then we can compare the h_i values to the radius of the bounding sphere to determine whether or not it lies inside each plane.

A scene graph data structure, described in Section 11.2.7.4, can help optimize frustum culling by allowing us to ignore objects whose bounding spheres are nowhere close to being inside the frustum.

Occlusion and Potentially Visible Sets

Even when objects lie entirely within the frustum, they may occlude one another. Removing objects from the visible list that are entirely occluded by other objects is called *occlusion culling*. In crowded environments viewed from ground level, there can be a great deal of inter-object occlusion, making occlusion culling extremely important. In less crowded scenes, or when scenes are viewed from above, much less occlusion may be present, and the cost of occlusion culling may outweigh its benefits.

Gross occlusion culling of a large-scale environment can be done by precalculating a *potentially visible set* (PVS). For any given camera vantage point, a PVS lists those scene objects that might be visible. A PVS errs on the side of including objects that aren't actually visible, rather than excluding objects that actually would have contributed to the rendered scene.

One way to implement a PVS system is to chop the level up into regions of some kind. Each region can be provided with a list of the other regions

当摄像机位于其中时可以看到。这些 PVS 可能由艺术家或游戏设计师手动指定。更常见的是，自动离线工具会根据用户指定的区域生成 PVS。此类工具通常通过从区域内随机分布的各个有利位置渲染场景来运行。每个区域的几何形状都经过颜色编码，因此可以通过扫描生成的帧缓冲区并将找到的区域颜色制成表格来找到可见区域列表。由于自动 PVS 工具并不完善，它们通常为用户提供一种机制来调整结果，方法是手动放置有利位置进行测试，或者手动指定应明确包含或排除在特定区域 PVS 之外的区域列表。

门户网站

确定场景哪些部分可见的另一种方法是使用门户。在门户渲染中，游戏世界被划分为半封闭的区域，这些区域通过孔洞（例如窗户和门口）相互连接。这些孔洞被称为门户。它们通常用描述其边界的多边形来表示。

要渲染包含入口的场景，我们首先渲染包含摄像机的区域。然后，对于区域中的每个入口，我们扩展一个类似视锥体的体积，该体积由从摄像机焦点延伸到入口边界多边形每条边的平面组成。相邻区域的内容可以按照与摄像机视锥体完全相同的方式剔除到该入口体积中。这确保只有相邻区域中可见的几何体才会被渲染。图 11.48 展示了这种技术。

遮挡体积（反门户）

如果我们颠倒门户的概念，金字塔体积也可以用来描述场景中因被物体遮挡而无法看到的区域。这些体积被称为遮挡体积或反门户。为了构建遮挡体积，我们找到每个遮挡物体的轮廓边缘，并从相机焦点向外延伸穿过这些边缘的平面。我们会根据这些遮挡体积测试更远的物体，如果它们完全位于遮挡区域内，则将其剔除。

如图 11.49 所示。

门户最适合用于渲染封闭的室内环境，其中“房间”之间的窗户和门道数量相对较少。在这种场景中，门户占总面积的比例相对较小。

that can be seen when the camera is inside it. These PVSs might be manually specified by the artists or game designers. More commonly, an automated offline tool generates the PVS based on user-specified regions. Such a tool usually operates by rendering the scene from various randomly distributed vantage points within a region. Every region's geometry is color coded, so the list of visible regions can be found by scanning the resulting frame buffer and tabulating the region colors that are found. Because automated PVS tools are imperfect, they typically provide the user with a mechanism for tweaking the results, either by manually placing vantage points for testing, or by manually specifying a list of regions that should be explicitly included or excluded from a particular region's PVS.

Portals

Another way to determine what portions of a scene are visible is to use *portals*. In portal rendering, the game world is divided up into semiclosed regions that are connected to one another via holes, such as windows and doorways. These holes are called portals. They are usually represented by polygons that describe their boundaries.

To render a scene with portals, we start by rendering the region that contains the camera. Then, for each portal in the region, we extend a frustum-like volume consisting of planes extending from the camera's focal point through each edge of the portal's bounding polygon. The contents of the neighboring region can be culled to this portal volume in exactly the same way geometry is culled against the camera frustum. This ensures that only the visible geometry in the adjacent regions will be rendered. Figure 11.48 provides an illustration of this technique.

Occlusion Volumes (*Antiportals*)

If we flip the portal concept on its head, pyramidal volumes can also be used to describe regions of the scene that *cannot* be seen because they are being occluded by an object. These volumes are known as *occlusion volumes* or *antiportals*. To construct an occlusion volume, we find the silhouette edges of each occluding object and extend planes outward from the camera's focal point through each of these edges. We test more-distant objects against these occlusion volumes and cull them if they lie entirely within the occlusion region. This is illustrated in Figure 11.49.

Portals are best used when rendering enclosed indoor environments with a relatively small number of windows and doorways between "rooms." In this kind of scene, the portals occupy a relatively small percentage of the total

图 11.48。入口用于定义类似视锥体的体积，用于剔除相邻区域的内容。在此示例中，对象 A、B 和 D 将被剔除，因为它们位于其中一个入口之外；其他对象将可见。

摄像机视锥的体积，导致大量位于入口之外的物体可以被剔除。反入口最适合用于大型户外环境，在这些环境中，附近的物体通常会遮挡摄像机视锥的大片区域。在这种情况下，反入口占据了摄像机视锥总体积的较大比例，导致大量物体被剔除。

图 11.49。由于对象 A、B 和 C 对应的反门户，对象 D、E、F 和 G 被剔除。因此，只有 A、B、C 和 H 可见。

Figure 11.48. Portals are used to define frustum-like volumes, which are used to cull the contents of neighboring regions. In this example, objects A, B and D will be culled because they lie outside one of the portals; the other objects will be visible.

volume of the camera frustum, resulting in a large number of objects outside the portals that can be culled. Antiportals are best applied to large outdoor environments, in which nearby objects often occlude large swaths of the camera frustum. In this case, the antiportals occupy a relatively large percentage of the total camera frustum volume, resulting in large numbers of culled objects.

Figure 11.49. As a result of the antiportals corresponding to objects A, B and C, objects D, E, F and G are culled. Therefore, only A, B, C and H are visible.

11.2.7.2 原始提交

一旦生成了可见几何图元列表，就必须将各个图元提交到 GPU 管线进行渲染。这可以通过调用 DirectX 中的 DrawIndexedPrimitive() 或 OpenGL 中的 glDrawArrays() 来实现。

渲染状态

正如我们在 11.2.4 节中了解到的，GPU 流水线中许多阶段的功能是固定的，但可以配置。即使是可编程阶段，其功能也部分由可配置参数驱动。以下列出了一些可配置参数的示例（但这绝不是完整的列表）：

- 世界观矩阵；
- 光方向矢量；
- 纹理绑定（即，对于给定的材质/着色器使用哪些纹理）；
- 纹理寻址和过滤模式；
- 滚动纹理和其他动画效果的时间基准；
- z 检验（启用或禁用）；以及
- alpha 混合选项。

GPU 管线中所有可配置参数的集合称为 硬件状态 或 渲染状态 。应用程序阶段负责确保每个提交的图元的硬件状态都得到正确且完整的配置。理想情况下，这些状态设置由与每个子网格关联的材质完整描述。因此，应用程序阶段的工作归结为：遍历可见网格实例列表，遍历每个子网格-材质对，根据材质的规范设置渲染状态，然后调用低级图元提交函数（DrawIndexedPrimitive()、glDrawArrays() 或类似函数）。

国家泄密

如果我们忘记在提交的图元之间设置渲染状态的某些方面，则前一个图元上使用的设置将“泄漏”到新的图元上。例如，渲染状态泄漏可能表现为对象具有错误的纹理或不正确的光照效果。显然，应用程序阶段绝不允许发生状态泄漏至关重要。

11.2.7.2 Primitive Submission

Once a list of visible geometric primitives has been generated, the individual primitives must be *submitted* to the GPU pipeline for rendering. This can be accomplished by making calls to `DrawIndexedPrimitive()` in DirectX or `glDrawArrays()` in OpenGL.

Render State

As we learned in Section 11.2.4, the functionality of many of the GPU pipeline's stages is fixed but configurable. And even programmable stages are driven in part by configurable parameters. Some examples of these configurable parameters are listed below (although this is by no means a complete list):

- world-view matrix;
- light direction vectors;
- texture bindings (i.e., which textures to use for a given material/shader);
- texture addressing and filtering modes;
- time base for scrolling textures and other animated effects;
- *z*-test (enabled or disabled); and
- alpha blending options.

The set of all configurable parameters within the GPU pipeline is known as the *hardware state* or *render state*. It is the application stage's responsibility to ensure that the hardware state is configured properly and completely for each submitted primitive. Ideally these state settings are described completely by the material associated with each submesh. So the application stage's job boils down to iterating through the list of visible mesh instances, iterating over each submesh-material pair, setting the render state based on the material's specifications and then calling the low-level primitive submission functions (`DrawIndexedPrimitive()`, `glDrawArrays()`, or similar).

State Leaks

If we forget to set some aspect of the render state between submitted primitives, the settings used on the previous primitive will "leak" over onto the new primitive. A *render state leak* might manifest itself as an object with the wrong texture or an incorrect lighting effect, for example. Clearly it's important that the application stage never allow state leaks to occur.

GPU 命令列表

应用程序阶段实际上通过命令列表与 GPU 通信。这些命令将渲染状态设置与需要绘制的几何体的引用交织在一起。例如，要使用材质 1 渲染对象 A 和 B，然后使用材质 2 渲染对象 C、D 和 E，命令列表可能如下所示：

- 设置材质 1 的渲染状态（多个命令，每个渲染状态设置一个命令）。
- 提交原语A。
- 提交原语B。
- 设置材质 2 的渲染状态（多个命令）。
- 提交原语C。
- 提交原语D。
- 提交原始 E。

底层 API 函数（例如 DrawIndexedPrimitive()）实际上只是构建并提交 GPU 命令列表。这些 API 调用本身的成本对于某些应用程序来说可能过高。为了最大限度地提高性能，一些游戏引擎会手动构建 GPU 命令列表，或者通过调用 Vulkan 等低级渲染 API（https://www.khronos.org/vulkan/）来构建。

11.2.7.3 几何排序

渲染状态设置是全局的——它们会应用于整个 GPU。因此，要更改渲染状态设置，必须先刷新整个 GPU 管线，然后才能应用新设置。如果管理不善，这可能会导致严重的性能下降。

显然，我们希望尽可能少地更改渲染设置。实现这一点的最佳方法是按材质对几何体进行排序。这样，我们可以安装材质 A 的设置，渲染与材质 A 相关的所有几何体，然后再渲染材质 B。

遗憾的是，按材质对几何体进行排序可能会对渲染性能产生不利影响，因为它会增加过度绘制（overdraw）——即同一像素被多个重叠三角形多次填充的情况。当然，一定的过度绘制是必要的，也是可取的，因为这是将透明和半透明表面正确地 Alpha 混合到场景中的唯一方法。然而，不透明像素的过度绘制始终会浪费 GPU 带宽。

早期 z 测试旨在在昂贵的像素着色器执行之前丢弃被遮挡的片段。但为了最大限度地利用早期 z 测试，我们需要按从前到后的顺序绘制三角形。这样，

The GPU Command List

The application stage actually communicates with the GPU via a command list. These commands interleave render state settings with references to the geometry that should be drawn. For example, to render objects A and B with material 1, followed by objects C, D and E using material 2, the command list might look like this:

- Set render state for material 1 (multiple commands, one per render state setting).
- Submit primitive A.
- Submit primitive B.
- Set render state for material 2 (multiple commands).
- Submit primitive C.
- Submit primitive D.
- Submit primitive E.

Under the hood, API functions like `DrawIndexedPrimitive()` actually just construct and submit GPU command lists. The cost of these API calls can themselves be too high for some applications. To maximize performance, some game engines build GPU command lists manually or by calling a low-level rendering API like Vulkan (https://www.khronos.org/vulkan/).

11.2.7.3 Geometry Sorting

Render state settings are global—they apply to the entire GPU as a whole. So in order to change render state settings, the entire GPU pipeline must be flushed before the new settings can be applied. This can cause massive performance degradation if not managed carefully.

Clearly we'd like to change render settings as infrequently as possible. The best way to accomplish this is to sort our geometry by material. That way, we can install material A's settings, render all geometry associated with material A and then move on to material B.

Unfortunately, sorting geometry by material can have a detrimental effect on rendering performance because it increases *overdraw*—a situation in which the same pixel is filled multiple times by multiple overlapping triangles. Certainly some overdraw is necessary and desirable, as it is the only way to properly alpha-blend transparent and translucent surfaces into a scene. However, overdraw of *opaque* pixels is always a waste of GPU bandwidth.

The early z-test is designed to discard occluded fragments before the expensive pixel shader has a chance to execute. But to take maximum advantage of early z, we need to draw the triangles in front-to-back order. That way, the

最近的三角形将立即填充 z 缓冲区，并且来自其后方较远三角形的所有片段都可以被快速丢弃，几乎不会出现过度绘制。

z-Prepass 救援

我们如何才能协调按材质对几何体进行排序的需求，以及按从前到后的顺序渲染不透明几何体的需求呢？答案在于 GPU 的一项名为 z-prepass 的功能。

z-prepass 背后的理念是渲染场景两次：第一次尽可能高效地生成 z-buffer 的内容；第二次用全彩信息填充帧缓冲区（但这次由于 z-buffer 的内容，不会出现过度绘制）。GPU 提供了一种特殊的双倍速渲染模式，在该模式下，像素着色器被禁用，只更新 z-buffer。在此阶段，不透明几何体可以按从前到后的顺序渲染，以最大限度地缩短生成 z-buffer 内容所需的时间。然后，几何体可以重新按材质顺序排列，并以全彩渲染，同时尽量减少状态变化，从而实现最大的流水线吞吐量。

不透明几何体渲染完成后，透明表面可以按从后到前的顺序绘制。这种强力方法使我们能够获得正确的 alpha 混合结果。顺序无关透明度 (OIT) 是一种允许以任意顺序渲染透明几何体的技术。它的工作原理是每个像素存储多个片段，对每个像素的片段进行排序，并仅在整个场景渲染完成后才进行混合。这种技术无需预先对几何体进行排序即可产生正确的结果，但它的内存成本很高，因为帧缓冲区必须足够大才能存储每个像素的所有半透明片段。

11.2.7.4 场景图

现代游戏世界可能非常庞大。大多数场景中的大部分几何体并不位于相机视锥体内，因此显式地剔除所有这些对象通常非常浪费资源。因此，我们希望设计一个数据结构来管理场景中的所有几何体，并允许我们在执行详细的视锥体剔除之前快速丢弃远离相机视锥体的大量世界元素。理想情况下，该数据结构还应帮助我们对场景中的几何体进行排序，例如，在 z 预渲染阶段按从前到后的顺序排序，或在全彩渲染阶段按材质顺序排序。

这种数据结构通常被称为场景图，指的是电影渲染引擎和 Maya 等 DCC 工具经常使用的类似图形的数据结构。然而，游戏的场景图实际上不一定是图，实际上，选择的数据结构通常是某种树（当然，

closest triangles will fill the *z*-buffer right off the bat, and all of the fragments coming from more-distant triangles behind them can be quickly discarded, with little or no overdraw.

z-Prepass to the Rescue

How can we reconcile the need to sort geometry by material with the conflicting need to render opaque geometry in a front-to-back order? The answer lies in a GPU feature known as *z-prepass*.

The idea behind *z*-prepass is to render the scene twice: the first time to generate the contents of the *z*-buffer as efficiently as possible and the second time to populate the frame buffer with full color information (but this time with no overdraw, thanks to the contents of the *z*-buffer). The GPU provides a special double-speed rendering mode in which the pixel shaders are disabled, and only the *z*-buffer is updated. Opaque geometry can be rendered in front-to-back order during this phase, to minimize the time required to generate the *z*-buffer contents. Then the geometry can be resorted into material order and rendered in full color with minimal state changes for maximum pipeline throughput.

Once the opaque geometry has been rendered, transparent surfaces can be drawn in back-to-front order. This brute-force method allows us to achieve the proper alpha-blended result. *Order-independent transparency* (OIT) is a technique that permits transparent geometry to be rendered in an arbitrary order. It works by storing multiple fragments per pixel, sorting each pixel's fragments and blending them only after the entire scene has been rendered. This technique produces correct results without the need for pre-sorting the geometry, but it comes at a high memory cost because the frame buffer must be large enough to store all of the translucent fragments for each pixel.

11.2.7.4 Scene Graphs

Modern game worlds can be very large. The majority of the geometry in most scenes does not lie within the camera frustum, so frustum culling all of these objects explicitly is usually incredibly wasteful. Instead, we would like to devise a data structure that manages all of the geometry in the scene and allows us to quickly discard large swaths of the world that are nowhere near the camera frustum prior to performing detailed frustum culling. Ideally, this data structure should also help us to sort the geometry in the scene, either in front-to-back order for the *z*-prepass or in material order for full-color rendering.

Such a data structure is often called a *scene graph*, in reference to the graph-like data structures often used by film rendering engines and DCC tools like Maya. However, a game's scene graph needn't actually be a graph, and in fact the data structure of choice is usually some kind of tree (which is, of course,

图（一种特殊的图）。大多数此类数据结构背后的基本思想是将三维空间进行划分，以便于丢弃与视锥体不相交的区域，而无需逐个剔除其中的所有单个对象。例如四叉树和八叉树、BSP 树、k-d 树和空间哈希技术。

四叉树和八叉树

四叉树递归地将空间划分为多个象限。每一级递归都由四叉树中的一个节点表示，该节点有四个子节点，每个子节点代表一个象限。这些象限通常由垂直方向、与轴对齐的平面分隔，因此这些象限呈正方形或矩形。然而，有些四叉树使用任意形状的区域来细分空间。

四叉树几乎可以用于存储和组织任何类型的空间分布数据。在渲染引擎中，为了实现高效的视锥体剔除，四叉树通常用于存储可渲染图元，例如网格实例、地形几何体的子区域或大型静态网格的单个三角形。可渲染图元存储在树的叶子节点中，我们通常力求在每个叶子节点区域内实现图元数量的大致均匀。这可以通过根据区域内图元的数量来决定是否继续或终止细分来实现。

为了确定哪些图元在相机视锥体内可见，我们会从根节点遍历树形结构到叶子节点，检查每个区域是否与视锥体相交。如果某个给定的象限与视锥体不相交，则说明其子区域也不会与视锥体相交，因此可以停止遍历该分支。这使我们能够比线性搜索更快地搜索潜在可见的图元（通常为 O(logn) 时间复杂度）。图 11.50 展示了一个四叉树空间细分的示例。

八叉树是四叉树的三维等价物，在递归细分的每一层将空间划分为八个子区域。八叉树的区域通常是立方体或长方体，但通常可以是任意形状的三维区域。

边界球树

与四叉树或八叉树将空间细分为（通常）矩形区域的方式相同，边界球树将空间按层次划分为球形区域。树的叶子节点包含场景中可渲染图元的边界球。我们将这些图元收集到小的逻辑

a special case of a graph). The basic idea behind most of these data structures is to partition three-dimensional space in a way that makes it easy to discard regions that do not intersect the frustum, without having to frustum cull all of the individual objects within them. Examples include quadtrees and octrees, BSP trees, *k*d-trees and spatial hashing techniques.

Quadtrees and Octrees

A *quadtree* divides space into quadrants recursively. Each level of recursion is represented by a node in the quadtree with four children, one for each quadrant. The quadrants are typically separated by vertically oriented, axis-aligned planes, so that the quadrants are square or rectangular. However, some quadtrees subdivide space using arbitrarily shaped regions.

Quadtrees can be used to store and organize virtually any kind of spatially distributed data. In the context of rendering engines, quadtrees are often used to store renderable primitives such as mesh instances, subregions of terrain geometry or individual triangles of a large static mesh, for the purposes of efficient frustum culling. The renderable primitives are stored at the leaves of the tree, and we usually aim to achieve a roughly uniform number of primitives within each leaf region. This can be achieved by deciding whether to continue or terminate the subdivision based on the number of primitives within a region.

To determine which primitives are visible within the camera frustum, we walk the tree from the root to the leaves, checking each region for intersection with the frustum. If a given quadrant does not intersect the frustum, then we know that none of its child regions will do so either, and we can stop traversing that branch of the tree. This allows us to search for potentially visible primitives much more quickly than would be possible with a linear search (usually in $O(\log n)$ time). An example of a quadtree subdivision of space is shown in Figure 11.50.

An *octree* is the three-dimensional equivalent of a quadtree, dividing space into eight subregions at each level of the recursive subdivision. The regions of an octree are often cubes or rectangular prisms but can be arbitrarily shaped three-dimensional regions in general.

Bounding Sphere Trees

In the same way that a quadtree or octree subdivides space into (usually) rectangular regions, a *bounding sphere tree* divides space into spherical regions hierarchically. The leaves of the tree contain the bounding spheres of the renderable primitives in the scene. We collect these primitives into small logical

图 11.50. 基于每个区域一个点的标准，将空间自上而下地划分为递归象限，以便存储在四叉树中。

组，并计算每个组的净边界球。这些组本身会被收集到更大的组中，这个过程持续下去，直到我们得到一个包含整个虚拟世界的边界球的组。为了生成潜在可见图元的列表，我们从根节点遍历树到叶子节点，将每个边界球与视锥体进行测试，并仅递归遍历与其相交的分支。

BSP树

二叉空间划分 (BSP) 树递归地将空间一分为二，直到每个半空间中的对象满足某些预定义的条件（类似于四叉树将空间划分为象限）。BSP 树用途广泛，包括碰撞检测和构造立体几何，其最著名的应用是提高 3D 图形中视锥体剔除和几何排序的性能。k d 树是 BSP 树概念在 k 维上的推广。

在渲染环境中，BSP 树在递归的每一层级上用一个平面划分空间。划分平面可以与轴对齐，但更常见的是，每个细分平面对应于场景中单个三角形的平面。然后，所有其他三角形被分类为位于平面的正面或背面。任何与划分平面相交的三角形本身都会被划分为三个新的三角形，使得每个三角形要么完全位于平面的前方，要么完全位于平面的后方，要么与平面共面。结果是一个二叉树，它包含一个划分平面，每个内部节点包含一个或多个三角形，叶子节点包含多个三角形。

Figure 11.50. A top-down view of a space divided recursively into quadrants for storage in a quadtree, based on the criterion of one point per region.

groups and calculate the net bounding sphere of each group. The groups are themselves collected into larger groups, and this process continues until we have a single group with a bounding sphere that encompasses the entire virtual world. To generate a list of potentially visible primitives, we walk the tree from the root to the leaves, testing each bounding sphere against the frustum, and only recursing down branches that intersect it.

BSP Trees

A binary space partitioning (BSP) tree divides space in half recursively until the objects within each half-space meet some predefined criteria (much as a quadtree divides space into quadrants). BSP trees have numerous uses, including collision detection and constructive solid geometry, as well as their most well-known application as a method for increasing the performance of frustum culling and geometry sorting for 3D graphics. A *k*d-tree is a generalization of the BSP tree concept to *k* dimensions.

In the context of rendering, a BSP tree divides space with a single plane at each level of the recursion. The dividing planes can be axis-aligned, but more commonly each subdivision corresponds to the plane of a single triangle in the scene. All of the other triangles are then categorized as being either on the front side or the back side of the plane. Any triangles that intersect the dividing plane are themselves divided into three new triangles, so that every triangle lies either entirely in front of or entirely behind the plane, or is coplanar with it. The result is a binary tree with a dividing plane and one or more triangles at each interior node and triangles at the leaves.

BSP 树可以像四叉树、八叉树或边界球树一样用于视锥体剔除。然而，当如上所述使用单个三角形生成时，BSP 树也可以用来将三角形严格按照从后到前或从前到后的顺序排序。这对于像《毁灭战士》这样的早期 3D 游戏尤为重要，因为这些游戏没有 αz 缓冲区的优势，因此被迫使用画家算法（即从后到前渲染场景）来确保正确的三角形间遮挡。

给定 3D 空间中的摄像机视点，从后向前的排序算法会从根节点开始遍历树。在每个节点，我们检查视点是位于该节点分割平面的前方还是后方。如果摄像机位于节点分割平面的前方，我们首先访问该节点的后方子节点，然后绘制与其分割平面共面的三角形，最后访问其前方子节点。同样，如果发现摄像机视点位于节点分割平面的后方，我们首先访问该节点的前方子节点，然后绘制与该节点分割平面共面的三角形，最后访问其后方子节点。这种遍历方案确保先访问距离摄像机最远的三角形，然后再访问距离摄像机较近的三角形，从而实现从后向前的排序。由于该算法会遍历场景中的所有三角形，因此遍历顺序与摄像机的观察方向无关。为了仅遍历可见三角形，需要进行二次视锥剔除步骤。图 11.51 显示了一个简单的 BSP 树，以及针对所示摄像机位置进行的树遍历。

全面介绍 BSP 树的生成和使用算法超出了本文的范围。请参阅 http://www.gamedev.net/reference/articles/article657.asp

图 11.51。BSP 树中三角形从后向前遍历的示例。为了简单起见，三角形在二维中以边为单位显示，但在真实的 BSP 树中，三角形和分割平面在空间中的方向是任意的。

A BSP tree can be used for frustum culling in much the same way a quadtree, octree or bounding sphere tree can. However, when generated with individual triangles as described above, a BSP tree can also be used to sort triangles into a strictly back-to-front or front-to-back order. This was particularly important for early 3D games like *Doom*, which did not have the benefit of a z-buffer and so were forced to use the painter's algorithm (i.e., to render the scene from back to front) to ensure proper inter-triangle occlusion.

Given a camera view point in 3D space, a back-to-front sorting algorithm walks the tree from the root. At each node, we check whether the view point is in front of or behind that node's dividing plane. If the camera is in front of a node's plane, we visit the node's back children first, then draw any triangles that are coplanar with its dividing plane, and finally we visit its front children. Likewise, when the camera's view point is found to be behind a node's dividing plane, we visit the node's front children first, then draw the triangles coplanar with the node's plane and finally we visit its back children. This traversal scheme ensures that the triangles farthest from the camera will be visited before those that are closer to it, and hence it yields a back-to-front ordering. Because this algorithm traverses *all* of the triangles in the scene, the order of the traversal is independent of the direction the camera is looking. A secondary frustum culling step would be required in order to traverse only visible triangles. A simple BSP tree is shown in Figure 11.51, along with the tree traversal that would be done for the camera position shown.

Full coverage of BSP tree generation and usage algorithms is beyond our scope here. See http://www.gamedev.net/reference/articles/article657.asp

Figure 11.51. An example of back-to-front traversal of the triangles in a BSP tree. The triangles are shown edge-on in two dimensions for simplicity, but in a real BSP tree the triangles and dividing planes would be arbitrarily oriented in space.

有关 BSP 树的更多详细信息。

11.2.7.5 选择场景图

显然，场景图有很多种。为你的游戏选择哪种数据结构取决于你期望渲染的场景的性质。为了做出明智的选择，你必须清楚地了解在为特定游戏渲染场景时需要什么，更重要的是，不需要什么。

例如，如果您正在开发一款格斗游戏，其中两个角色在一个几乎静态的环境中围绕一个环形区域进行战斗，那么您可能根本不需要太多的场景图。如果您的游戏主要发生在封闭的室内环境中，那么 BSP 树或传送门系统可能就很适合您。如果动作发生在相对平坦的户外地形上，并且场景主要从上方观看（例如在策略游戏或上帝游戏中），那么一个简单的四叉树可能就足以实现高速渲染。另一方面，如果户外场景主要从地面上某人的视角观看，我们可能需要额外的剔除机制。密集场景可以从遮挡体（反传送门）系统中受益，因为会有大量遮挡物。另一方面，如果您的户外场景非常稀疏，添加反传送门系统可能不会带来好处（甚至可能会影响您的帧速率）。

最终，场景图的选择应该基于通过实际测量渲染引擎性能获得的硬数据。你可能会惊讶地发现所有循环的实际运行情况！但一旦了解了这些情况，你就可以选择场景图数据结构和/或其他优化方案来针对具体问题。

11.3 高级照明和全局照明

为了渲染照片级逼真的场景，我们需要物理上精确的全局光照算法。全面介绍这些技术超出了我们的范围。在接下来的章节中，我们将简要概述当今游戏行业中最流行的技术。我们的目标是让您了解这些技术，并为进一步研究奠定基础。有关该主题的深入探讨，请参阅[10]。

11.3.1 基于图像的照明

许多先进的照明和着色技术大量使用图像数据，通常以二维纹理贴图的形式呈现。这些技术包括

for more details on BSP trees.

11.2.7.5 Choosing a Scene Graph

Clearly there are many different kinds of scene graphs. Which data structure to select for your game will depend upon the nature of the scenes you expect to be rendering. To make the choice wisely, you must have a clear understanding of what is required—and more importantly what is *not* required—when rendering scenes for your particular game.

For example, if you're implementing a fighting game, in which two characters battle it out in a ring surrounded by a mostly static environment, you may not need much of a scene graph at all. If your game takes place primarily in enclosed indoor environments, a BSP tree or portal system may serve you well. If the action takes place outdoors on relatively flat terrain, and the scene is viewed primarily from above (as might be the case in a strategy game or god game), a simple quadtree might be all that's required to achieve high rendering speeds. On the other hand, if an outdoor scene is viewed primarily from the point of view of someone on the ground, we may need additional culling mechanisms. Densely populated scenes can benefit from an occlusion volume (antiportal) system, because there will be plenty of occluders. On the other hand, if your outdoor scene is very sparse, adding an antiportal system probably won't pay dividends (and might even hurt your frame rate).

Ultimately, your choice of scene graph should be based on hard data obtained by actually measuring the performance of your rendering engine. You may be surprised to learn where all your cycles are actually going! But once you know, you can select scene graph data structures and/or other optimizations to target the specific problems at hand.

11.3 Advanced Lighting and Global Illumination

In order to render photorealistic scenes, we need physically accurate global illumination algorithms. A complete coverage of these techniques is beyond our scope. In the following sections, we will briefly outline the most prevalent techniques in use within the game industry today. Our goal here is to provide you with an awareness of these techniques and a jumping-off point for further investigation. For an excellent in-depth coverage of this topic, see [10].

11.3.1 Image-Based Lighting

A number of advanced lighting and shading techniques make heavy use of image data, usually in the form of two-dimensional texture maps. These are

图 11.52. 法线贴图表面的示例。

称为基于图像的照明算法。

11.3.1.1 法线贴图

法线贴图指定了每个纹素的表面法线方向向量。这使得 3D 建模人员能够向渲染引擎提供表面形状的高精度描述，而无需对模型进行高阶曲面细分（如果通过顶点法线提供相同的信息，则需要进行高阶曲面细分）。使用法线贴图，可以使单个平面三角形看起来像是由数百万个小三角形构成的。图 11.52 展示了一个法线贴图的示例。

法向量通常编码在纹理的 RGB 颜色通道中，并带有适当的偏差，以克服 RGB 通道严格为正而法向量分量可以为负这一事实。有时纹理中只存储两个坐标；假设表面法线为单位向量，则第三个坐标可以在运行时轻松计算。

11.3.1.2 高度图：凹凸贴图、视差贴图和位移贴图

顾名思义，高度图编码了理想表面在三角形表面上方或下方的高度。高度图通常编码为灰度图像，因为我们只需要每个纹素一个高度值。高度图可用于凹凸贴图、视差遮挡贴图和位移贴图——这三种技术可以使平面看起来具有高度变化。

在凹凸贴图（Bump Mapping）中，高度图是一种生成表面法线的低成本方法。这种技术主要用于早期的 3D 图形技术——如今，大多数游戏引擎都会存储表面法线信息。

Figure 11.52. An example of a normal-mapped surface.

called *image-based lighting* algorithms.

11.3.1.1 Normal Mapping

A *normal map* specifies a surface normal direction vector at each texel. This allows a 3D modeler to provide the rendering engine with a highly detailed description of a surface's shape, without having to tessellate the model to a high degree (as would be required if this same information were to be provided via vertex normals). Using a normal map, a single flat triangle can be made to look as though it were constructed from millions of tiny triangles. An example of normal mapping is shown in Figure 11.52.

The normal vectors are typically encoded in the RGB color channels of the texture, with a suitable bias to overcome the fact that RGB channels are strictly positive while normal vector components can be negative. Sometimes only two coordinates are stored in the texture; the third can be easily calculated at runtime, given the assumption that the surface normals are unit vectors.

11.3.1.2 Heightmaps: Bump, Parallax and Displacement Mapping

As its name implies, a *heightmap* encodes the height of the ideal surface above or below the surface of the triangle. Heightmaps are typically encoded as grayscale images, since we only need a single height value per texel. Heightmaps can be used for *bump mapping*, *parallax occlusion mapping* and *displacement mapping*—three techniques that can make a planar surface appear to have height variation.

In bump mapping, a heightmap is used as a cheap way to generate surface normals. This technique was primarily used in the early days of 3D graphics—nowadays, most game engines store surface normal information

图 11.53. 凹凸贴图（左）、视差遮挡贴图（中）和位移贴图（右）的比较。

明确地在法线贴图中，而不是从高度图计算法线。

视差遮挡贴图利用高度图中的信息来人为调整渲染平面时使用的纹理坐标，使表面看起来包含随着摄像机移动而半正确移动的表面细节。（顽皮狗公司曾使用此技术制作《神秘海域》系列游戏中的子弹撞击贴花。）位移贴图（也称为浮雕贴图）通过实际细分并挤压表面多边形来生成真实的表面细节，同样使用高度图来确定每个顶点的位移量。由于生成的是真实的几何体，因此这种贴图可以产生最令人信服的效果——能够正确地进行自遮挡和自阴影。图 11.53 比较了凹凸贴图、视差贴图和位移贴图。图 11.54 展示了在 DirectX 9 中实现的位移贴图示例。

11.3.1.3 镜面/光泽贴图

当光线直接从光滑表面反射时，我们称之为镜面反射。镜面反射的强度取决于观察者、光源和表面法线之间的相对角度。正如我们在 11.1.3.2 节中看到的，镜面反射强度的形式为 kS(R · V)α，其中 R 是光线方向矢量关于表面法线的反射，V 是朝向观察者的方向，kS 是表面的整体镜面反射率，α 称为镜面反射强度。

许多表面并非均匀地呈现光泽。例如，当人的脸部汗湿且脏污时，湿润区域会显得有光泽，而干燥或脏污的区域则会显得暗淡。我们可以将高细节的镜面反射信息编码到一种特殊的纹理贴图中，称为镜面反射贴图。

如果我们将 k S 的值存储在镜面反射贴图的纹理像素中，我们就可以控制每个纹理像素上应用的镜面反射量。这种镜面反射贴图有时被称为光泽贴图。它也被称为镜面反射遮罩，因为零值纹理像素可以用来“遮盖”表面中我们不想应用镜面反射的区域。如果我们存储

Figure 11.53. Comparison of bump mapping (left), parallax occlusion mapping (center) and displacement mapping (right).

explicitly in a *normal map*, rather than calculating the normals from a heightmap.

Parallax occlusion mapping uses the information in a heightmap to artificially adjust the texture coordinates used when rendering a flat surface, in such a way as to make the surface appear to contain surface details that move semi-correctly as the camera moves. (This technique was used to produce the bullet impact decals in the *Uncharted* series of games by Naughty Dog.)

Displacement mapping (also known as relief mapping) produces real surface details by actually tessellating and then extruding surface polygons, again using a heightmap to determine how much to displace each vertex. This produces the most convincing effect—one that properly self-occludes and self-shadows—because real geometry is being generated. Figure 11.53 compares bump mapping, parallax mapping and displacement mapping. Figure 11.54 shows an example of displacement mapping implemented in DirectX 9.

11.3.1.3 Specular/Gloss Maps

When light reflects directly off a shiny surface, we call this *specular* reflection. The intensity of a specular reflection depends on the relative angles of the viewer, the light source and the surface normal. As we saw in Section 11.1.3.2, the specular intensity takes the form $k_S(\mathbf{R} \cdot \mathbf{V})^\alpha$, where **R** is the reflection of the light's direction vector about the surface normal, **V** is the direction to the viewer, k_S is the overall specular reflectivity of the surface and α is called the specular power.

Many surfaces aren't uniformly glossy. For example, when a person's face is sweaty and dirty, wet regions appear shiny, while dry or dirty areas appear dull. We can encode high-detail specularity information in a special texture map known as a *specular map*.

If we store the value of k_S in the texels of a specular map, we can control how much specular reflection should be applied at each texel. This kind of specular map is sometimes called a *gloss map*. It is also called a *specular mask*, because zero-valued texels can be used to "mask off" regions of the surface where we do not want specular reflection applied. If we store the value of

图 11.54。DirectX 9 位移贴图。简单的源几何体在运行时进行曲面细分，以生成表面细节。

通过镜面反射贴图中的 α，我们可以控制镜面高光在每个纹素上的“聚焦”程度。这种纹理被称为镜面反射强度贴图。图 11.55 展示了一个光泽贴图的示例。

11.3.1.4 环境映射

环境贴图看起来像是从场景中某个物体的视角拍摄的全景照片，水平方向覆盖 360 度，垂直方向覆盖 180 度或 360 度。环境贴图的作用类似于对物体周围一般光照环境的描述。它通常用于低成本渲染反射。

最常见的两种格式是球形环境贴图和立方体环境贴图。球形贴图看起来像是通过鱼眼镜头拍摄的照片，它被处理为映射到一个半径无限大的球体内部，以被渲染的对象为中心。球形贴图的问题在于它们是使用球面坐标来处理的。在赤道附近，水平和垂直方向都有足够的分辨率。然而，随着垂直（方位角）接近垂直方向，纹理沿水平（天顶）轴的分辨率会降低到单个

Figure 11.54. DirectX 9 displacement mapping. Simple source geometry is tessellated at runtime to produce the surface details.

α in our specular map, we can control the amount of “focus” our specular highlights will have at each texel. This kind of texture is called a *specular power map*. An example of a gloss map is shown in Figure 11.55.

11.3.1.4 Environment Mapping

An environment map looks like a panoramic photograph of the environment taken from the point of view of an object in the scene, covering a full 360 degrees horizontally and either 180 degrees or 360 degrees vertically. An environment map acts like a description of the general lighting environment surrounding an object. It is generally used to inexpensively render reflections.

The two most common formats are *spherical environment maps* and *cubic environment maps*. A spherical map looks like a photograph taken through a fisheye lens, and it is treated as though it were mapped onto the inside of a sphere whose radius is infinite, centered about the object being rendered. The problem with sphere maps is that they are addressed using spherical coordinates. Around the equator, there is plenty of resolution both horizontally and vertically. However, as the vertical (azimuthal) angle approaches vertical, the resolution of the texture along the horizontal (zenith) axis decreases to a single

图 11.55。这张来自 EA 游戏《Fight Night Round 3》的截图展示了如何使用光泽贴图来控制施加到表面每个纹素的镜面反射程度。（参见 Color Plate XX。）

纹理像素。立方体贴图的设计就是为了避免这个问题。

立方体贴图看起来就像一张由六个主要方向（上、下、左、右、前、后）拍摄的照片拼凑而成的合成照片。在渲染过程中，立方体贴图被处理为映射到一个无限远处长方体的六个内表面上，并以被渲染的对象为中心。

为了读取物体表面点 P 对应的环境贴图纹素，我们将从相机发出的射线发射到点 P，并使其绕 P 点的表面法线反射。反射的射线会一直跟随，直到与环境贴图的球体或立方体相交。在对点 P 进行着色时，会用到此交点处的纹素值。

11.3.1.5 三维纹理

现代图形硬件也支持三维纹理。3D纹理可以被认为是一堆2D纹理的堆叠。GPU知道如何根据给定的三维纹理坐标（u，v，w）来寻址和过滤3D纹理。

三维纹理可用于描述物体的外观或体积属性。例如，我们可以渲染一个大理石球体，并允许它被任意平面切割。无论切割的位置如何，纹理在切割处看起来都是连续且正确的，因为纹理在整个球体体积中都是清晰且连续的。

Figure 11.55. This screenshot from EA's *Fight Night Round 3* shows how a gloss map can be used to control the degree of specular reflection that should be applied to each texel of a surface. (See Color Plate XX.)

texel. Cube maps were devised to avoid this problem.

A cube map looks like a composite photograph pieced together from photos taken in the six primary directions (up, down, left, right, front and back). During rendering, a cube map is treated as though it were mapped onto the six inner surfaces of a box at infinity, centered on the object being rendered.

To read the environment map texel corresponding to a point **P** on the surface of an object, we take the ray from the camera to the point **P** and reflect it about the surface normal at **P**. The reflected ray is followed until it intersects the sphere or cube of the environment map. The value of the texel at this intersection point is used when shading the point **P**.

11.3.1.5 Three-Dimensional Textures

Modern graphics hardware also includes support for three-dimensional textures. A 3D texture can be thought of as a stack of 2D textures. The GPU knows how to address and filter a 3D texture, given a three-dimensional texture coordinate (u, v, w).

Three-dimensional textures can be useful for describing the appearance or volumetric properties of an object. For example, we could render a marble sphere and allow it to be cut by an arbitrary plane. The texture would look continuous and correct across the cut no matter where it was made, because the texture is well-defined and continuous throughout the entire volume of the sphere.

11.3.2 高动态范围照明

电视机或 CRT 显示器等显示设备只能产生有限范围的强度。这就是为什么帧缓冲区中的颜色通道被限制在 0 到 1 的范围内。但在现实世界中，光强度可以任意增大。高动态范围 (HDR) 照明试图捕捉这种广泛的光强度范围。

HDR 光照执行光照计算时不会任意限制最终强度。生成的图像以允许强度超过 1 的格式存储。最终效果是，图像中极暗和极亮的区域都能被呈现，且不会丢失任何区域的细节。

在屏幕上显示之前，会使用一个称为色调映射的过程，将图像的强度范围平移和缩放到显示设备支持的范围。这样做可以让渲染引擎重现许多现实世界的视觉效果，例如从黑暗的房间走进明亮的区域时出现的暂时性失明，或者光线似乎从明亮的背光物体后面渗出（这种效果称为光晕）。

表示 HDR 图像的一种方法是使用 32 位浮点数而不是 8 位整数来存储 R、G 和 B 通道。另一种选择是采用完全不同的颜色模型。log-LUV 颜色模型是 HDR 照明的流行选择。在此模型中，颜色表示为强度通道（L）和两个色度通道（U 和 V）。由于人眼对强度变化比色度变化更敏感，因此 L 通道以 16 位存储，而 U 和 V 各仅存储 8 位。此外，L 使用对数刻度（以 2 为底）表示，以便捕捉非常宽的光强度范围。

11.3.3 全局照明

正如我们在 11.1.3.1 节中提到的，全局照明是指一类光照算法，它考虑光线从光源到虚拟相机的路径上与场景中多个物体的相互作用。全局照明考虑了各种效果，例如一个表面遮挡另一个表面时产生的阴影、反射、焦散，以及一个物体的颜色“渗入”到其周围物体的方式。在接下来的章节中，我们将简要介绍一些最常见的全局照明技术。其中一些方法旨在重现单个孤立的效果，例如阴影或反射。而其他一些方法，例如光能传递和光线追踪方法，则旨在提供一个全局光传输的整体模型。

11.3.2 High Dynamic Range Lighting

A display device like a television set or CRT monitor can only produce a limited range of intensities. This is why the color channels in the frame buffer are limited to a zero to one range. But in the real world, light intensities can grow arbitrarily large. *High dynamic range* (HDR) lighting attempts to capture this wide range of light intensities.

HDR lighting performs lighting calculations without clamping the resulting intensities arbitrarily. The resulting image is stored in a format that permits intensities to grow beyond one. The net effect is an image in which extreme dark and light regions can be represented without loss of detail within either type of region.

Prior to display on-screen, a process called *tone mapping* is used to shift and scale the image's intensity range into the range supported by the display device. Doing this permits the rendering engine to reproduce many real-world visual effects, like the temporary blindness that occurs when you walk from a dark room into a brightly lit area, or the way light seems to bleed out from behind a brightly back-lit object (an effect known as *bloom*).

One way to represent an HDR image is to store the R, G and B channels using 32-bit floating-point numbers, instead of 8-bit integers. Another alternative is to employ an entirely different color model altogether. The log-LUV color model is a popular choice for HDR lighting. In this model, color is represented as an intensity channel (L) and two chromaticity channels (U and V). Because the human eye is more sensitive to changes in intensity than it is to changes in chromaticity, the L channel is stored in 16 bits while U and V are given only eight bits each. In addition, L is represented using a logarithmic scale (base two) in order to capture a very wide range of light intensities.

11.3.3 Global Illumination

As we noted in Section 11.1.3.1, global illumination refers to a class of lighting algorithms that account for light's interactions with multiple objects in the scene, on its way from the light source to the virtual camera. Global illumination accounts for effects like the shadows that arise when one surface occludes another, reflections, caustics and the way the color of one object can "bleed" onto the objects around it. In the following sections, we'll take a brief look at some of the most common global illumination techniques. Some of these methods aim to reproduce a single isolated effect, like shadows or reflections. Others like radiosity and ray tracing methods aim to provide a holistic model of global light transport.

11.3.3.1 阴影渲染

当表面阻挡光线路径时，就会产生阴影。理想点光源产生的阴影会很清晰，但在现实世界中，阴影的边缘会很模糊；这被称为半影。半影的产生是因为现实世界的光源会覆盖一定区域，从而产生以不同角度擦过物体边缘的光线。

两种最流行的阴影渲染技术是阴影体和阴影贴图。我们将在以下章节中简要介绍它们。在这两种技术中，场景中的对象通常分为三类：投射阴影的对象、接收阴影的对象以及渲染阴影时完全不考虑的对象。同样，光源也会被标记以指示它们是否应该生成阴影。这项重要的优化限制了在场景中生成阴影所需处理的光源-物体组合的数量。

阴影体积

在阴影体技术中，从产生阴影的光源的有利位置观察每个阴影投射体，并识别阴影投射体的轮廓边缘。这些边缘沿着光源发出的光线方向进行挤压。最终会得到一个新的几何体，用于描述光线被阴影投射体遮挡的空间体积。如图 11.56 所示。

图 11.56.从光源的角度看，通过挤压阴影投射物体的轮廓边缘而生成的阴影体积。

11.3.3.1 Shadow Rendering

Shadows are created when a surface blocks light's path. The shadows caused by an ideal point light source would be sharp, but in the real world shadows have blurry edges; this is called the penumbra. A penumbra arises because real-world light sources cover some area and so produce light rays that graze the edges of an object at different angles.

The two most prevalent shadow rendering techniques are *shadow volumes* and *shadow maps*. We'll briefly describe each in the sections below. In both techniques, objects in the scene are generally divided into three categories: objects that cast shadows, objects that are to receive shadows and objects that are entirely excluded from consideration when rendering shadows. Likewise, the lights are tagged to indicate whether or not they should generate shadows. This important optimization limits the number of light-object combinations that need to be processed in order to produce the shadows in a scene.

Shadow Volumes

In the shadow volume technique, each shadow caster is viewed from the vantage point of a shadow-generating light source, and the shadow caster's silhouette edges are identified. These edges are extruded in the direction of the light rays emanating from the light source. The result is a new piece of geometry that describes the volume of space in which the light is occluded by the shadow caster in question. This is shown in Figure 11.56.

Figure 11.56. A shadow volume generated by extruding the silhouette edges of a shadow casting object as seen from the point of view of the light source.

阴影体利用一种特殊的全屏缓冲区（称为模板缓冲区）来生成阴影。该缓冲区存储与屏幕每个像素对应的单个整数值。渲染过程可以通过模板缓冲区中的值进行屏蔽——例如，我们可以配置 GPU 仅渲染对应模板值非零的片段。此外，还可以配置 GPU，使渲染的几何体以各种有用的方式更新模板缓冲区中的值。

为了渲染阴影，首先绘制场景以在帧缓冲区中生成无阴影图像以及精确的z缓冲区。模板缓冲区被清空，使其每个像素都包含零。然后，从摄像机的视角渲染每个阴影体，正面三角形将模板缓冲区中的值加一，而背面三角形则将其减一。在屏幕上完全没有阴影体的区域，模板缓冲区的像素当然会保留为零。当阴影体的正面和背面都可见时，模板缓冲区也会包含零，因为正面会增加模板值，而背面会再次减少模板值。在阴影体的背面被“真实”场景几何体遮挡的区域，模板值将为一。这告诉我们屏幕上哪些像素处于阴影中。因此，我们可以在第三次渲染阴影，只需使包含非零模板缓冲区值的屏幕区域变暗即可。

阴影贴图

阴影贴图技术实际上是从光源角度（而非相机角度）对每个片段进行深度测试。场景渲染分为两个步骤：首先，从光源角度渲染场景并保存深度缓冲区的内容，从而生成阴影贴图纹理。其次，像往常一样渲染场景，并使用阴影贴图判断每个片段是否处于阴影中。对于场景中的每个片段，阴影贴图会告诉我们光线是否被靠近光源的几何体遮挡，就像z缓冲区会告诉我们片段是否被靠近相机的三角形遮挡一样。

阴影贴图仅包含深度信息——每个纹素记录了它与光源的距离。因此，阴影贴图通常使用硬件的双倍速度 z 轴模式渲染（因为我们只关心深度信息）。对于点光源，渲染阴影贴图时使用透视投影；对于平行光光源，则使用正交投影。

A shadow volume is used to generate a shadow by making use of a special full-screen buffer known as the stencil buffer. This buffer stores a single integer value corresponding to each pixel of the screen. Rendering can be masked by the values in the stencil buffer—for example, we could configure the GPU to only render fragments whose corresponding stencil values are nonzero. In addition, the GPU can be configured so that rendered geometry updates the values in the stencil buffer in various useful ways.

To render shadows, the scene is first drawn to generate an unshadowed image in the frame buffer, along with an accurate z-buffer. The stencil buffer is cleared so that it contains zeros at every pixel. Each shadow volume is then rendered from the point of view of the camera in such a way that front-facing triangles increase the values in the stencil buffer by one, while back-facing triangles decrease them by one. In areas of the screen where the shadow volume does not appear at all, of course the stencil buffer's pixels will be left containing zero. The stencil buffer will also contain zeros where both the front and back faces of the shadow volume are visible, because the front face will increase the stencil value but the back face will decrease it again. In areas where the back face of the shadow volume has been occluded by "real" scene geometry, the stencil value will be one. This tells us which pixels of the screen are in shadow. So we can render shadows in a third pass, by simply darkening those regions of the screen that contain a nonzero stencil buffer value.

Shadow Maps

The shadow mapping technique is effectively a per-fragment depth test performed from the point of view of the light instead of from the point of view of the camera. The scene is rendered in two steps: First, a *shadow map* texture is generated by rendering the scene from the point of view of the light source and saving off the contents of the depth buffer. Second, the scene is rendered as usual, and the shadow map is used to determine whether or not each fragment is in shadow. At each fragment in the scene, the shadow map tells us whether or not the light is being occluded by some geometry that is closer to the light source, in just the same way that the z-buffer tells us whether a fragment is being occluded by a triangle that is closer to the camera.

A shadow map contains only depth information—each texel records how far away it is from the light source. Shadow maps are therefore typically rendered using the hardware's double-speed z-only mode (since all we care about is the depth information). For a point light source, a perspective projection is used when rendering the shadow map; for a directional light source, an orthographic projection is used instead.

图 11.57。最左边的图像是阴影贴图——从特定光源的角度渲染的 z 缓冲区内容。中间图像的像素为黑色，表示光空间深度测试失败（碎片位于阴影中），白色表示测试成功（碎片不在阴影中）。最右边的图像显示了最终渲染的阴影场景。

要使用阴影贴图渲染场景，我们像往常一样从摄像机的视角绘制场景。对于每个三角形的每个顶点，我们计算它在光空间中的位置 - 即在最初生成阴影贴图时使用的相同“视图空间”中。这些光空间坐标可以像任何其他顶点属性一样在三角形上进行插值。这给了我们每个片段在光空间中的位置。要确定给定片段是否在阴影中，我们将片段的光空间（x，y）坐标转换为阴影贴图中的纹理坐标（u，v）。然后，我们将片段的光空间z坐标与阴影深度图中相应纹素处存储的深度进行比较。如果片段的光空间z比阴影贴图中的纹素距离光源更远，那么它一定是被某个更靠近光源的其他几何体遮挡了 - 因此它位于阴影中。同样，如果片段的光空间 z 轴比阴影贴图中的纹素更靠近光源，则该片段不会被遮挡，也不会处于阴影中。基于此信息，可以相应地调整片段的颜色。阴影映射过程如图 11.57 所示。

11.3.3.2 环境光遮蔽

环境光遮蔽是一种模拟接触阴影（当场景仅受环境光照射时产生的柔和阴影）的技术。实际上，环境光遮蔽描述的是表面上每个点对光线的“可及性”。例如，一段管道的内部比其外部更难受到环境光的照射。如果在阴天将管道放置在室外，其内部通常会比外部显得更暗。

图 11.58 展示了环境光遮蔽如何在汽车底部、轮毂以及车身面板接缝处产生阴影。环境光遮蔽是通过在表面某一点构建

Figure 11.57. The far left image is a shadow map—the contents of the z-buffer as rendered from the point of view of a particular light source. The pixels of the center image are black where the light-space depth test failed (fragment in shadow) and white where it succeeded (fragment not in shadow). The far right image shows the final scene rendered with shadows.

To render a scene using a shadow map, we draw the scene as usual from the point of view of the camera. For each vertex of every triangle, we calculate its position in *light space*—i.e., in the same “view space” that was used when generating the shadow map in the first place. These light-space coordinates can be interpolated across the triangle, just like any other vertex attribute. This gives us the position of each fragment *in light space*. To determine whether a given fragment is in shadow or not, we convert the fragment’s light-space (x, y)-coordinates into texture coordinates (u, v) within the shadow map. We then compare the fragment’s light-space z-coordinate with the depth stored at the corresponding texel in the shadow depth map. If the fragment’s light-space z is farther away from the light than the texel in the shadow map, then it must be occluded by some other piece of geometry that is closer to the light source—hence it is in shadow. Likewise, if the fragment’s light-space z is closer to the light source than the texel in the shadow map, then it is not occluded and is not in shadow. Based on this information, the fragment’s color can be adjusted accordingly. The shadow mapping process is illustrated in Figure 11.57.

11.3.3.2 Ambient Occlusion

Ambient occlusion is a technique for modeling *contact shadows*—the soft shadows that arise when a scene is illuminated by only ambient light. In effect, ambient occlusion describes how “accessible” each point on a surface is to light in general. For example, the interior of a section of pipe is less accessible to ambient light than its exterior. If the pipe were placed outside on an overcast day, its interior would generally appear darker than its exterior.

Figure 11.58 shows how ambient occlusion produces shadows underneath a car and in its wheel wells, as well as within the seams between body panels. Ambient occlusion is measured at a point on a surface by constructing

图 11.58. 使用环境光遮蔽渲染的汽车。注意车辆底部和轮舱内的暗区。

一个以该点为中心的半径很大的半球，并确定该半球面积中有多少百分比从该点可见。对于静态物体，它可以离线预先计算，因为环境光遮蔽与视线方向和入射光方向无关。它通常存储在纹理贴图中，该纹理贴图记录了表面每个纹素的环境光遮蔽程度。

11.3.3.3 反思

当光线在高镜面（光泽）表面上反射时，会在表面形成场景另一部分的图像，从而产生反射。反射可以通过多种方式实现。环境贴图用于在光泽物体表面上产生周围环境的一般反射。在平面（例如镜子）上，可以通过将相机的位置反射到反射面的平面上，然后将该反射视点的场景渲染成纹理来产生直接反射。之后，纹理会在第二次渲染中应用到反射面上（参见图 11.59）。

11.3.3.4 焦散

焦散是由水面或抛光金属等非常光亮的表面在强烈的反射或折射下产生的明亮镜面高光。当反射表面移动时，例如水面，焦散效果会在反射表面闪烁并“游动”。焦散效果可以通过将包含半随机明亮高光的纹理（可以是动画的）投射到受影响的表面上来产生。图 11.60 展示了此技术的一个示例。

Figure 11.58. A car rendered with ambient occlusion. Notice the darkened areas underneath the vehicle and in the wheel wells.

a hemisphere with a very large radius centered on that point and determing what percentage of that hemisphere's area is visible from the point in question. It can be precomputed offline for static objects, because ambient occlusion is independent of view direction and the direction of incident light. It is typically stored in a texture map that records the level of ambient occlusion at each texel across the surface.

11.3.3.3 Reflections

Reflections occur when light bounces off a highly specular (shiny) surface producing an image of another portion of the scene in the surface. Reflections can be implemented in a number of ways. Environment maps are used to produce general reflections of the surrounding environment on the surfaces of shiny objects. Direct reflections in flat surfaces like mirrors can be produced by reflecting the camera's position about the plane of the reflective surface and then rendering the scene from that reflected point of view into a texture. The texture is then applied to the reflective surface in a second pass (see Figure 11.59).

11.3.3.4 Caustics

Caustics are the bright specular highlights arising from intense reflections or refractions from very shiny surfaces like water or polished metal. When the reflective surface moves, as is the case for water, the caustic effects glimmer and "swim" across the surfaces on which they fall. Caustic effects can be produced by projecting a (possibly animated) texture containing semi-random bright highlights onto the affected surfaces. An example of this technique is shown in Figure 11.60.

图 11.59。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的镜面反射效果，是通过将场景渲染到纹理上，然后应用到镜面表面来实现的。（参见彩色图 XXI。）

图 11.60.通过将动画纹理投射到受影响的表面来产生水焦散。

11.3.3.5 次表面散射

当光线从表面的某一点进入，在表面下方散射，然后再次出现在表面的另一个点时，我们称之为次表面散射。这种现象就是人体皮肤、蜡和大理石雕像呈现“暖光”的原因（例如，图 11.61）。次表面散射可以用 BRDF 的一个更高级的变体来描述（参见 11.1.3.2 节），即双向表面散射反射分布函数 (BSSRDF)。

次表面散射可以用多种方式模拟。基于深度图的次表面散射会渲染阴影图（参见第 11.3.3.1 节），但它不是用来确定哪些像素处于阴影中，而是用来测量光束需要传播多远才能穿过所有

Figure 11.59. Mirror reflections in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) implemented by rendering the scene to a texture that is subsequently applied to the mirror's surface. (See Color Plate XXI.)

Figure 11.60. Water caustics produced by projecting an animated texture onto the affected surfaces.

11.3.3.5 Subsurface Scattering

When light enters a surface at one point, is scattered beneath the surface, and then reemerges at a different point on the surface, we call this *subsurface scattering*. This phenomenon is responsible for the "warm glow" of human skin, wax and marble statues (e.g., Figure 11.61). Subsurface scattering is described by a more-advanced variant of the BRDF (see Section 11.1.3.2) known as the BSSRDF (*bidirectional surface scattering reflectance distribution function*).

Subsurface scattering can be simulated in a number of ways. Depth-map–based subsurface scattering renders a shadow map (see Section 11.3.3.1), but instead of using it to determine which pixels are in shadow, it is used to measure how far a beam of light would have to travel in order to pass all the

图 11.61。左侧为未使用次表面散射（即使用 BRDF 光照模型）渲染的龙。右侧为使用次表面散射（即使用 BSSRDF 模型）渲染的同一条龙。图片由弗吉尼亚大学的 Rui Wang 绘制。

穿过遮挡物体。然后，为物体的阴影侧添加一个人工漫反射照明项，其强度与光线到达物体另一侧所需的传播距离成反比。这使得物体在光源对面的一侧看起来略微发光，但仅限于物体相对较薄的部分。有关次表面散射技术的更多信息，请参阅http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html。

11.3.3.6 预计算辐射传输（PRT）

预计算辐射传递 (PRT) 是一种流行的技术，旨在实时模拟基于辐射度的渲染方法的效果。它通过预先计算并存储入射光线从各个可能方向接近时与表面相互作用（反射、折射、散射等）的完整描述来实现这一点。在运行时，可以查找对特定入射光线的响应，并将其快速转换为非常精确的照明结果。

一般来说，光在表面某一点的响应是一个复函数，定义在以该点为中心的半球上。为了使PRT技术切实可行，需要对该函数进行简洁的表示。一种常见的方法是将该函数近似为球谐基函数的线性组合。这本质上相当于将简单的标量函数f(x)编码为移位和缩放正弦波的线性组合的三维等效函数。

PRT 的细节远远超出了我们的讨论范围。更多信息，请参阅 http://web4.cs.ucl.ac.uk/staff/j.kautz/publications/prtSIG02.pdf。DirectX SDK 中提供了一个 DirectX 示例程序，演示了 PRT 照明技术——请参阅 http://msdn.microsoft.com/en-us/library/bb147287。

有关更多详细信息，请参阅.aspx。

Figure 11.61. On the left, a dragon rendered without subsurface scattering (i.e., using a BRDF lighting model). On the right, the same dragon rendered with subsurface scattering (i.e., using a BSSRDF model). Images rendered by Rui Wang at the University of Virginia.

way through the occluding object. The shadowed side of the object is then given an artificial diffuse lighting term whose intensity is inversely proportional to the distance the light had to travel in order to emerge on the opposite side of the object. This causes objects to appear to be glowing slightly on the side opposite to the light source but only where the object is relatively thin. For more information on subsurface scattering techniques, see http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html.

11.3.3.6 Precomputed Radiance Transfer (PRT)

Precomputed radiance transfer (PRT) is a popular technique that attempts to simulate the effects of radiosity-based rendering methods in real time. It does so by precomputing and storing a complete description of how an incident light ray would interact with a surface (reflect, refract, scatter, etc.) when approaching from every possible direction. At runtime, the response to a particular incident light ray can be looked up and quickly converted into very accurate lighting results.

In general the light's response at a point on the surface is a complex function defined on a hemisphere centered about the point. A compact representation of this function is required to make the PRT technique practical. A common approach is to approximate the function as a linear combination of spherical harmonic basis functions. This is essentially the three-dimensional equivalent of encoding a simple scalar function $f(x)$ as a linear combination of shifted and scaled sine waves.

The details of PRT are far beyond our scope. For more information, see http://web4.cs.ucl.ac.uk/staff/j.kautz/publications/prtSIG02.pdf. PRT lighting techniques are demonstrated in a DirectX sample program available in the DirectX SDK—see http://msdn.microsoft.com/en-us/library/bb147287.aspx for more details.

11.3.4 延迟渲染

在传统的基于三角形光栅化的渲染中，所有光照和着色计算都在世界空间、视图空间或切线空间中的三角形片段上执行。这种技术的问题在于它本质上效率低下。首先，我们可能会做一些不必要的工作。我们对三角形的顶点进行着色，却在光栅化阶段发现整个三角形都被 z 测试进行了深度剔除。早期的 z 测试有助于消除不必要的像素着色器评估，但即使这样也并非完美。此外，为了处理具有大量光源的复杂场景，我们最终会得到大量不同版本的顶点和像素着色器——这些版本处理不同数量的光源、不同类型的光源、不同数量的蒙皮权重等等。

延迟渲染是另一种场景着色方法，可以解决许多此类问题。在延迟渲染中，大多数光照计算都在屏幕空间而非视图空间完成。我们可以高效地渲染场景，无需担心光照问题。在此阶段，我们将所有照亮像素所需的信息存储在一个称为 G 缓冲区的“深”帧缓冲区中。场景完全渲染完成后，我们会使用 G 缓冲区中的信息执行光照和着色计算。这通常比视图空间光照效率高得多，避免了着色器变体的激增，并可以相对轻松地渲染一些非常令人满意的效果。

G缓冲区在物理上可以实现为一组缓冲区，但从概念上讲，它是一个单帧缓冲区，其中包含关于场景中物体在屏幕上每个像素的光照和表面属性的丰富信息。典型的G缓冲区可能包含以下每像素属性：深度、视图空间或世界空间中的表面法线、漫反射颜色、镜面反射强度，甚至预计算辐射传递 (PRT) 系数。以下来自Guerrilla Games的《杀戮地带2》（图11.62）的屏幕截图展示了G缓冲区的一些典型组件。

关于延迟渲染的深入讨论超出了我们的范围，但 Guerrilla Games 的人们已经准备了一个关于这个主题的精彩演示文稿，可以在 http://www.slideshare.net/guerrillagames/deferred 上找到。

-在杀戮地带-2-9691589 中渲染。

11.3.5 基于物理的着色

传统游戏照明引擎要求艺术家和灯光师在众多不同的渲染引擎系统中调整各种有时非直观的参数，以便在游戏中实现所需的“外观”。

11.3.4 Deferred Rendering

In traditional triangle-rasterization–based rendering, all lighting and shading calculations are performed on the triangle fragments in world space, view space or tangent space. The problem with this technique is that it is inherently inefficient. For one thing, we potentially do work that we don't need to do. We shade the vertices of triangles, only to discover during the rasterization stage that the entire triangle is being depth-culled by the z-test. Early z-tests help eliminate unnecessary pixel shader evaluations, but even this isn't perfect. What's more, in order to handle a complex scene with lots of lights, we end up with a proliferation of different versions of our vertex and pixel shaders—versions that handle different numbers of lights, different types of lights, different numbers of skinning weights, etc.

Deferred rendering is an alternative way to shade a scene that addresses many of these problems. In deferred rendering, the majority of the lighting calculations are done in screen space, not view space. We efficiently render the scene without worrying about lighting. During this phase, we store all the information we're going to need to light the pixels in a "deep" frame buffer known as the *G-buffer*. Once the scene has been fully rendered, we use the information in the G-buffer to perform our lighting and shading calculations. This is usually much more efficient than view-space lighting, avoids the proliferation of shader variants and permits some very pleasing effects to be rendered relatively easily.

The G-buffer may be physically implemented as a collection of buffers, but conceptually it is a single frame buffer containing a rich set of information about the lighting and surface properties of the objects in the scene at every pixel on the screen. A typical G-buffer might contain the following per-pixel attributes: depth, surface normal in view space or world space, diffuse color, specular power, even precomputed radiance transfer (PRT) coefficients. The following sequence of screenshots from Guerrilla Games' *Killzone 2* (Figure 11.62) shows some of the typical components of the G-buffer.

An in-depth discussion of deferred rendering is beyond our scope, but the folks at Guerrilla Games have prepared an excellent presentation on the topic, which is available at http://www.slideshare.net/guerrillagames/deferred-rendering-in-killzone-2-9691589.

11.3.5 Physically Based Shading

Traditional game lighting engines have required artists and lighters to tweak a wide variety of sometimes non-intuitive parameters, across numerous disparate rendering engine systems, in order to achieve a desired "look" in-game.

图 11.62。Guerrilla Games 出品的《杀戮地带 2》的截图，展示了延迟渲染中使用的 G 缓冲区的一些典型组件。上图显示了最终渲染的图像。下方从左上角开始顺时针依次为反照率（漫反射）颜色、深度、视图空间法线、屏幕空间二维运动矢量（用于运动模糊）、镜面反射功率和镜面反射强度。（参见彩色图 XXII。）

这可能是一个艰巨且耗时的过程。更糟糕的是，在某种光照条件下有效的参数设置在另一种光照场景下可能并不起作用。为了解决这些问题，渲染程序员正在转向基于物理的着色模型。

基于物理的着色模型试图模拟现实世界中光线传播和与材质交互的方式，使美术师和灯光师能够使用直观的、以真实世界单位测量的真实物理量来调整着色器参数。本书不包含基于物理的着色的完整讨论，但您可以点击此处了解更多信息：https://www.marmoset.co/toolbag/learn/pbr-theory。

11.4 视觉效果和叠加

到目前为止，我们讨论的渲染管线主要负责渲染三维实体对象。一些专门的渲染系统通常位于该管线之上，负责渲染各种视觉元素，例如粒子效果、贴花（用于表示弹孔、裂缝、划痕和其他表面细节的小型几何体叠加层）、毛发、雨水或飘落的雪花、水以及其他专门的视觉效果。此外，还可以应用全屏后期特效，包括晕影（降低屏幕边缘的亮度和饱和度）、运动模糊、深度

Figure 11.62. Screenshots from *Killzone 2* by Guerrilla Games, showing some of the typical components of the G-buffer used in deferred rendering. The upper image shows the final rendered image. Below it, clockwise from the upper left, are the albedo (diffuse) color, depth, view-space normal, screen-space 2D motion vector (for motion blurring), specular power and specular intensity. (See Color Plate XXII.)

This can be an arduous and time-consuming process. What's worse, parameter settings that work well under one set of lighting conditions might not work well under other lighting scenarios. To address these problems, rendering programmers are turning toward *physically based shading* models.

A physically based shading model attempts to approximate the ways in which light travels and interacts with materials in the real world, allowing artists and lighters to tweak shader parameters using intuitive, real-world quantities measured in real-world units. A complete discussion of physically based shading is beyond the scope of this book, but you can start to learn more about it here: https://www.marmoset.co/toolbag/learn/pbr-theory.

11.4 Visual Effects and Overlays

The rendering pipeline we've discussed to this point is responsible primarily for rendering three-dimensional solid objects. A number of specialized rendering systems are typically layered on top of this pipeline, responsible for rendering visual elements like particle effects, decals (small geometry overlays that represent bullet holes, cracks, scratches and other surface details), hair and fur, rain or falling snow, water and other specialized visual effects. Full-screen post effects may be applied, including vignette (a reduction of brightness and saturation around the edges of the screen), motion blur, depth

场景模糊、人工/增强色彩等等。最后，游戏的菜单系统和平视显示器 (HUD) 通常通过在屏幕空间中渲染文本和其他二维或三维图形并将其叠加在三维场景之上来实现。

深入介绍这些引擎系统超出了我们的范围。在接下来的章节中，我们将简要概述这些渲染系统，并为您提供更多信息。

11.4.1 粒子效果

粒子渲染系统负责渲染无定形物体，例如烟雾、火花、火焰等等。这些被称为粒子效果。粒子效果与其他可渲染几何体的主要区别如下：

- 它由大量相对简单的几何图形组成 - 通常是称为四边形的简单卡片，每个卡片由两个三角形组成。
- 几何体通常面向相机（即广告牌），这意味着引擎必须采取措施确保每个四边形的面法线始终直接指向相机的焦点。
- 其材质几乎总是半透明的。因此，粒子效果具有一些严格的渲染顺序约束，这些约束不适用于场景中的大多数不透明物体。
- 粒子动画的呈现方式丰富多样。它们的位置、方向、大小（比例）、纹理坐标以及许多着色器参数都会逐帧变化。这些变化可以通过手动编写的动画曲线或程序化方法定义。
- 粒子通常会持续生成和销毁。粒子发射器是世界中一个逻辑实体，它以用户指定的速率生成粒子；当粒子撞击到预先定义的死亡平面，或者存活了用户定义的时间长度，或者根据用户指定的其他条件，粒子就会被销毁。

粒子效果可以使用规则三角形网格几何体搭配合适的着色器来渲染。然而，由于上述特性，在实际生产的游戏引擎中，通常需要使用专门的粒子效果动画和渲染系统来实现它们。图 11.63 展示了一些粒子效果的示例。

of field blurring, artificial/enhanced colorization, and the list goes on. Finally, the game's menu system and heads-up display (HUD) are typically realized by rendering text and other two- or three-dimensional graphics in screen space overlaid on top of the three-dimensional scene.

An in-depth coverage of these engine systems is beyond our scope. In the following sections, we'll provide a brief overview of these rendering systems, and point you in the direction of additional information.

11.4.1 Particle Effects

A particle rendering system is concerned with rendering amorphous objects like clouds of smoke, sparks, flame and so on. These are called *particle effects*. The key features that differentiate a particle effect from other kinds of renderable geometry are as follows:

- It is composed of a very *large number* of *relatively simple* pieces of geometry—most often simple cards called *quads*, composed of two triangles each.
- The geometry is often *camera-facing* (i.e., billboarded), meaning that the engine must take steps to ensure that the face normals of each quad always point directly at the camera's focal point.
- Its materials are almost always *semitransparent* or *translucent*. As such, particle effects have some stringent *rendering order* constraints that do not apply to the majority of opaque objects in a scene.
- Particles *animate* in a rich variety of ways. Their positions, orientations, sizes (scales), texture coordinates and many of their shader parameters vary from frame to frame. These changes are defined either by hand-authored animation curves or via procedural methods.
- Particles are typically *spawned and killed* continually. A particle emitter is a logical entity in the world that creates particles at some user-specified rate; particles are killed when they hit a predefined death plane, or when they have lived for a user-defined length of time, or as decided by some other user-specified criteria.

Particle effects could be rendered using regular triangle mesh geometry with appropriate shaders. However, because of the unique characteristics listed above, a specialized particle effect animation and rendering system is always used to implement them in a real production game engine. A few example particle effects are shown in Figure 11.63.

图 11.63。《神秘海域 3：德雷克的诡计》中的火焰、烟雾和子弹追踪粒子效果（© 2011/TM SIE。由顽皮狗创作和开发，PlayStation 3）。（参见彩色图 XXIII。）

粒子系统的设计和实现是一个内容丰富的主题，单独来看可以占用好几章。更多关于粒子系统的信息，请参阅[2，第 10.7 节]、[16，第 20.5 节]、[11，第 13.7 节]和[12，第 4.1.2 节]。

11.4.2 贴花

贴花是相对较小的几何体，它覆盖在场景中的常规几何体之上，允许动态修改表面的视觉外观。例如弹孔、脚印、划痕、裂缝等。

现代引擎最常用的方法是将贴花建模为一个矩形区域，该矩形区域将沿着射线投影到场景中。这会在三维空间中生成一个长方体。该长方体首先与贴花的表面相交。相交几何体的三角形会被提取出来，并根据贴花投影的长方体的四个边界平面进行裁剪。通过为每个顶点生成合适的纹理坐标，将生成的三角形与所需的贴花纹理进行纹理映射。然后，这些纹理映射后的三角形会被渲染到常规场景的上方，通常会使用视差贴图来赋予它们深度感，并带有轻微的 z 轴偏移（通常通过稍微移动近平面来实现），这样它们就不会与叠加在其上的几何体发生 z 轴冲突。最终呈现出弹孔、划痕或其他类型的表面修饰效果。

图 11.64 描绘了一些弹孔贴花。

有关创建和渲染贴花的更多信息，请参阅[9，第 4.8 节]和[32，第 9.2 节]。

Figure 11.63. Flame, smoke and bullet tracer particle effects in *Uncharted 3: Drake's Deception* (© 2011/™ SIE. Created and developed by Naughty Dog, PlayStation 3). (See Color Plate XXIII.)

Particle system design and implementation is a rich topic that could occupy many chapters all on its own. For more information on particle systems, see [2, Section 10.7], [16, Section 20.5], [11, Section 13.7] and [12, Section 4.1.2].

11.4.2 Decals

A *decal* is a relatively small piece of geometry that is overlaid on top of the regular geometry in the scene, allowing the visual appearance of the surface to be modified dynamically. Examples include bullet holes, foot prints, scratches, cracks, etc.

The approach most often used by modern engines is to model a decal as a rectangular area that is to be projected along a ray into the scene. This gives rise to a rectangular prism in 3D space. Whatever surface the prism intersects first becomes the surface of the decal. The triangles of the intersected geometry are extracted and clipped against the four bounding planes of the decal's projected prism. The resulting triangles are texture-mapped with a desired decal texture by generating appropriate texture coordinates for each vertex. These texture-mapped triangles are then rendered over the top of the regular scene, often using parallax mapping to give them the illusion of depth and with a slight z-bias (usually implemented by shifting the near plane slightly) so they don't experience z-fighting with the geometry on which they are overlaid. The result is the appearance of a bullet hole, scratch or other kind of surface modification. Some bullet-hole decals are depicted in Figure 11.64.

For more information on creating and rendering decals, see [9, Section 4.8] and [32, Section 9.2].

图 11.64。《神秘海域 3：德雷克的诡计》中的视差贴花（© 2011/TM SIE。由顽皮狗创作和开发，PlayStation 3）。（参见彩色图 XXIV。）

11.4.3 环境影响

任何在较为自然或逼真的环境中进行的游戏都需要某种环境渲染效果。这些效果通常由专门的渲染系统实现。我们将在以下章节中简要介绍一些较为常见的渲染系统。

11.4.3.1 天空

游戏世界中的天空需要包含生动的细节，但从技术上讲，它距离摄像机非常远。因此，我们无法按照实际情况进行建模，只能求助于各种专门的渲染技术。

一种简单的方法是在渲染任何 3D 几何体之前，先用天空纹理填充帧缓冲区。天空纹理的纹理像素比应约为 1:1，以便纹理的分辨率大致或精确地与屏幕分辨率一致。天空纹理可以旋转和滚动，以与游戏中摄像机的运动保持一致。在渲染天空时，我们确保将帧缓冲区中所有像素的深度设置为最大深度值。这确保了 3D 场景元素始终位于天空之上。街机热门游戏《Hydro Thunder》正是以这种方式渲染了天空。

在现代游戏平台上，像素着色的开销可能很高，天空渲染通常在场景其余部分渲染完成后进行。首先将 z 缓冲区清空至最大 z 值。然后渲染场景。

Figure 11.64. Parallax-mapped decals from *Uncharted 3: Drake's Deception* (© 2011/™ SIE. Created and developed by Naughty Dog, PlayStation 3). (See Color Plate XXIV.)

11.4.3 Environmental Effects

Any game that takes place in a somewhat natural or realistic environment requires some kind of environmental rendering effects. These effects are usually implemented via specialized rendering systems. We'll take a brief look at a few of the more common of these systems in the following sections.

11.4.3.1 Skies

The sky in a game world needs to contain vivid detail, yet technically speaking it lies an extremely long distance away from the camera. Therefore, we cannot model it as it really is and must turn instead to various specialized rendering techniques.

One simple approach is to fill the frame buffer with the sky texture prior to rendering any 3D geometry. The sky texture should be rendered at an approximate 1:1 texel-to-pixel ratio, so that the texture is roughly or exactly the resolution of the screen. The sky texture can be rotated and scrolled to correspond to the motions of the camera in-game. During rendering of the sky, we make sure to set the depth of all pixels in the frame buffer to the maximum possible depth value. This ensures that the 3D scene elements will always sort on top of the sky. The arcade hit *Hydro Thunder* rendered its skies in exactly this manner.

On modern game platforms, where pixel shading costs can be high, sky rendering is often done *after* the rest of the scene has been rendered. First the z-buffer is cleared to the maximum z-value. Then the scene is rendered.

最后渲染天空，启用 z 测试，关闭 z 写入，并使用比最大值小一的 αz 测试值。这使得天空仅在未被地形、建筑物和树木等近距离物体遮挡的地方绘制。最后绘制天空可确保其像素着色器以尽可能少的屏幕像素运行。

对于玩家可以朝任何方向观看的游戏，我们可以使用天空穹顶或天空盒。穹顶或天空盒的渲染中心始终位于摄像机的当前位置，因此无论摄像机在游戏世界中移动到何处，它看起来都位于无穷远处。与天空纹理方法一样，天空盒或穹顶在任何其他 3D 几何体之前渲染，并且渲染天空时，帧缓冲区中的所有像素都设置为最大 z 值。这意味着穹顶或天空盒相对于场景中的其他物体实际上可以很小。它的大小无关紧要，只要它在绘制时填满整个帧缓冲区即可。有关天空渲染的更多信息，请参阅 [2，第 10.3 节] 和 [44，第 253 页]。

云通常也由专门的渲染和动画系统实现。在早期的游戏，例如《毁灭战士》和《雷神之锤》中，云只是带有滚动半透明云纹理的平面。最近的云技术包括面向摄像机的卡片（公告牌）、基于粒子效果的云和体积云效果。

11.4.3.2 地形

地形系统的目标是模拟地球表面，并提供一个画布，用于布局其他静态和动态元素。地形有时会在 Maya 等软件中明确建模。但如果玩家能够看到很远的地方，我们通常需要某种动态曲面细分或其他细节级别 (LOD) 系统。我们可能还需要限制表示非常大的户外区域所需的数据量。

高度场地形是建模大片地形区域的热门选择之一。
由于高度场通常存储在灰度纹理图中，因此数据大小可以保持相对较小。在大多数基于高度场的地形系统中，水平面（y = 0）以规则的网格图案进行细分，地形顶点的高度通过采样高度场纹理来确定。单位面积的三角形数量可以根据与摄像机的距离而变化，从而允许在远处看到大规模特征，同时仍允许为附近地形呈现大量细节。图 11.65 显示了通过高度场位图定义的地形示例。

地形系统通常提供专门的工具来“绘制”高度场本身，雕刻出道路、河流等地形特征。纹理

Finally the sky is rendered, with z-testing enabled, z writing turned off, and using a z-test value that is one less than the maximum. This causes the sky to be drawn only where it is not occluded by closer objects like terrain, buildings and trees. Drawing the sky last ensures that its pixel shader is run for the minimum possible number of screen pixels.

For games in which the player can look in any direction, we can use a *sky dome* or *sky box*. The dome or box is rendered with its center always at the camera's current location, so that it appears to lie at infinity, no matter where the camera moves in the game world. As with the sky texture approach, the sky box or dome is rendered before any other 3D geometry, and all of the pixels in the frame buffer are set to the maximum z-value when the sky is rendered. This means that the dome or box can actually be tiny, relative to other objects in the scene. Its size is irrelevant, as long as it fills the entire frame buffer when it is drawn. For more information on sky rendering, see [2, Section 10.3] and [44, page 253].

Clouds are often implemented with a specialized rendering and animation system as well. In early games like *Doom* and *Quake*, the clouds were just planes with scrolling semitransparent cloud textures on them. More-recent cloud techniques include camera-facing cards (billboards), particle-effect based clouds and volumetric cloud effects.

11.4.3.2 Terrain

The goal of a terrain system is to model the surface of the earth and provide a canvas of sorts upon which other static and dynamic elements can be laid out. Terrain is sometimes modeled explicitly in a package like Maya. But if the player can see far into the distance, we usually want some kind of dynamic tessellation or other level of detail (LOD) system. We may also need to limit the amount of data required to represent very large outdoor areas.

Height field terrain is one popular choice for modeling large terrain areas. The data size can be kept relatively small because a height field is typically stored in a grayscale texture map. In most height-field– based terrain systems, the horizontal ($y = 0$) plane is tessellated in a regular grid pattern, and the heights of the terrain vertices are determined by sampling the height field texture. The number of triangles per unit area can be varied based on distance from the camera, thereby allowing large-scale features to be seen in the distance, while still permitting a good deal of detail to be represented for nearby terrain. An example of a terrain defined via a height field bitmap is shown in Figure 11.65.

Terrain systems usually provide specialized tools for “painting” the height field itself, carving out terrain features like roads, rivers and so on. Texture

图 11.65。灰度高度场位图（左）可用于控制地形网格（右）中顶点的垂直位置。在此示例中，水面与地形网格相交，形成了岛屿。

地形系统中的贴图通常是四种或更多种纹理的混合。这使得艺术家只需暴露其中一个纹理层，即可在草地、泥土、砾石和其他地形特征上进行“绘制”。这些纹理层可以相互交叉混合，以提供平滑的纹理过渡。一些地形工具还允许剪切地形的各个部分，以便以规则网格几何体的形式插入建筑物、战壕和其他特殊地形特征。地形创作工具有时直接集成到游戏世界编辑器中，而在其他引擎中，它们可能是独立的工具。

当然，高度场地形只是游戏中模拟地球表面的众多选项之一。有关地形渲染的更多信息，请参阅[8，第 4.16 至 4.19 节]和[9，第 4.2 节]。

11.4.3.3 水

如今，水体渲染器在游戏中已是司空见惯。水的种类繁多，包括海洋、水池、河流、瀑布、喷泉、喷水口、水坑和潮湿的固体表面。每种类型的水通常都需要一些专门的渲染技术。有些还需要动态运动模拟。大型水体可能需要动态曲面细分或其他类似于地形系统中所采用的 LOD 方法。

水系统有时会与游戏的刚体动力学系统（例如漂浮、水流冲击等）以及游戏玩法（例如光滑表面、游泳机制、潜水机制、在垂直水流中穿梭等）相互作用。水效果通常是通过组合不同的渲染技术和子系统来实现的。例如，瀑布可能会利用

Figure 11.65. A grayscale height field bitmap (left) can be used to control the vertical positions of the vertices in a terrain grid mesh (right). In this example, a water plane intersects the terrain mesh to create islands.

mapping in a terrain system is often a blend between four or more textures. This allows artists to “paint” in grass, dirt, gravel and other terrain features by simply exposing one of the texture layers. The layers can be cross-blended from one to another to provide smooth textural transitions. Some terrain tools also permit sections of the terrain to be cut out to permit buildings, trenches and other specialized terrain features to be inserted in the form of regular mesh geometry. Terrain authoring tools are sometimes integrated directly into the game world editor, while in other engines they may be stand-alone tools.

Of course, height field terrain is just one of many options for modeling the surface of the Earth in a game. For more information on terrain rendering, see [8, Sections 4.16 through 4.19] and [9, Section 4.2].

11.4.3.3 Water

Water renderers are commonplace in games nowadays. There are lots of different kinds of water, including oceans, pools, rivers, waterfalls, fountains, jets, puddles and damp solid surfaces. Each type of water generally requires some specialized rendering technology. Some also require dynamic motion simulations. Large bodies of water may require dynamic tessellation or other LOD methodologies similar to those employed in a terrain system.

Water systems sometimes interact with a game’s rigid body dynamics system (flotation, force from water jets, etc.) and with gameplay (slippery surfaces, swimming mechanics, diving mechanics, riding vertical jets of water and so on). Water effects are often created by combining disparate rendering technologies and subsystems. For example, a waterfall might make use

专业的水体着色器、滚动纹理、底部雾气的粒子效果、类似贴花的泡沫覆盖层等等，不胜枚举。如今的游戏提供了一些相当惊艳的水体效果，而对实时流体动力学等技术的积极研究有望在未来几年使水体模拟更加丰富逼真。有关水体渲染和模拟技术的更多信息，请参阅[2，第 9.3、9.5 和 9.6 节]、[15] 和 [8，第 2 .6 和 5.11 节]。

11.4.4 覆盖

大多数游戏都配备了抬头显示器、游戏内图形用户界面和菜单系统。这些叠加层通常由直接在视图空间或屏幕空间中渲染的二维和三维图形组成。

叠加层通常在主场景之后渲染，并禁用 z 测试以确保它们显示在三维场景之上。二维叠加层通常通过使用正交投影在屏幕空间中渲染四边形（三角形对）来实现。三维叠加层可以使用正交投影或常规透视投影进行渲染，其中几何体位于视图空间中，使其跟随摄像机移动。

11.4.4.1 文本和字体

游戏引擎的文本/字体系统通常实现为一种特殊的二维（有时是三维）叠加层。其核心在于，文本渲染系统需要能够显示与文本字符串对应的字符字形序列，并以各种方向排列在屏幕上。

字体通常通过称为字形图集的纹理图实现，其中包含各种所需的字形。此纹理通常由单个 alpha 通道组成 - 每个像素的值表示被字形内部覆盖的像素的百分比。字体描述文件提供信息，例如纹理内每个字形的边界框，以及字体布局信息，例如字距调整、基线偏移等。通过绘制四边形来渲染字形，其（u，v）坐标对应于图集纹理图中所需字形的边界框。纹理图提供 alpha 值，而颜色是单独指定的，允许从同一张图集渲染任何颜色的字形。

字体渲染的另一个选择是使用像 FreeType (https://www.freetype.org/) 这样的字体库。FreeType 库使游戏或其他应用程序能够读取各种格式的字体，包括

of specialized water shaders, scrolling textures, particle effects for mist at the base, a decal-like overlay for foam, and the list goes on. Today's games offer some pretty amazing water effects, and active research into technologies like real-time fluid dynamics promises to make water simulations even richer and more realistic in the years ahead. For more information on water rendering and simulation techniques, see [2, Sections 9.3, 9.5 and 9.6], [15] and [8, Sections 2.6 and 5.11].

11.4.4 Overlays

Most games have heads-up displays, in-game graphical user interfaces and menu systems. These *overlays* are typically comprised of two- and three-dimensional graphics rendered directly in view space or screen space.

Overlays are generally rendered after the primary scene, with z-testing disabled to ensure that they appear on top of the three-dimensional scene. Two-dimensional overlays are typically implemented by rendering quads (triangle pairs) in screen space using an orthographic projection. Three-dimensional overlays may be rendered using an orthographic projection or via the regular perspective projection with the geometry positioned in view space so that it follows the camera around.

11.4.4.1 Text and Fonts

A game engine's text/font system is typically implemented as a special kind of two-dimensional (or sometimes three-dimensional) overlay. At its core, a text rendering system needs to be capable of displaying a sequence of character glyphs corresponding to a text string, arranged in various orientations on the screen.

A font is often implemented via a texture map known as a *glyph atlas*, which contains the various required glyphs. This texture typically consists of a single alpha channel—the value at each pixel representing the percentage of that pixel that is covered by the interior of a glyph. A font description file provides information such as the bounding boxes of each glyph within the texture, and font layout information such as kerning, baseline offsets and so on. A glyph is rendered by drawing a quad whose (u, v) coordinates correspond to the bounding box of the desired glyph within the atlas texture map. The texture map provides the alpha value, while the color is specified separately, allowing glyphs of any color to be rendered from the same atlas.

Another option for font rendering is to make use of a font library like FreeType (https://www.freetype.org/). The FreeType library enables a game or other application to read fonts in a wide variety of formats, including

TrueType (TTF) 和 OpenType (OTF)，并将字形以任意所需的点大小渲染到内存像素图中。FreeType 使用其贝塞尔曲线轮廓渲染每个字形，因此可以产生非常精确的结果。

通常，像游戏这样的实时应用程序会使用 FreeType 将必要的字形预渲染到图集中，然后该图集又用作纹理贴图，每帧将字形渲染为简单的四边形。但是，通过在引擎中嵌入 FreeType 或类似的库，可以根据需要动态地将一些字形渲染到图集中。这在渲染具有大量可能字形的语言（例如中文或韩语）的文本时非常有用。

渲染高质量字符字形的另一种方法是使用有符号距离场来描述字形。在这种方法中，字形被渲染为像素图（就像使用 FreeType 等库一样），但每个像素的值不再是 Alpha“覆盖”值。相反，每个像素包含从该像素中心到字形最近边缘的有符号距离。在字形内部，距离为负值；在字形轮廓外部，距离为正值。当从有符号距离场纹理图集渲染字形时，像素着色器会使用这些距离来计算高精度的 Alpha 值。最终结果是文本在任何距离或视角下都看起来平滑流畅。您可以在线搜索 Konstantin Käfer 的文章“在 Mapbox GL 中使用有符号距离场绘制文本”，或 Valve 的 Chris Green 撰写的文章“改进的矢量纹理和特效 Alpha 测试放大”，了解更多关于有符号距离场文本渲染的信息。

字形也可以直接根据定义它们的贝塞尔曲线轮廓进行渲染。Terathon Software LLC 的 Slug 字体渲染库在 GPU 上执行基于轮廓的字形渲染，从而使该技术适用于实时游戏应用程序。

良好的文本/字体系统必须考虑到不同语言固有的字符集和阅读方向的差异。文本字符串中字符的布局过程称为字符串整形。根据语言的不同，字符的布局方式从左到右或从右到左，每个字符都与一个公共基线对齐。字符间距部分取决于字体创建者提供的度量标准（存储在字体文件中），部分取决于字距调整规则，该规则规定了上下文字符间距的调整。

一些文本系统还提供各种有趣的功能，例如以各种方式在屏幕上为角色设置动画，以及为单个角色设置动画等等。然而，在实现游戏字体系统时，务必记住，只有那些真正需要的功能才能被重新定义。

TrueType (TTF) and OpenType (OTF), and to render glyphs into in-memory pixmaps at any desired point size. FreeType renders each glyph using its Bezier curve outlines, so it produces very accurate results.

Typically a real-time application like a game will use FreeType to prerender the necessary glyphs into an atlas, which is in turn used as a texture map to render glyphs as simple quads every frame. However, by embedding FreeType or a similar library in your engine, it's possible to render some glyphs into the atlas on the fly, on an as-needed basis. This can be useful when rendering text in a language with a very large number of possible glyphs, like Chinese or Korean.

Yet another way to render high-quality character glyphs is to use *signed distance fields* to describe the glyphs. In this approach, glyphs are rendered to pixmaps (as they would be with a library like FreeType), but the value at each pixel is no longer an alpha "coverage" value. Instead, each pixel contains a signed distance from that pixel center to the nearest edge of the glyph. Inside the glyph, the distances are negative; outside the glyph's outlines, they are positive. When rendering a glyph from a signed distance field texture atlas, the distances are used in the pixel shader to calculate highly accurate alpha values. The net result is text that looks smooth at any distance or viewing angle. You can read more about signed distance field text rendering by searching online for Konstantin Käfer's article entitled "Drawing Text with Signed Distance Fields in Mapbox GL," or the article written by Chris Green of Valve entitled "Improved Alpha-Tested Magnification for Vector Textures and Special Effects."

Glyphs can also be rendered directly from the Bézier curve outlines that define them. The Slug font rendering library by Terathon Software LLC performs its outline-based glyph rendering on the GPU, thereby making this technique practical for use in a real-time game application.

A good text/font system must account for the differences in character sets and reading directions inherent in various languages. Laying out the characters in a text string is a process known as *shaping* the string. The characters are laid out from left to right or right to left, depending on the language, with each character aligned to a common *baseline*. The spacing between characters is determined in part by metrics provided by the creator of the font (and stored in the font file), and partly by *kerning* rules that dictate contextual intercharacter spacing adjustments.

Some text systems also provide various fun features like the ability to animate characters across the screen in various ways, the ability to animate individual characters and so on. However, it's important to remember when implementing a game font system that only those features that are *actually re-*

图 11.66. CRT 伽马响应对图像质量的影响以及如何校正该影响。图片由 www.wikipedia.org 提供。

图 11.67。伽马编码和解码曲线。图片由 www.wikipedia.org 提供。

游戏所需的所有功能都应该实现。例如，如果你的游戏根本不需要显示动画文本，那么为引擎提供高级文本动画就毫无意义。

11.4.5 伽马校正

CRT 显示器对亮度值的响应往往呈非线性。也就是说，如果将线性递增的 R、G 或 B 值发送到 CRT，屏幕上显示的图像在人眼感知上将是非线性的。从视觉上看，图像的暗区会比实际暗。如图 11.66 所示。

典型 CRT 显示器的伽马响应曲线可以用以下公式简单建模

$$V_{\text{out}} = V_{\text{in}}^{\gamma}$$

其中 γ CRT > 1 。为了校正这种效应，发送到 CRT 显示器的颜色通常会经过逆变换（即使用伽马值 γ corr < 1 ）。典型 CRT 显示器的 γ CRT 值为 2.2，因此校正值通常为 γ corr $\approx$ 1/2.2 = 0.455 。这些伽马编码和解码曲线如图 11.67 所示。

3D渲染引擎可以执行伽马编码，以确保最终图像中的值经过正确的伽马校正。然而，一个问题是，用于表示纹理贴图的位图图像通常本身就经过了伽马校正。高质量的渲染引擎会考虑到这一情况，在渲染之前对纹理进行伽马解码。

Figure 11.66. The effect of a CRT's gamma response on image quality and how the effect can be corrected for. Image courtesy of www.wikipedia.org.

Figure 11.67. Gamma encoding and decoding curves. Image courtesy of www.wikipedia.org.

quired by the game should be implemented. There's no point in furnishing your engine with an advanced text animation if your game never needs to display animated text, for example.

11.4.5 Gamma Correction

CRT monitors tend to have a nonlinear response to luminance values. That is, if a linearly increasing ramp of R, G or B values were to be sent to a CRT, the image that would result on-screen would be perceptually nonlinear to the human eye. Visually, the dark regions of the image would look darker than they should. This is illustrated in Figure 11.66.

The gamma response curve of a typical CRT display can be modeled quite simply by the formula

$$V_{\text{out}} = V_{\text{in}}^{\gamma}$$

where $\gamma_{\text{CRT}} > 1$. To correct for this effect, the colors sent to the CRT display are usually passed through an inverse transformation (i.e., using a gamma value $\gamma_{\text{corr}} < 1$). The value of γ_{CRT} for a typical CRT monitor is 2.2, so the correction value is usually $\gamma_{\text{corr}} \approx 1/2.2 = 0.455$. These gamma encoding and decoding curves are shown in Figure 11.67.

Gamma encoding can be performed by the 3D rendering engine to ensure that the values in the final image are properly gamma-corrected. One problem that is encountered, however, is that the bitmap images used to represent texture maps are often gamma-corrected themselves. A high-quality rendering engine takes this fact into account, by gamma-decoding the textures prior to

渲染，然后重新编码最终渲染场景的伽玛，以便其颜色可以在屏幕上正确再现。

11.4.6 全屏后期效果

全屏后期效果是应用于渲染的三维场景，以提供额外的真实感或风格化的外观。这些效果通常通过将整个屏幕内容传递给像素着色器来实现，该着色器会应用所需的效果。这可以通过渲染一个已映射包含未过滤场景的纹理的全屏四边形来实现。以下是一些全屏后期效果的示例：

- 运动模糊。这通常是通过渲染屏幕空间速度矢量缓冲区并使用该矢量场选择性地模糊渲染图像来实现的。模糊是通过在图像上传递卷积核来实现的（有关详细信息，请参阅 Dale A. Schumacher 发表在 [5] 中的“通过离散卷积实现图像平滑和锐化”一文）。

- 景深模糊。可以通过使用深度缓冲区的内容来调整每个像素应用的模糊程度，从而产生这种模糊效果。

- 晕影。这种电影效果会降低屏幕角落处图像的亮度或饱和度，以达到戏剧效果。有时，这种效果会通过在屏幕上渲染纹理叠加来实现。这种效果的变体可用于产生经典的圆形效果，以指示玩家正在通过双筒望远镜或武器瞄准镜进行观察。

- 彩色化。屏幕像素的颜色可以以任意方式改变，作为后期处理效果。例如，除了红色之外的所有颜色都可以降低饱和度为灰色，以产生类似于《辛德勒名单》中穿红外套的小女孩的著名场景的惊人效果。

11.5 进一步阅读

本章篇幅虽短，涵盖了丰富的内容，但这只是皮毛。毫无疑问，您肯定想更深入地探索其中的许多主题。如果您想全面了解游戏和电影中三维计算机图形和动画的创作过程，我强烈推荐[27]。[2] 深入介绍了现代实时渲染的基础技术，而[16] 则是众所周知的计算机图形学相关领域的权威参考指南。

rendering and then re-encoding the gamma of the final rendered scene so that its colors can be reproduced properly on-screen.

11.4.6 Full-Screen Post Effects

Full-screen *post effects* are effects applied to a rendered three-dimensional scene that provide additional realism or a stylized look. These effects are often implemented by passing the entire contents of the screen through a pixel shader that applies the desired effect(s). This can be accomplished by rendering a full-screen quad that has been mapped with a texture containing the unfiltered scene. A few examples of full-screen post effects are given below:

- *Motion blur*. This is typically implemented by rendering a buffer of screen-space velocity vectors and using this vector field to selectively blur the rendered image. Blurring is accomplished by passing a *convolution kernel* over the image (see “Image Smoothing and Sharpening by Discrete Convolution” by Dale A. Schumacher, published in [5], for details).
- *Depth of field blur*. This blur effect can be produced by using the contents of the depth buffer to adjust the degree of blur applied at each pixel.
- *Vignette*. In this filmic effect, the brightness or saturation of the image is reduced at the corners of the screen for dramatic effect. It is sometimes implemented by literally rendering a texture overlay on top of the screen. A variation on this effect is used to produce the classic circular effect used to indicate that the player is looking through a pair of binoculars or a weapon scope.
- *Colorization*. The colors of screen pixels can be altered in arbitrary ways as a post-processing effect. For example, all colors except red could be desaturated to grey to produce a striking effect similar to the famous scene of the little girl in the red coat from *Schindler's List*.

11.5 Further Reading

We've covered a lot of material in a very short space in this chapter, but we've only just scratched the surface. No doubt you'll want to explore many of these topics in much greater detail. For an excellent overview of the entire process of creating three-dimensional computer graphics and animation for games and film, I highly recommend [27]. The technology that underlies modern real-time rendering is covered in excellent depth in [2], while [16] is well known as the definitive reference guide to all things related to computer graphics.

其他关于 3D 渲染的优秀书籍包括 [49]、[11] 和 [12]。[32] 非常详尽地介绍了 3D 渲染的数学原理。如果没有《图形精粹》系列（[20]、[5]、[28]、[22] 和 [42]）和/或《GPU 精粹》系列（[15]、[44] 和 [40]）中的一本或多本书籍，图形程序员的图书馆就不完整。当然，这份简短的参考书单仅仅是个开始——在你的游戏程序员生涯中，你无疑会遇到更多关于渲染和着色器的优秀书籍。

Other great books on 3D rendering include [49], [11] and [12]. The mathematics of 3D rendering is covered very well in [32]. No graphics programmer's library would be complete without one or more books from the *Graphics Gems* series ([20], [5], [28], [22] and [42]) and/or the *GPU Gems* series ([15], [44] and [40]). Of course, this short reference list is only the beginning—you will undoubtedly encounter a great many more excellent books on rendering and shaders over the course of your career as a game programmer.

12
动画系统

大多数现代 3D 游戏都围绕着角色展开——通常是人类或类人生物，有时是动物或外星人。角色的独特之处在于它们需要以流畅、自然的方式移动。这带来了一系列新的技术挑战，远远超出了模拟和制作车辆、抛射物、足球和俄罗斯方块等刚性物体动画所需的技术。赋予角色自然动作的任务由一个称为角色动画系统的引擎组件负责。

正如我们将看到的，动画系统为游戏设计师提供了一套强大的工具，既可以应用于角色，也可以应用于非角色。任何非100%刚性的游戏对象都可以利用动画系统。因此，每当你在游戏中看到带有活动部件的车辆、铰接式机械装置、在微风中轻轻摇曳的树木，甚至是爆炸的建筑，这些对象很可能至少部分地使用了游戏引擎的动画系统。

12.1 角色动画的类型

自《大金刚》问世以来，角色动画技术取得了长足的进步。起初，游戏采用非常简单的技术来提供逼真的动作效果。随着游戏硬件的改进，更先进的技术开始出现。

12
Animation Systems

The majority of modern 3D games revolve around *characters*—often human or humanoid, sometimes animal or alien. Characters are unique because they need to move in a fluid, organic way. This poses a host of new technical challenges, over and above what is required to simulate and animate rigid objects like vehicles, projectiles, soccer balls and Tetris pieces. The task of imbuing characters with natural-looking motion is handled by an engine component known as the *character animation system*.

As we'll see, an animation system gives game designers a powerful suite of tools that can be applied to non-characters as well as characters. Any game object that is not 100% rigid can take advantage of the animation system. So whenever you see a vehicle with moving parts, a piece of articulated machinery, trees waving gently in the breeze or even an exploding building in a game, chances are good that the object makes at least partial use of the game engine's animation system.

12.1 Types of Character Animation

Character animation technology has come a long way since *Donkey Kong*. At first, games employed very simple techniques to provide the illusion of lifelike movement. As game hardware improved, more-advanced techniques became

实时可行。如今，游戏设计师拥有众多强大的动画方法。在本节中，我们将简要回顾角色动画的演变，并概述现代游戏引擎中最常用的三种技术。

12.1.1 赛璐珞动画

所有游戏动画技术的前身被称为传统动画，或手绘动画。这是最早的动画片中使用的技术。运动的幻觉是通过快速连续地显示一系列静止图像（称为帧）来产生的。实时3D渲染可以被认为是传统动画的电子形式，它将一系列静止的全屏图像反复呈现给观看者，从而产生运动的幻觉。

赛璐珞动画是一种特殊的传统动画。赛璐珞片是一种透明的塑料片，可以在其上绘画或绘图。可以将一系列赛璐珞片动画放置在固定的背景画或绘图上，以产生运动的视觉效果，而无需反复重绘静态背景。

与赛璐珞动画等效的电子技术是一种称为精灵动画的技术。精灵是一个小位图，可以叠加在全屏背景图像上而不会破坏它，通常借助专门的图形硬件绘制。因此，精灵之于 2D 游戏动画就像赛璐珞之于传统动画。这种技术在 2D 游戏时代是必不可少的。图 12.1 展示了著名的精灵位图序列，几乎在 Mattel Intellivision 制作的每一款游戏中，它都用于产生奔跑的人形角色的幻觉。帧序列的设计使得即使无限重复也能流畅地动画 - 这称为循环动画。这种特殊的动画用现代的说法叫做跑步周期，因为它让角色看起来像是在奔跑。角色通常有许多循环动画周期，包括各种空闲周期、行走周期和跑步周期。

图 12.1. 大多数 Intellivision 游戏中使用的精灵位图序列。

feasible in real time. Today, game designers have a host of powerful animation methods at their disposal. In this section, we'll take a brief look at the evolution of character animation and outline the three most-common techniques used in modern game engines.

12.1.1 Cel Animation

The precursor to all game animation techniques is known as *traditional animation*, or *hand-drawn animation*. This is the technique used in the earliest animated cartoons. The illusion of motion is produced by displaying a sequence of still pictures known as *frames* in rapid succession. Real-time 3D rendering can be thought of as an electronic form of traditional animation, in that a sequence of still full-screen images is presented to the viewer over and over to produce the illusion of motion.

Cel animation is a specific type of traditional animation. A *cel* is a transparent sheet of plastic on which images can be painted or drawn. An animated sequence of cels can be placed on top of a fixed background painting or drawing to produce the illusion of motion without having to redraw the static background over and over.

The electronic equivalent to cel animation is a technology known as *sprite animation*. A sprite is a small bitmap that can be overlaid on top of a full-screen background image without disrupting it, often drawn with the aid of specialized graphics hardware. Hence, a sprite is to 2D game animation what a cel was to traditional animation. This technique was a staple during the 2D game era. Figure 12.1 shows the famous sequence of sprite bitmaps that were used to produce the illusion of a running humanoid character in almost every Mattel Intellivision game ever made. The sequence of frames was designed so that it animates smoothly even when it is repeated indefinitely—this is known as a *looping animation*. This particular animation would be called a *run cycle* in modern parlance, because it makes the character appear to be running. Characters typically have a number of looping animation cycles, including various idle cycles, a walk cycle and a run cycle.

Figure 12.1. The sequence of sprite bitmaps used in most Intellivision games.

12.1.2 刚性分层动画

早期的 3D 游戏，例如《毁灭战士》，继续沿用类似精灵的动画系统：游戏中的怪物只不过是面向摄像机的四边形，每个四边形都显示一系列纹理位图（称为动画纹理），以产生运动的视觉效果。这种技术至今仍用于低分辨率和/或远距离物体——例如体育场内的人群，或在背景中远距离作战的成群士兵。但对于高质量的前景角色，3D 图形带来了对改进角色动画方法的需求。

最早的 3D 角色动画方法是一种称为刚性分层动画的技术。在这种方法中，角色被建模为刚性部分的集合。人形角色的典型分解可能是骨盆、躯干、上臂、下臂、大腿、小腿、手、脚和头部。刚性部分以分层方式相互约束，类似于哺乳动物的骨骼在关节处连接的方式。这使角色可以自然地移动。例如，当移动上臂时，下臂和手将自动跟随。典型的层次结构以骨盆为根，躯干和大腿为其直属子级，依此类推，如下所示：

```
Pelvis
    Torso
        UpperRightArm
            LowerRightArm
                RightHand
        UpperLeftArm
            UpperLeftArm
                LeftHand
        Head
    UpperRightLeg
        LowerRightLeg
            RightFoot
    UpperLeftLeg
        UpperLeftLeg
            LeftFoot
```

刚性层次技术的一个大问题是，由于关节处“开裂”，角色身体的行为通常不太令人满意。如图 12.2 所示。刚性层次动画对于由刚性部件构成的机器人和机械效果很好，但当应用于“肉体”角色时，它就会失效。

12.1.2 Rigid Hierarchical Animation

Early 3D games like *Doom* continued to make use of a sprite-like animation system: Its monsters were nothing more than camera-facing quads, each of which displayed a sequence of texture bitmaps (known as an *animated texture*) to produce the illusion of motion. And this technique is still used today for low-resolution and/or distant objects—for example crowds in a stadium, or hordes of soldiers fighting a distant battle in the background. But for high-quality foreground characters, 3D graphics brought with it the need for improved character animation methods.

The earliest approach to 3D character animation is a technique known as *rigid hierarchical animation*. In this approach, a character is modeled as a collection of rigid pieces. A typical breakdown for a humanoid character might be pelvis, torso, upper arms, lower arms, upper legs, lower legs, hands, feet and head. The rigid pieces are constrained to one another in a hierarchical fashion, analogous to the manner in which a mammal's bones are connected at the joints. This allows the character to move naturally. For example, when the upper arm is moved, the lower arm and hand will automatically follow it. A typical hierarchy has the pelvis at the root, with the torso and upper legs as its immediate children and so on as shown below:

```
Pelvis
    Torso
        UpperRightArm
            LowerRightArm
                RightHand
        UpperLeftArm
            UpperLeftArm
                LeftHand
        Head
    UpperRightLeg
        LowerRightLeg
            RightFoot
    UpperLeftLeg
        UpperLeftLeg
            LeftFoot
```

The big problem with the rigid hierarchy technique is that the behavior of the character's body is often not very pleasing due to "cracking" at the joints. This is illustrated in Figure 12.2. Rigid hierarchical animation works well for robots and machinery that really are constructed of rigid parts, but it breaks down under scrutiny when applied to "fleshy" characters.

12.1.3 逐顶点动画和变形目标

刚性层次动画往往看起来不自然，因为它本身就很僵硬。我们真正想要的是一种移动单个顶点的方法，这样三角形就可以拉伸，从而产生更自然的运动效果。

实现这一点的一种方法是应用一种称为顶点动画的强力技术。在这种方法中，网格的顶点由艺术家制作动画，并导出运动数据，这些数据会告诉游戏引擎如何在运行时移动每个顶点。这种技术可以产生任何可以想象到的网格变形（仅受表面细分的限制）。然而，这是一种数据密集型技术，因为必须为网格的每个顶点存储随时间变化的运动信息。因此，它很少应用于实时游戏。

一些实时游戏中使用了这种技术的一种变体，称为变形目标动画。在这种方法中，动画师会移动网格的顶点，以创建一组相对较小的固定极端姿势。动画是通过在运行时混合两个或多个这样的固定姿势来生成的。每个顶点的位置是通过在每个极端姿势下顶点位置之间的简单线性插值 (LERP) 来计算的。

变形目标技术常用于面部动画，因为人脸是极其复杂的解剖结构，由大约 50 块肌肉驱动。变形目标动画使动画师能够完全控制面部网格的每个顶点，从而能够制作出精细或极端的动作，完美地模拟面部肌肉结构。图 12.3 展示了一组面部变形目标。

随着计算能力的不断提升，一些工作室开始使用包含数百个关节的关节式面部绑定来替代变形目标。另一些工作室则将两种技术结合起来，先使用关节式绑定来实现面部的主要姿势，然后通过变形目标进行微调。

图 12.2. 关节处开裂是刚性分层动画中的一个大问题。

12.1.3 Per-Vertex Animation and Morph Targets

Rigid hierarchical animation tends to look unnatural because it is rigid. What we really want is a way to move individual vertices so that triangles can stretch to produce more natural-looking motion.

One way to achieve this is to apply a brute-force technique known as *per-vertex animation*. In this approach, the vertices of the mesh are animated by an artist, and motion data is exported, which tells the game engine how to move each vertex at runtime. This technique can produce any mesh deformation imaginable (limited only by the tessellation of the surface). However, it is a data-intensive technique, since time-varying motion information must be stored for each vertex of the mesh. For this reason, it has little application to real-time games.

A variation on this technique known as *morph target animation* is used in some real-time games. In this approach, the vertices of a mesh are moved by an animator to create a relatively small set of fixed, extreme poses. Animations are produced by *blending* between two or more of these fixed poses at runtime. The position of each vertex is calculated using a simple linear interpolation (LERP) between the vertex's positions in each of the extreme poses.

The morph target technique is often used for facial animation, because the human face is an extremely complex piece of anatomy, driven by roughly 50 muscles. Morph target animation gives an animator full control over every vertex of a facial mesh, allowing him or her to produce both subtle and extreme movements that approximate the musculature of the face well. Figure 12.3 shows a set of facial morph targets.

As computing power continues to increase, some studios are using jointed facial rigs containing hundreds of joints as an alternative to morph targets. Other studios combine the two techniques, using jointed rigs to achieve the primary pose of the face and then applying small tweaks via morph targets.

Figure 12.2. Cracking at the joints is a big problem in rigid hierarchical animation.

12.1.4 蒙皮动画

随着游戏硬件性能的进一步提升，一种名为“蒙皮动画”的动画技术应运而生。这项技术拥有逐顶点动画和变形目标动画的诸多优势——允许动画网格的三角形变形。此外，它还拥有刚性层次动画更高效的性能和内存使用特性。它能够生成相当逼真的皮肤和衣服运动近似效果。

蒙皮动画最早出现在《超级马里奥 64》等游戏中，至今仍是游戏行业和电影行业最流行的动画技术。许多著名的现代游戏和电影角色，包括《侏罗纪公园》中的恐龙、《合金装备 4》中的索利德 · 斯内克、《指环王》中的咕噜、《神秘海域》中的内森 · 德雷克、《玩具总动员》中的巴斯光年、《战争机器》中的马库斯 · 菲尼克斯和《最后生还者》中的乔尔，全部或部分地使用了蒙皮动画技术。本章的其余部分将主要研究蒙皮/骨骼动画。

在蒙皮动画中，骨架由刚性“骨骼”构成，就像刚性层次动画一样。然而，这些刚性部分不会渲染到屏幕上，而是保持隐藏状态。一个称为蒙皮的平滑连续三角形网格与骨架的关节绑定；其顶点跟踪关节的运动。蒙皮网格的每个顶点可以加权到多个关节，因此蒙皮可以随着关节的运动而自然地拉伸。

在图 12.4 中，我们看到了 Crank the Weasel，这是 Eric Browning 于 2001 年为 Midway Home Entertainment 设计的游戏角色。Crank 的外皮由三角形网格组成，就像任何其他 3D 模型一样。然而，在他的内部，我们可以看到使他的皮肤活动起来的刚性骨骼和关节。

图 12.3.《最后生还者：重制版》中艾莉角色的一组面部变形目标（© 2014/TM SIE。由顽皮狗创建和开发，PlayStation 4）。

12.1.4 Skinned Animation

As the capabilities of game hardware improved further, an animation technology known as *skinned animation* was developed. This technique has many of the benefits of per-vertex and morph target animation—permitting the triangles of an animated mesh to deform. But it also enjoys the much more efficient performance and memory usage characteristics of rigid hierarchical animation. It is capable of producing reasonably realistic approximations to the movement of skin and clothing.

Skinned animation was first used by games like *Super Mario 64*, and it is still the most prevalent technique in use today, both by the game industry and the feature film industry. A host of famous modern game and movie characters, including the dinosaurs from *Jurrassic Park*, Solid Snake (*Metal Gear Solid 4*), Gollum (*Lord of the Rings*), Nathan Drake (*Uncharted*), Buzz Lightyear (*Toy Story*), Marcus Fenix (*Gears of War*) and Joel (*The Last of Us*) were all animated, in whole or in part, using skinned animation techniques. The remainder of this chapter will be devoted primarily to the study of skinned/skeletal animation.

In skinned animation, a *skeleton* is constructed from rigid “bones,” just as in rigid hierarchical animation. However, instead of rendering the rigid pieces on-screen, they remain hidden. A smooth continuous triangle mesh called a *skin* is bound to the joints of the skeleton; its vertices track the movements of the joints. Each vertex of the skin mesh can be weighted to multiple joints, so the skin can stretch in a natural way as the joints move.

In Figure 12.4, we see Crank the Weasel, a game character designed by Eric Browning for Midway Home Entertainment in 2001. Crank’s outer skin is composed of a mesh of triangles, just like any other 3D model. However, inside him we can see the rigid bones and joints that make his skin move.

Figure 12.3. A set of facial morph targets for the Ellie character in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4).

12.1.5 动画方法作为数据压缩技术

最灵活的动画系统，能够让动画师控制物体表面上几乎每一个微小的点。当然，这样的动画制作方式，最终会包含无限的数据量！对三角形网格顶点进行动画处理，是对这一理想的简化——实际上，我们通过限制顶点的移动，压缩了描述动画所需的信息量。（对一组控制点进行动画处理，类似于对由高阶面片构建的模型进行顶点动画处理。）变形目标可以被认为是一种额外的压缩，它通过对系统施加额外的约束来实现——顶点被限制为只能沿着固定数量的预定义顶点位置之间的线性路径移动。骨骼动画只是通过施加约束来压缩顶点动画数据的另一种方式。在这种情况下，相对大量顶点的运动被限制为跟随相对较少数量的骨骼关节的运动。

在权衡各种动画技术之间的利弊时，将它们视为压缩方法会很有帮助，这在很多方面类似于视频压缩技术。我们通常应该选择能够提供最佳压缩效果且不会产生不可接受的视觉伪影的动画方法。骨骼动画在将单个关节的运动放大为多个顶点的运动时，能够提供最佳压缩效果。角色的肢体在大多数情况下表现得像刚体，因此它们可以

图 12.4. Eric Browning 的 Crank the Weasel 角色及其内部骨骼结构。

12.1.5 Animation Methods as Data Compression Techniques

The most flexible animation system conceivable would give the animator control over literally every infinitesimal point on an object's surface. Of course, animating like this would result in an animation that contains a potentially infinite amount of data! Animating the vertices of a triangle mesh is a simplification of this ideal—in effect, we are *compressing* the amount of information needed to describe an animation by restricting ourselves to moving only the vertices. (Animating a set of control points is the analog of vertex animation for models constructed out of higher-order patches.) Morph targets can be thought of as an additional level of compression, achieved by imposing additional constraints on the system—vertices are constrained to move only along linear paths between a fixed number of predefined vertex positions. Skeletal animation is just another way to compress vertex animation data by imposing constraints. In this case, the motions of a relatively large number of vertices are constrained to follow the motions of a relatively small number of skeletal joints.

When considering the trade-offs between various animation techniques, it can be helpful to think of them as compression methods, analogous in many respects to video compression techniques. We should generally aim to select the animation method that provides the best compression without producing unacceptable visual artifacts. Skeletal animation provides the best compression when the motion of a single joint is magnified into the motions of many vertices. A character's limbs act like rigid bodies for the most part, so they can

Figure 12.4. Eric Browning's Crank the Weasel character, with internal skeletal structure.

图 12.5。该角色的骨盆关节与其他四个关节（尾部、脊柱和两条腿）相连，因此产生四块骨骼。

使用骨架可以非常高效地移动。然而，面部运动往往更加复杂，各个顶点的运动更加独立。为了使用骨架方法逼真地制作面部动画，所需的关节数量接近网格中的顶点数量，从而降低了其作为压缩技术的有效性。这就是为什么在面部动画制作中，变形目标技术通常比骨架方法更受青睐的原因之一。（另一个常见原因是，变形目标往往是动画师更自然的工作方式。）

12.2 骨骼

骨架由一层层的刚性部件（称为关节）组成。在游戏行业中，我们经常互换使用“关节”和“骨骼”这两个术语，但骨骼这个术语实际上是用词不当。从技术上讲，关节是动画师直接操纵的对象，而骨骼只是关节之间的空白处。例如，考虑 Crank the Weasel 角色模型中的骨盆关节。它是一个单关节，但由于它连接到其他四个关节（尾巴、脊椎和左右髋关节），所以这个关节看起来好像有四根骨头伸出来。图 12.5 对此进行了更详细的展示。游戏引擎不关心骨骼——只有关节才重要。所以，每当你听到行业中使用“骨骼”这个术语时，请记住，99% 的时间我们实际上都在谈论关节。

Figure 12.5. The pelvis joint of this character connects to four other joints (tail, spine and two legs), and so it produces four bones.

be moved very efficiently with a skeleton. However, the motion of a face tends to be much more complex, with the motions of individual vertices being more independent. To convincingly animate a face using the skeletal approach, the required number of joints approaches the number of vertices in the mesh, thus diminishing its effectiveness as a compression technique. This is one reason why morph target techniques are often favored over the skeletal approach for facial animation. (Another common reason is that morph targets tend to be a more natural way for animators to work.)

12.2 Skeletons

A skeleton is comprised of a *hierarchy* of rigid pieces known as *joints*. In the game industry, we often use the terms “joint” and “bone” interchangeably, but the term *bone* is actually a misnomer. Technically speaking, the joints are the objects that are directly manipulated by the animator, while the bones are simply the empty spaces between the joints. As an example, consider the pelvis joint in the Crank the Weasel character model. It is a single joint, but because it connects to four other joints (the tail, the spine and the left and right hip joints), this one joint appears to have four bones sticking out of it. This is shown in more detail in Figure 12.5. Game engines don’t care a whip about bones—only the joints matter. So whenever you hear the term “bone” being used in the industry, remember that 99% of the time we are actually speaking about joints.

12.2.1 骨骼层次结构

正如我们之前提到的，骨架中的关节构成了一个层级结构或树状结构。一个关节被选定为根关节，所有其他关节都是它的子关节、孙关节等等。蒙皮动画的典型关节层级结构看起来与典型的刚体层级结构几乎完全相同。例如，人形角色的关节层级结构可能类似于图 12.6 中所示的结构。

图 12.6. 骨骼层次结构的示例，如 Maya 的 Hypergraph Hierarchy 视图中所示。

我们通常为每个关节分配一个从 0 到 N − 1 的索引。由于每个关节有且仅有一个父关节，因此可以通过在每个关节中存储其父关节的索引来完整描述骨架的层次结构。根关节没有父关节，因此其父关节索引通常设置为无效值，例如 -1。

12.2.2 在内存中表示骨架

骨架通常由一个小型的顶层数据结构表示，该结构包含一个用于各个关节的数据结构数组。关节通常按一定顺序排列，以确保子关节在数组中始终出现在其父关节之后。这意味着关节 0 始终是骨架的根。

关节索引通常用于在动画数据结构中引用关节。例如，子关节通常通过指定其索引来引用其父关节。同样，在蒙皮三角形网格中，顶点通过索引引用与其绑定的一个或多个关节。这比通过名称引用关节更高效，无论是在所需的存储量方面（关节索引可以是 8 位宽，只要我们愿意接受每个骨架最多 256 个关节），还是在查找引用关节所需的时间方面（我们可以使用关节索引立即跳转到数组中的所需关节）。

每个关节数据结构通常包含以下信息：

- 关节的名称，可以是字符串或散列的 32 位字符串 ID。
- 骨架内关节父级的索引。
- 关节的绑定姿势逆变换。关节的绑定姿势是该关节绑定到皮肤网格顶点时的位置、方向和比例。我们通常存储此变换的逆变换，具体原因我们将在后续章节中深入探讨。

典型的骨架数据结构可能看起来像这样：

```
struct Joint
{
    Matrix4x3   m_invBindPose; // inverse bind pose
```


12.2.1 The Skeleal Hierarchy

As we've mentioned, the joints in a skeleton form a hierarchy or tree structure. One joint is selected as the root, and all other joints are its children, grandchildren and so on. A typical joint hierarchy for skinned animation looks almost identical to a typical rigid hierarchy. For example, a humanoid character's joint hierarchy might look something like the one depicted in Figure 12.6.

Figure 12.6. Example of a skeletal hierarchy, as it would appear in Maya's Hypergraph Hierarchy view.

We usually assign each joint an index from 0 to $N-1$. Because each joint has one and only one parent, the hierarchical structure of a skeleton can be fully described by storing the index of its parent with each joint. The root joint has no parent, so its parent index is usually set to an invalid value such as -1.

12.2.2 Representing a Skeleton in Memory

A skeleton is usually represented by a small top-level data structure that contains an array of data structures for the individual joints. The joints are usually listed in an order that ensures a child joint will always appear after its parent in the array. This implies that joint zero is always the root of the skeleton.

Joint indices are usually used to refer to joints within animation data structures. For example, a child joint typically refers to its parent joint by specifying its index. Likewise, in a skinned triangle mesh, a vertex refers to the joint or joints to which it is bound by index. This is much more efficient than referring to joints by name, both in terms of the amount of storage required (a joint index can be 8 bits wide, as long as we are willing to accept a maximum of 256 joints per skeleton) and in terms of the amount of time it takes to look up a referenced joint (we can use the joint index to jump immediately to a desired joint in the array).

Each joint data structure typically contains the following information:

- The *name* of the joint, either as a string or a hashed 32-bit string id.
- The *index* of the joint's *parent* within the skeleton.
- The *inverse bind pose transform* of the joint. The bind pose of a joint is the position, orientation and scale of that joint at the time it was bound to the vertices of the skin mesh. We usually store the *inverse* of this transformation for reasons we'll explore in more depth in the following sections.

A typical skeleton data structure might look something like this:

```
struct Joint
{
    Matrix4x3   m_invBindPose; // inverse bind pose
```

```
                                // transform
    const char* m_name;         // human-readable joint
                                // name
    U8          m_iParent;      // parent index or 0xFF
                                // if root
};

struct Skeleton
{
    U32     m_jointCount;  // number of joints
    Joint*  m_aJoint;      // array of joints
};
```

12.3 姿势

无论使用何种技术制作动画，无论是基于赛璐珞的动画、刚性层次动画还是蒙皮/骨骼动画，每个动画都会随着时间的推移而发生。通过将角色的身体排列成一系列离散的静止姿势，然后快速连续地显示这些姿势（通常以每秒 30 或 60 个姿势的速度），可以使角色产生运动的幻觉。（实际上，正如我们将在 12.4.1.1 节中看到的，我们经常在相邻姿势之间进行插值，而不是逐字显示单个姿势。）在骨骼动画中，骨骼的姿势直接控制网格的顶点，而姿势是动画师为角色注入生命的主要工具。因此，显然，在制作骨骼动画之前，我们必须首先了解如何设置它的姿势。

骨架的姿势是通过以任意方式旋转、平移以及可能的缩放其关节来调整的。关节的姿势定义为该关节相对于某个参考系的位置、方向和比例。关节姿势通常用 4×4 或 4×3 矩阵表示，或者用 SRT 数据结构（缩放、四元数旋转和向量平移）表示。骨架的姿势只是其所有关节姿势的集合，通常用一个简单的矩阵数组或 SRT 表示。

12.3.1 绑定姿势

图 12.7 展示了同一骨架的两种不同姿势。左侧的姿势是一种特殊姿势，称为绑定姿势，有时也称为参考姿势或静止姿势。这是 3D 网格在绑定到骨架之前的姿势（因此得名）。换句话说，如果网格被渲染为一个规则的、未蒙皮的三角形网格，且没有任何骨架，那么它将呈现这种姿势。绑定姿势也称为 T 姿势，因为


```
                                // transform
    const char* m_name;         // human-readable joint
                                // name
    U8          m_iParent;      // parent index or 0xFF
                                // if root
};

struct Skeleton
{
    U32     m_jointCount;  // number of joints
    Joint*  m_aJoint;      // array of joints
};
```

12.3 Poses

No matter what technique is used to produce an animation, be it cel-based, rigid hierarchical or skinned/skeletal, every animation takes place over time. A character is imbued with the illusion of motion by arranging the character's body into a sequence of discrete, still *poses* and then displaying those poses in rapid succession, usually at a rate of 30 or 60 *poses per second*. (Actually, as we'll see in Section 12.4.1.1, we often *interpolate* between adjacent poses rather than displaying a single pose verbatim.) In skeletal animation, the pose of the skeleton directly controls the vertices of the mesh, and posing is the animator's primary tool for breathing life into her characters. So clearly, before we can animate a skeleton, we must first understand how to *pose* it.

A skeleton is posed by rotating, translating and possibly scaling its joints in arbitrary ways. The *pose* of a joint is defined as the joint's position, orientation and scale, relative to some frame of reference. A joint pose is usually represented by a 4×4 or 4×3 matrix, or by an SRT data structure (scale, quaternion rotation and vector translation). The pose of a skeleton is just the set of all of its joints' poses and is normally represented as a simple array of matrices or SRTs.

12.3.1 Bind Pose

Two different poses of the same skeleton are shown in Figure 12.7. The pose on the left is a special pose known as the *bind pose*, also sometimes called the *reference pose* or the *rest pose*. This is the pose of the 3D mesh prior to being bound to the skeleton (hence the name). In other words, it is the pose that the mesh would assume if it were rendered as a regular, unskinned triangle mesh, without any skeleton at all. The bind pose is also called the *T-pose* because

角色通常站立时双脚略微分开，双臂伸展成字母 T 的形状。选择这种特殊的姿势是因为它可以使四肢远离身体和彼此，从而使将顶点绑定到关节的过程更容易。

图 12.7 同一骨架的两种不同姿势。左侧的姿势是一种特殊姿势，称为绑定姿势。

12.3.2 局部姿势

关节的姿势通常是相对于其父关节指定的。父关节相关姿势允许关节自然地运动。例如，如果我们旋转肩关节，但保持肘关节、腕关节和手指相对于父关节的姿势不变，那么整个手臂就会像我们预期的那样，以刚性的方式绕肩关节旋转。我们有时会使用术语“局部姿势”来描述父关节相关姿势。局部姿势几乎总是以 SRT 格式存储，其原因我们将在讨论动画混合时探讨。

从图形上看，许多 3D 创作软件（例如 Maya）将关节表示为小球体。然而，关节不仅能平移，还能旋转和缩放，因此这种可视化可能会造成一些误导。事实上，关节定义的坐标空间与我们遇到的其他空间（例如模型空间、世界空间或视图空间）在原理上并无不同。因此，最好将关节描绘成一组笛卡尔坐标轴。Maya 允许用户显示关节的局部坐标轴——如图 12.8 所示。

从数学上讲，关节姿势只不过是一种仿射变换。关节 j 的姿态可以写成 4×4 仿射变换矩阵 P j ，它由平移向量 T j 、3×3 对角尺度矩阵

the character is usually standing with his feet slightly apart and his arms outstretched in the shape of the letter T. This particular stance is chosen because it keeps the limbs away from the body and each other, making the process of binding the vertices to the joints easier.

Figure 12.7. Two different poses of the same skeleton. The pose on the left is the special pose known as *bind pose*.

12.3.2 Local Poses

A joint's pose is most often specified relative to its *parent* joint. A parent-relative pose allows a joint to move naturally. For example, if we rotate the shoulder joint, but leave the parent-relative poses of the elbow, wrist and fingers unchanged, the entire arm will rotate about the shoulder in a rigid manner, as we'd expect. We sometimes use the term *local pose* to describe a parent-relative pose. Local poses are almost always stored in SRT format, for reasons we'll explore when we discuss animation blending.

Graphically, many 3D authoring packages like Maya represent joints as small spheres. However, a joint has a rotation and a scale, not just a translation, so this visualization can be a bit misleading. In fact, a joint actually defines a coordinate space no different in principle from the other spaces we've encountered (like model space, world space or view space). So it is best to picture a joint as a set of Cartesian coordinate axes. Maya gives the user the option of displaying a joint's local coordinate axes—this is shown in Figure 12.8.

Mathematically, a joint pose is nothing more than an *affine transformation*. The pose of joint j can be written as the 4×4 affine transformation matrix $\mathbf{P}_j$, which is comprised of a translation vector $\mathbf{T}_j$, a 3×3 diagonal scale matrix

图 12.8。骨骼层次结构中的每个关节都定义了一组局部坐标空间轴，称为关节空间。

S j 和一个 3 × 3 旋转矩阵 R j 。整个骨架 P skel 的姿态可以写成所有姿态 P j 的集合，其中 j 的范围从 0 到 N − 1：

$$\mathbf{P}_j = \begin{bmatrix} \mathbf{S}_j\mathbf{R}_j & 0 \\ \mathbf{T}_j & 1 \end{bmatrix},$$

$$\mathbf{P}^{\text{skel}} = \left\{ \mathbf{P}_j \right\}\Big|_{j=0}^{N-1}.$$

12.3.2.1 联合尺度

一些游戏引擎假设关节永远不会被缩放，在这种情况下 S j 会被简单地省略并假设为单位矩阵。其他引擎假设缩放将是均匀的（如果存在），这意味着它在所有三个维度上都是相同的。在这种情况下，可以使用单个标量值 sj 表示缩放。有些引擎甚至允许非均匀缩放，在这种情况下缩放] 可以用三元素向量 sj = [s jx s jy s jz 紧凑地表示。向量 sj 的元素对应于 3×3 缩放矩阵 S j 的三个对角线元素，因此它本身并不是一个真正的向量。游戏引擎几乎从不允许剪切，因此 S j 几乎从不由完整的 3×3 缩放/剪切矩阵表示，尽管它当然可以。

在姿势或动画中省略或限制比例有很多好处。显然，使用低维比例表示可以节省内存。（均匀缩放需要每个关节每个动画帧一个浮点标量，而非均匀缩放需要三个浮点标量，而完整的 3 × 3 比例剪切矩阵则需要九个浮点标量。）将我们的引擎限制为均匀缩放还有一个额外的好处，那就是确保关节的边界球永远不会

Figure 12.8. Every joint in a skeletal hierarchy defines a set of local coordinate space axes, known as joint space.

$\mathbf{S}_j$ and a 3×3 rotation matrix $\mathbf{R}_j$. The pose of an entire skeleton $\mathbf{P}^{\text{skel}}$ can be written as the set of all poses $\mathbf{P}_j$, where j ranges from 0 to $N-1$:

$$\mathbf{P}_j = \begin{bmatrix} \mathbf{S}_j\mathbf{R}_j & 0 \\ \mathbf{T}_j & 1 \end{bmatrix},$$

$$\mathbf{P}^{\text{skel}} = \left\{ \mathbf{P}_j \right\}\Big|_{j=0}^{N-1}.$$

12.3.2.1 Joint Scale

Some game engines assume that joints will never be scaled, in which case $\mathbf{S}_j$ is simply omitted and assumed to be the identity matrix. Other engines make the assumption that scale will be *uniform* if present, meaning it is the same in all three dimensions. In this case, scale can be represented using a single scalar value s_j. Some engines even permit *nonuniform* scale, in which case scale can be compactly represented by the three-element vector $\mathbf{s}_j = \begin{bmatrix} s_{jx} & s_{jy} & s_{jz} \end{bmatrix}$. The elements of the vector $\mathbf{s}_j$ correspond to the three diagonal elements of the 3×3 scaling matrix $\mathbf{S}_j$, so it is not really a vector *per se*. Game engines almost never permit shear, so $\mathbf{S}_j$ is almost never represented by a full 3×3 scale/shear matrix, although it certainly *could* be.

There are a number of benefits to omitting or constraining scale in a pose or animation. Clearly using a lower-dimensional scale representation can save memory. (Uniform scale requires a single floating-point scalar per joint per animation frame, while nonuniform scale requires three floats, and a full 3×3 scale-shear matrix requires nine.) Restricting our engine to uniform scale has the added benefit of ensuring that the bounding sphere of a joint will never

可以转换成椭圆体，就像以非均匀方式缩放时一样。这大大简化了基于每个关节执行此类测试的引擎中的视锥体和碰撞测试的数学计算。

12.3.2.2 在内存中表示关节姿势

如上所述，关节姿势通常以 SRT 格式存储。在 C++ 中，这样的数据结构可能如下所示，其中 Q 位于首位，以确保正确对齐和最佳结构打包。（你能明白为什么吗？）

```
struct JointPose
{
    Quaternion  m_rot;   // R
    Vector3     m_trans; // T
    F32         m_scale; // S (uniform scale only)
};
```

如果允许非均匀缩放，我们可以像这样定义关节姿势：

```
struct JointPose
{
    Quaternion  m_rot;   // R
    Vector4     m_trans; // T
    Vector4     m_scale; // S
};
```

整个骨架的局部姿势可以表示如下，其中可以理解的是，数组 m_aLocalPose 是动态分配的，以包含足够多的 JointPose 出现次数，以匹配骨架中的关节数量。

```
struct SkeletonPose
{
    Skeleton*  m_pSkeleton;  // skeleton + num joints
    JointPose* m_aLocalPose; // local joint poses
};
```

12.3.2.3 关节位姿作为基础的变化

务必记住，局部关节姿势是相对于该关节的直接父关节指定的。任何仿射变换都可以被认为是将点和向量从一个坐标空间变换到另一个坐标空间。因此，当关节姿势变换 P j 应用于关节 j 坐标系中表示的点或向量时，结果将是该点或向量在父关节空间中表示的相同点或向量。

be transformed into an ellipsoid, as it could be when scaled in a nonuniform manner. This greatly simplifies the mathematics of frustum and collision tests in engines that perform such tests on a per-joint basis.

12.3.2.2 Representing a Joint Pose in Memory

As we mentioned above, joint poses are usually stored in SRT format. In C++, such a data structure might look like this, where Q is first to ensure proper alignment and optimal structure packing. (Can you see why?)

```
struct JointPose
{
    Quaternion  m_rot;   // R
    Vector3     m_trans; // T
    F32         m_scale; // S (uniform scale only)
};
```

If nonuniform scale is permitted, we might define a joint pose like this instead:

```
struct JointPose
{
    Quaternion  m_rot;   // R
    Vector4     m_trans; // T
    Vector4     m_scale; // S
};
```

The local pose of an entire skeleton can be represented as follows, where it is understood that the array `m_aLocalPose` is dynamically allocated to contain just enough occurrences of `JointPose` to match the number of joints in the skeleton.

```
struct SkeletonPose
{
    Skeleton*  m_pSkeleton;  // skeleton + num joints
    JointPose* m_aLocalPose; // local joint poses
};
```

12.3.2.3 The Joint Pose as a Change of Basis

It's important to remember that a *local* joint pose is specified relative to the joint's immediate parent. Any affine transformation can be thought of as transforming points and vectors from one coordinate space to another. So when the joint pose transform $\mathbf{P}_j$ is applied to a point or vector that is expressed in the coordinate system of the joint j, the result is that same point or vector expressed in the space of the parent joint.

正如我们在前面章节中所做的那样，我们将采用下标来表示变换方向的惯例。由于关节姿态需要从子关节空间 (C) 到其父关节空间 (P) 的点和向量，因此我们可以将其写为 (PC → P) j 。或者，我们可以引入函数 p(j) ，它返回关节 j 的父关节索引，并将关节 j 的局部姿态写为

$\mathbf{P}_{j\to p(j)}$.
有时我们需要将点和向量反向变换——从父空间变换到子关节空间。这种变换恰好是局部关节位姿的逆变换。从数学上讲，P p (j) → j =

$\left(\mathbf{P}_{j\to p(j)}\right)^{-1}$.

12.3.3 全局姿态

有时在模型空间或世界空间中表达关节的姿势会比较方便。这被称为全局姿势。有些引擎以矩阵形式表示全局姿势，而有些引擎则使用 SRT 格式。

从数学上讲，关节 (j → M) 的模型空间姿态可以通过从相关关节一直到根关节的骨骼层次结构进行遍历，并在遍历过程中乘以局部姿态 (j → p (j))。考虑图 12.9 所示的层次结构。根关节的父空间定义为模型空间，因此 p (0) ≡ M。因此，关节 J 2 的模型空间姿态可以写成如下形式：

$$\mathbf{P}_{2\to \mathrm{M}} = \mathbf{P}_{2\to 1}\mathbf{P}_{1\to 0}\mathbf{P}_{0\to \mathrm{M}}.$$

同样，关节 J 5 的模型空间姿势就是

$$\mathbf{P}_{5\to \mathrm{M}} = \mathbf{P}_{5\to 4}\mathbf{P}_{4\to 3}\mathbf{P}_{3\to 0}\mathbf{P}_{0\to \mathrm{M}}.$$

一般来说，任何关节j的全局姿态（关节到模型变换）可以写成如下形式：

$$\mathbf{P}_{j\to \mathrm{M}} = \prod_{i=j}^{0}\mathbf{P}_{i\to p(i)}, \tag{12.1}$$

其中，可以理解为，在乘积的每次迭代之后，i 变为 p (i) （关节 i 的父关节），并且 p (0) ≡ M。

12.3.3.1 在内存中表示全局位姿

我们可以扩展 SkeletonPose 数据结构以包含全局姿势，如下所示，我们再次根据骨架中的关节数量动态分配 m_aGlobalPose 数组：

As we've done in earlier chapters, we'll adopt the convention of using subscripts to denote the direction of a transformation. Since a joint pose takes points and vectors from the *child* joint's space (C) to that of its *parent* joint (P), we can write it $(\mathbf{P}_{\mathrm{C}\to\mathrm{P}})_j$. Alternatively, we can introduce the function $p(j)$, which returns the parent index of joint j, and write the local pose of joint j as $\mathbf{P}_{j\to p(j)}$.

On occasion we will need to transform points and vectors in the opposite direction—from *parent space* into the space of the *child* joint. This transformation is just the inverse of the local joint pose. Mathematically, $\mathbf{P}_{p(j)\to j} = \left(\mathbf{P}_{j\to p(j)}\right)^{-1}$.

12.3.3 Global Poses

Sometimes it is convenient to express a joint's pose in model space or world space. This is called a *global pose*. Some engines express global poses in matrix form, while others use the SRT format.

Mathematically, the model-space pose of a joint $(j \to \mathrm{M})$ can be found by walking the skeletal hierarchy from the joint in question all the way to the root, multiplying the local poses $(j \to p(j))$ as we go. Consider the hierarchy shown in Figure 12.9. The parent space of the root joint is defined to be model space, so $p(0) \equiv \mathrm{M}$. The model-space pose of joint J_2 can therefore be written as follows:

$$\mathbf{P}_{2\to \mathrm{M}} = \mathbf{P}_{2\to 1}\mathbf{P}_{1\to 0}\mathbf{P}_{0\to \mathrm{M}}.$$

Likewise, the model-space pose of joint J_5 is just

$$\mathbf{P}_{5\to \mathrm{M}} = \mathbf{P}_{5\to 4}\mathbf{P}_{4\to 3}\mathbf{P}_{3\to 0}\mathbf{P}_{0\to \mathrm{M}}.$$

In general, the global pose (joint-to-model transform) of any joint j can be written as follows:

$$\mathbf{P}_{j\to \mathrm{M}} = \prod_{i=j}^{0}\mathbf{P}_{i\to p(i)}, \tag{12.1}$$

where it is understood that i becomes $p(i)$ (the parent of joint i) after each iteration in the product, and $p(0) \equiv \mathrm{M}$.

12.3.3.1 Representing a Global Pose in Memory

We can extend our `SkeletonPose` data structure to include the global pose as follows, where again we dynamically allocate the `m_aGlobalPose` array based on the number of joints in the skeleton:

图 12.9. 可以通过从相关关节向根和模型空间原点移动层次结构来计算全局姿势，并在移动过程中连接每个关节的子到父（局部）变换。

```
struct SkeletonPose
{
    Skeleton*  m_pSkeleton;   // skeleton + num joints
    JointPose* m_aLocalPose;  // local joint poses
    Matrix44*  m_aGlobalPose; // global joint poses
};
```

12.4 剪辑

在电影中，每个场景的方方面面在制作动画之前都会经过精心策划。这包括场景中每个角色和道具的动作，甚至摄像机的移动。这意味着整个场景可以制作成一个长而连续的帧序列。而且，当角色不在镜头中时，根本不需要制作动画。

游戏动画则不同。游戏是一种互动体验，因此玩家无法预先预测角色的移动和行为。玩家可以完全控制其角色，通常也对摄像机有部分控制权。即使是计算机驱动的非玩家角色（NPC）的决策也会受到人类玩家不可预测的行为的强烈影响。因此，游戏动画几乎从来不会创建为长而连续的帧序列。相反，游戏角色的动作必须分解成大量精细的动作。我们将这些单独的动作称为动画剪辑，有时也简称为动画。

每个剪辑都会使角色执行一个明确定义的动作。
有些片段设计为循环播放，例如行走循环或跑步循环。另一些片段则设计为播放一次，例如投掷物体或绊倒摔倒在地。有些片段会影响角色的整个身体，例如角色跳到空中。其他片段则影响

Figure 12.9. A global pose can be calculated by walking the hierarchy from the joint in question towards the root and model-space origin, concatenating the child-to-parent (local) transforms of each joint as we go.

```
struct SkeletonPose
{
    Skeleton*  m_pSkeleton;   // skeleton + num joints
    JointPose* m_aLocalPose;  // local joint poses
    Matrix44*  m_aGlobalPose; // global joint poses
};
```

12.4 Clips

In a film, every aspect of each scene is carefully planned out before any animations are created. This includes the movements of every character and prop in the scene, and even the movements of the camera. This means that an entire scene can be animated as one long, contiguous sequence of frames. And characters need not be animated at all whenever they are off-camera.

Game animation is different. A game is an interactive experience, so one cannot predict beforehand how the characters are going to move and behave. The player has full control over his or her character and usually has partial control over the camera as well. Even the decisions of the computer-driven non-player characters are strongly influenced by the unpredictable actions of the human player. As such, game animations are almost never created as long, contiguous sequences of frames. Instead, a game character's movement must be broken down into a large number of fine-grained motions. We call these individual motions *animation clips*, or sometimes just *animations*.

Each clip causes the character to perform a single well-defined action. Some clips are designed to be looped—for example, a walk cycle or run cycle. Others are designed to be played once—for example, throwing an object or tripping and falling to the ground. Some clips affect the entire body of the character—the character jumping into the air for instance. Other clips affect

图 12.10. 动画的局部时间轴，显示选定时间点的姿势。图片由顽皮狗公司提供，© 2014/TM SIE。

只记录身体的某个部分——比如角色挥动右臂的动作。任何一个游戏角色的动作通常都会被分解成数千个片段。

此规则的唯一例外是当游戏角色参与游戏中的非交互部分时，称为游戏内过场动画 (IGC)、非交互序列 (NIS) 或全动态视频 (FMV)。非交互序列通常用于传达不适合交互式游戏的故事元素，其制作方式与计算机生成的电影大致相同（尽管它们通常会使用游戏内资产，例如角色网格、骨骼和纹理）。术语 IGC 和 NIS 通常指由游戏引擎本身实时渲染的非交互序列。术语 FMV 适用于已预渲染为 MP4、WMV 或其他类型电影文件并由引擎的全屏电影播放器□□在运行时播放的序列。

这种动画风格的一种变体是半交互式序列，称为快速反应事件 (QTE)。在 QTE 中，玩家必须在非交互式序列中恰好按下按钮才能看到成功动画并继续游戏；否则，系统会播放失败动画，玩家必须重试，这可能会失去生命或遭受其他后果。

12.4.1 本地时间轴

我们可以将每个动画剪辑视为一个局部时间轴，通常用独立变量 t 表示。剪辑开始时 t = 0，剪辑结束时 t = T，其中 T 是剪辑的时长。变量 t 的每个唯一值称为时间索引。图 12.10 显示了一个例子。

Figure 12.10. The local timeline of an animation showing poses at selected time indices. Images courtesy of Naughty Dog, Inc., © 2014/™ SIE.

only a part of the body—perhaps the character waving his right arm. The movements of any one game character are typically broken down into literally thousands of clips.

The only exception to this rule is when game characters are involved in a noninteractive portion of the game, known as an *in-game cinematic* (IGC), *noninteractive sequence* (NIS) or *full-motion video* (FMV). Noninteractive sequences are typically used to communicate story elements that do not lend themselves well to interactive gameplay, and they are created in much the same way computer-generated films are made (although they often make use of in-game assets like character meshes, skeletons and textures). The terms IGC and NIS typically refer to noninteractive sequences that are rendered in real time by the game engine itself. The term FMV applies to sequences that have been prerendered to an MP4, WMV or other type of movie file and are played back at runtime by the engine's full-screen movie player.

A variation on this style of animation is a semi-interactive sequence known as a *quick time event* (QTE). In a QTE, the player must hit a button at the right moment during an otherwise noninteractive sequence in order to see the success animation and proceed; otherwise, a failure animation is played, and the player must try again, possibly losing a life or suffering some other consequence as a result.

12.4.1 The Local Timeline

We can think of every animation clip as having a local timeline, usually denoted by the independent variable t. At the start of a clip, $t = 0$, and at the end, $t = T$, where T is the duration of the clip. Each unique value of the variable t is called a *time index*. An example of this is shown in Figure 12.10.

12.4.1.1 位姿插值和连续时间

重要的是要意识到，显示给观众的帧速率不一定与动画师创建姿势的速率相同。在电影和游戏动画中，动画师几乎不会每1/30秒或1/60秒就调整一次角色的姿势。相反，动画师会在片段中的特定时间生成重要的姿势，称为关键姿势或关键帧，然后计算机通过线性或基于曲线的插值来计算这些姿势之间的位置。如图12.11所示。

由于动画引擎具有插值姿势的能力（我们将在本章后面深入探讨），我们实际上可以在剪辑过程中的任何时间点采样角色的姿势——而不仅仅是在整数帧索引处。换句话说，动画剪辑的时间轴是连续的。在计算机动画中，时间变量 t 是一个实数（浮点数），而不是整数。

电影动画未能充分利用动画时间轴的连续性，因为它的帧率被锁定在每秒 24、30 或 60 帧。在电影中，观看者会在第 1、2、3 帧等等处看到角色的姿势——例如，根本不需要在第 3.7 帧处找到角色的姿势。因此，在电影动画中，动画师并不会过多（甚至根本不会）关注角色在整数帧索引之间的表现。

相比之下，实时游戏的帧率总是会略有变化，具体取决于当前 CPU 和 GPU 的负载。此外，游戏动画有时会进行时间缩放，以使角色看起来比原始动画移动得更快或更慢。因此，在实时游戏中，动画片段几乎从不以整数帧数进行采样。理论上，如果时间缩放为 1.0，则应该在第 1、2、3 帧等处进行采样。但实际上，玩家实际上可能会看到第 1.1 帧、第 1.9 帧、第 3.2 帧等。如果时间缩放为 0.5，那么玩家实际上可能会看到

图 12.11。动画师创建相对较少的关键姿势，引擎通过插值填充其余姿势。

12.4.1.1 Pose Interpolation and Continuous Time

It's important to realize that the rate at which frames are displayed to the viewer is not necessarily the same as the rate at which poses are created by the animator. In both film and game animation, the animator almost never poses the character every 1/30 or 1/60 of a second. Instead, the animator generates important poses known as *key poses* or *key frames* at specific times within the clip, and the computer calculates the poses in between via linear or curve-based interpolation. This is illustrated in Figure 12.11.

Because of the animation engine's ability to *interpolate* poses (which we'll explore in depth later in this chapter), we can actually sample the pose of the character at *any time* during the clip—not just on integer frame indices. In other words, an animation clip's timeline is *continuous*. In computer animation, the time variable *t* is a *real* (floating-point) number, not an *integer*.

Film animation doesn't take full advantage of the continuous nature of the animation timeline, because its frame rate is locked at exactly 24, 30 or 60 frames per second. In film, the viewer sees the characters' poses at frames 1, 2, 3 and so on—there's never any need to find a character's pose on frame 3.7, for example. So in film animation, the animator doesn't pay much (if any) attention to how the character looks in between the integral frame indices.

In contrast, a real-time game's frame rate always varies a little, depending on how much load is currently being placed on the CPU and GPU. Also, game animations are sometimes *time-scaled* in order to make the character appear to move faster or slower than originally animated. So in a real-time game, an animation clip is almost *never* sampled on integer frame numbers. In theory, with a time scale of 1.0, a clip should be sampled at frames 1, 2, 3 and so on. But in practice, the player might actually see frames 1.1, 1.9, 3.2 and so on. And if the time scale is 0.5, then the player might actually see frames

Figure 12.11. An animator creates a relatively small number of key poses, and the engine fills in the rest of the poses via interpolation.

图 12.12。以每秒 30 帧采样的一秒钟动画持续 30 帧，由 31 个样本组成。

1.1、1.4、1.9、2.6、3.2等等。负时间刻度甚至可以用来反向播放动画。因此，在游戏动画中，时间既是连续的，又是可伸缩的。

12.4.1.2 时间单位

由于动画的时间轴是连续的，因此最好以秒为单位来测量时间。假设我们预先定义了帧的持续时间，时间也可以以帧为单位来测量。对于游戏动画，典型的帧持续时间为 1/30 或 1/60 秒。但是，重要的是不要错误地将时间变量 t 定义为计算整帧的整数。无论选择哪种时间单位，t 都应该是实数（浮点数）、定点数或测量非常小的子帧时间间隔的整数。目标是在时间测量中具有足够的分辨率，以便在帧之间进行“补间”或缩放动画的播放速度等操作。

12.4.1.3 框架与样本

不幸的是，“帧”一词在游戏行业中有着多种常见含义。这可能会导致很多混淆。有时，一帧被认为是一段持续时间为 1/30 或 1/60 秒的时间段。但在其他情况下，“帧”一词指的是单个时间点（例如，我们可能会说角色在第 42 帧时的姿势）。

我个人更喜欢使用术语样本来指代单个时间点，而保留使用帧一词来描述持续时间为 1/30 或 1/60 秒的时间段。例如，以每秒 30 帧的速率创建的一秒钟动画将包含 31 个样本，持续时间为 30 帧，如图 12.12 所示。术语“样本”来自信号处理领域。连续时间信号（即函数 f(t)）可以通过以均匀的时间间隔对该信号进行采样来转换为一组离散数据点。有关采样的更多信息，请参见第 14.3.2.1 节。

Figure 12.12. A one-second animation sampled at 30 frames per second is 30 frames in duration and consists of 31 samples.

1.1, 1.4, 1.9, 2.6, 3.2 and so on. A negative time scale can even be used to play an animation in reverse. So in game animation, time is both *continuous* and *scalable*.

12.4.1.2 Time Units

Because an animation's timeline is continuous, time is best measured in units of seconds. Time can also be measured in units of *frames*, presuming we define the duration of a frame beforehand. Typical frame durations are 1/30 or 1/60 of a second for game animation. However, it's important not to make the mistake of defining your time variable t as an integer that counts whole frames. No matter which time units are selected, t should be a real (floating-point) quantity, a fixed-point number or an integer that measures very small subframe time intervals. The goal is to have sufficient resolution in your time measurements for doing things like "tweening" between frames or scaling an animation's playback speed.

12.4.1.3 Frame versus Sample

Unfortunately, the term *frame* has more than one common meaning in the game industry. This can lead to a great deal of confusion. Sometimes a frame is taken to be a *period of time* that is 1/30 or 1/60 of a second in duration. But in other contexts, the term frame is applied to a *single point in time* (e.g., we might speak of the pose of the character "at frame 42").

I personally prefer to use the term *sample* to refer to a single point in time, and I reserve the word *frame* to describe a time period that is 1/30 or 1/60 of a second in duration. So for example, a one-second animation created at a rate of 30 frames per second would consist of 31 *samples* and would be 30 *frames* in duration, as shown in Figure 12.12. The term "sample" comes from the field of signal processing. A continuous-time signal (i.e., a function $f(t)$) can be converted into a set of discrete data points by sampling that signal at uniformly spaced time intervals. See Section 14.3.2.1 for more information on sampling.

12.4.1.4 帧、样本和循环剪辑

当一个片段被设计成反复播放时，我们称其为循环片段。假设将一个 1 秒（30 帧/31 个样本）的片段的两个副本前后摆放，则第一个片段的样本 31 将与第二个片段的样本 1 在时间上完全重合，如图 12.13 所示。为了使片段能够正确循环，我们可以看到，片段结尾处角色的姿势必须与开头的姿势完全匹配。这反过来意味着循环片段的最后一个样本（在我们的例子中是样本 31）是多余的。因此，许多游戏引擎会省略循环片段的最后一个样本。

这使我们得出以下控制任何动画剪辑中的样本和帧数量的规则：

- 如果剪辑是非循环的，则 N 帧动画将具有 N + 1 个唯一样本。
- 如果剪辑正在循环，则最后一个样本是多余的，因此 N 帧动画将具有 N 个唯一样本。

图 12.13. 循环剪辑的最后一个样本与其第一个样本的时间一致，因此是多余的。

12.4.1.5 标准化时间（相位）

有时使用标准化的时间单位 u 会比较方便，例如，无论动画的持续时间 T 是多少，u 在动画开始时都等于 0，在动画结束时都等于 1。我们有时将标准化的时间称为动画剪辑的相位，因为 u 在动画循环播放时的作用类似于正弦波的相位。

如图 12.14 所示。

当同步两个或多个动画剪辑（这些剪辑的绝对时长不一定相同）时，标准化时间非常有用。例如，我们可能希望将 2 秒（60 帧）的跑步循环平滑地淡入淡出为 3 秒（90 帧）的行走循环。为了使淡入淡出效果良好，我们需要确保两个动画始终保持同步，以便

12.4.1.4 Frames, Samples and Looping Clips

When a clip is designed to be played over and over repeatedly, we say it is *looped*. If we imagine two copies of a 1 s (30-frame/31-sample) clip laid back-to-front, then sample 31 of the first clip will coincide exactly in time with sample 1 of the second clip, as shown in Figure 12.13. For a clip to loop properly, then, we can see that the pose of the character at the end of the clip must exactly match the pose at the beginning. This, in turn, implies that the last sample of a looping clip (in our example, sample 31) is redundant. Many game engines therefore omit the last sample of a looping clip.

This leads us to the following rules governing the number of samples and frames in any animation clip:

- If a clip is *non-looping*, an N-frame animation will have $N + 1$ unique samples.
- If a clip is *looping*, then the last sample is redundant, so an N-frame animation will have N unique samples.

Figure 12.13. The last sample of a looping clip coincides in time with its first sample and is, therefore, redundant.

12.4.1.5 Normalized Time (Phase)

It is sometimes convenient to employ a normalized time unit u, such that $u = 0$ at the start of the animation, and $u = 1$ at the end, no matter what its duration T may be. We sometimes refer to normalized time as the *phase* of the animation clip, because u acts like the phase of a sine wave when the animation is looped. This is illustrated in Figure 12.14.

Normalized time is useful when synchronizing two or more animation clips that are not necessarily of the same absolute duration. For example, we might want to smoothly cross-fade from a 2-second (60-frame) run cycle into a 3-second (90-frame) walk cycle. To make the cross-fade look good, we want to ensure that the two animations remain synchronized at all times, so that the

图 12.14. 动画片段，展示标准化的时间单位。图片由顽皮狗公司提供，© 2014/TM SIE。

确保两个片段中的双脚正确对齐。我们只需将行走片段的标准化起始时间 u walk 设置为与跑步片段的标准化时间索引 u run 匹配即可实现这一点。然后，我们以相同的标准化速率推进两个片段，使它们保持同步。这比使用绝对时间索引 t walk 和 t run 进行同步要容易得多，而且不容易出错。

12.4.2 全球时间线

正如每个动画片段都有一个局部时间轴（其时钟从片段开头的 0 开始），游戏中的每个角色也都有一个全局时间轴（其时钟从角色首次生成到游戏世界时开始，或者从关卡或整个游戏的开始开始）。在本书中，我们将使用时间变量 τ 来测量全局时间，以免将其与局部时间变量 t 混淆。

我们可以将播放动画视为简单地将该剪辑的局部时间轴映射到角色的全局时间轴。例如，图 12.15 展示了从全局时间 τ start = 102 秒开始播放动画剪辑 A。

正如我们上面所见，播放循环动画就像将无限数量的片段副本（从后向前）放置在全局时间轴上。我们也可以想象循环播放有限次动画，这相当于放置有限数量的片段副本。如图 12.16 所示。

图 12.15.从全局时间 102 秒开始播放动画剪辑 A。

Figure 12.14. An animation clip, showing normalized time units. Images courtesy of Naughty Dog, Inc., © 2014/™ SIE.

feet line up properly in both clips. We can accomplish this by simply setting the normalized start time of the walk clip, u_{walk}, to match the normalized time index of the run clip, u_{run}. We then advance both clips at the same normalized rate so that they remain in sync. This is quite a bit easier and less error-prone than doing the synchronization using the absolute time indices t_{walk} and t_{run}.

12.4.2 The Global Timeline

Just as every animation clip has a local timeline (whose clock starts at 0 at the beginning of the clip), every character in a game has a global timeline (whose clock starts when the character is first spawned into the game world, or perhaps at the start of the level or the entire game). In this book, we'll use the time variable τ to measure global time, so as not to confuse it with the local time variable t.

We can think of *playing* an animation as simply *mapping* that clip's local timeline onto the character's global timeline. For example, Figure 12.15 illustrates playing animation clip A starting at a global time of $\tau_{\text{start}} = 102$ seconds.

As we saw above, playing a looping animation is like laying down an infinite number of back-to-front copies of the clip onto the global timeline. We can also imagine looping an animation a finite number of times, which corresponds to laying down a finite number of copies of the clip. This is illustrated in Figure 12.16.

Figure 12.15. Playing animation clip A starting at a global time of 102 seconds.

图 12.16 播放循环动画相当于连续放置多个剪辑副本。

对剪辑进行时间缩放，使其播放速度看起来比原始动画更快或更慢。为此，我们只需在将剪辑图像放到全局时间轴上时对其进行缩放即可。时间缩放最自然地表示为播放速率，我们将其表示为 R 。例如，如果动画要以两倍速度播放（R = 2），那么在将其映射到全局时间轴时，我们会将剪辑的本地时间轴缩放为其正常长度的一半（1/R = 0.5）。如图 12.17 所示。

反向播放剪辑相当于使用时间尺度 − 1，如图 12.18 所示。

为了将动画剪辑映射到全局时间轴上，我们需要有关该剪辑的以下信息：

- 其全局启动时间τstart，

图 12.17.以两倍速度播放动画相当于将其本地时间轴缩放 1/2 倍。

图 12.18。反向播放剪辑对应的时间尺度为 −1。

Figure 12.16. Playing a looping animation corresponds to laying down multiple back-to-back copies of the clip.

Time-scaling a clip makes it appear to play back more quickly or more slowly than originally animated. To accomplish this, we simply scale the image of the clip when it is laid down onto the global timeline. Time-scaling is most naturally expressed as a *playback rate*, which we'll denote R. For example, if an animation is to play back at twice the speed ($R = 2$), then we would scale the clip's local timeline to one-half ($1/R = 0.5$) of its normal length when mapping it onto the global timeline. This is shown in Figure 12.17.

Playing a clip in reverse corresponds to using a time scale of -1, as shown in Figure 12.18.

In order to map an animation clip onto a global timeline, we need the following pieces of information about the clip:

- its global start time τ_{start},

Figure 12.17. Playing an animation at twice the speed corresponds to scaling its local timeline by a factor of 1/2.

Figure 12.18. Playing a clip in reverse corresponds to a time scale of -1.

- 其播放速率 R ，
- 其持续时间 T ，以及
- 循环的次数，我们将其表示为 N 。

有了这些信息，我们可以使用以下两个关系将任何全局时间 τ 映射到相应的局部时间 t，反之亦然：

$$t = (\tau - \tau_{\text{start}})R, \tag{12.2}$$
$$\tau = \tau_{\text{start}} + \frac{1}{R}t.$$

如果动画不循环（N = 1），那么我们应该将 t 限制在有效范围 [0, T] 内，然后再使用它从剪辑中采样姿势：

$$t = \text{clamp}\left[(\tau - \tau_{\text{start}})R\right]\Big|_0^T.$$

如果动画无限循环（N = ∞），则我们将结果除以持续时间 T 后取余数，使 t 处于有效范围内。这可以通过模运算符（mod，或 C/C++ 中的 % ）实现，如下所示：

$$t =)^{(}\text{R}^{(\tau-\tau\text{开始}} \bmod T.$$

如果剪辑循环有限次（1 < N < ∞），我们必须首先将 t 限制在范围 [0, NT] 内，然后将该结果除以 T，以使 t 进入对剪辑进行采样的有效范围：

$$t = \left(\text{clamp}\left[(\tau - \tau_{\text{start}})R\right]\Big|_0^{NT}\right) \bmod T.$$

大多数游戏引擎直接使用本地动画时间轴，而不直接使用全局时间轴。然而，直接使用全局时间轴可以带来一些非常有用的好处。首先，它使动画同步变得非常简单。

12.4.3 本地时钟和全局时钟的比较

动画系统必须跟踪当前正在播放的每个动画的时间索引。为此，我们有两个选择：

- 本地时钟。在这种方法中，每个剪辑都有自己的本地时钟，通常用以秒或帧为单位存储的浮点时间索引表示，或者以标准化的时间单位存储（在这种情况下，它通常被称为动画的阶段）。在剪辑开始播放时，本地

- its playback rate R,
- its duration T, and
- the number of times it should loop, which we'll denote N.

Given this information, we can map from any global time τ to the corresponding local time t, and vice versa, using the following two relations:

$$t = (\tau - \tau_{\text{start}})R, \tag{12.2}$$
$$\tau = \tau_{\text{start}} + \frac{1}{R}t.$$

If the animation doesn't loop ($N = 1$), then we should clamp t into the valid range $[0, T]$ before using it to sample a pose from the clip:

$$t = \text{clamp}\left[(\tau - \tau_{\text{start}})R\right]\Big|_0^T.$$

If the animation loops forever ($N = \infty$), then we bring t into the valid range by taking the *remainder* of the result after dividing by the duration T. This is accomplished via the *modulo* operator (mod, or % in C/C++), as shown below:

$$t = \big((\tau - \tau_{\text{start}})R\big) \bmod T.$$

If the clip loops a *finite* number of times ($1 < N < \infty$), we must first clamp t into the range $[0, NT]$ and then modulo *that* result by T in order to bring t into a valid range for sampling the clip:

$$t = \left(\text{clamp}\left[(\tau - \tau_{\text{start}})R\right]\Big|_0^{NT}\right) \bmod T.$$

Most game engines work directly with local animation timelines and don't use the global timeline directly. However, working directly in terms of global times can have some incredibly useful benefits. For one thing, it makes synchronizing animations trivial.

12.4.3 Comparison of Local and Global Clocks

The animation system must keep track of the time indices of every animation that is currently playing. To do so, we have two choices:

- *Local clock.* In this approach, each clip has its own local clock, usually represented by a floating-point time index stored in units of seconds or frames, or in normalized time units (in which case it is often called the *phase* of the animation). At the moment the clip begins to play, the local

时间索引 t 通常取零。为了使动画在时间上向前推进，我们会分别推进每个剪辑的本地时钟。如果剪辑的播放速率为非单位速率 R ，则其本地时钟推进的量必须按 R 进行缩放。

- 全局时钟。在这种方法中，角色有一个全局时钟，通常以秒为单位，每个剪辑仅记录其开始播放的全局时间τstart。剪辑的本地时钟可根据此信息使用公式（12.2）计算得出。

本地时钟方法的优点是简单，是设计动画系统时最显而易见的选择。然而，全局时钟方法也有一些明显的优势，尤其是在同步动画时，无论是在单个角色的上下文中，还是在场景中的多个角色之间。

12.4.3.1 使用本地时钟同步动画

我们说，在本地时钟方法中，片段本地时间轴的原点（t = 0）通常定义为与片段开始播放的时刻一致。因此，要同步两个或多个片段，它们必须在游戏时间中完全相同的时刻播放。这看起来很简单，但当用于播放动画的命令来自不同的引擎子系统时，就会变得相当棘手。

例如，假设我们想要将玩家角色的出拳动画与非玩家角色相应的击打反应动画同步。问题在于，玩家的出拳动作是由玩家子系统在检测到游戏手柄上的按钮被按下后触发的。同时，非玩家角色 (NPC) 的击打反应动画由人工智能 (AI) 子系统播放。如果在游戏循环中，AI 代码在玩家代码之前运行，则玩家出拳和 NPC 反应之间会有一帧的延迟。如果玩家代码在 AI 代码之前运行，那么当 NPC 试图击打玩家时，就会出现相反的问题。如果使用消息传递（事件）系统在两个子系统之间进行通信，则可能会产生额外的延迟（有关更多详细信息，请参见第 16.8 节）。图 12.19 说明了这个问题。

```
void GameLoop()
{
    while (!quit)
    {
        // preliminary updates...
```


time index t is usually taken to be zero. To advance the animations forward in time, we advance the local clocks of each clip individually. If a clip has a non-unit playback rate R, the amount by which its local clock advances must be scaled by R.

- *Global clock.* In this approach, the character has a global clock, usually measured in seconds, and each clip simply records the global time at which it started playing, τ_{start}. The clips' local clocks are *calculated* from this information using Equation (12.2).

The local clock approach has the benefit of being simple, and it is the most obvious choice when designing an animation system. However, the global clock approach has some distinct advantages, especially when it comes to synchronizing animations, either within the context of a single character or across multiple characters in a scene.

12.4.3.1 Synchronizing Animations with a Local Clock

With a local clock approach, we said that the origin of a clip's local timeline ($t = 0$) is usually defined to coincide with the moment at which the clip starts playing. Thus, to synchronize two or more clips, they must be played at exactly the same moment in game time. This seems simple enough, but it can become quite tricky when the commands used to play the animations are coming from disparate engine subsystems.

For example, let's say we want to synchronize the player character's punch animation with a non-player character's corresponding hit reaction animation. The problem is that the player's punch is initiated by the player subsystem in response to detecting that a button was hit on the joy pad. Meanwhile, the non-player character's (NPC) hit reaction animation is played by the artificial intelligence (AI) subsystem. If the AI code runs *before* the player code in the game loop, there will be a one-frame delay between the start of the player's punch and the start of the NPC's reaction. And if the player code runs before the AI code, then the opposite problem occurs when an NPC tries to punch the player. If a message-passing (event) system is used to communicate between the two subsystems, additional delays might be incurred (see Section 16.8 for more details). This problem is illustrated in Figure 12.19.

```
void GameLoop()
{
    while (!quit)
    {
        // preliminary updates...
```

```
        UpdateAllNpcs(); // react to punch event
                         // from last frame

        // more updates...

        UpdatePlayer(); // punch button hit - start punch
                        // anim, and send event to NPC to
                        // react

        // still more updates...
    }
}
```

图 12.19.当使用本地时钟时，不同游戏系统的执行顺序可能会引入动画同步问题。

12.4.3.2 使用全局时钟同步动画

全局时钟方法有助于缓解许多此类同步问题，因为根据定义，时间线的原点（τ= 0）在所有剪辑中都是相同的。如果两个或多个动画的全局开始时间在数值上相等，则剪辑将完全同步开始。如果它们的播放速率也相等，那么它们将保持同步而不会漂移。播放每个动画的代码何时执行不再重要。即使播放命中反应的 AI 代码最终比玩家的出拳代码晚一帧运行，仍然可以轻松地通过简单地记下出拳的全局开始时间并设置反应动画的全局开始时间来匹配它来保持两个剪辑同步。如图 12.20 所示。

当然，我们需要确保两个角色的全局时钟匹配，但这很容易做到。我们可以将全局开始时间调整为


```
        UpdateAllNpcs(); // react to punch event
                         // from last frame

        // more updates...

        UpdatePlayer(); // punch button hit - start punch
                        // anim, and send event to NPC to
                        // react

        // still more updates...
    }
}
```

Figure 12.19. The order of execution of disparate gameplay systems can introduce animation synchronization problems when local clocks are used.

12.4.3.2 Synchronizing Animations with a Global Clock

A global clock approach helps to alleviate many of these synchronization problems, because the origin of the timeline ($\tau = 0$) is common across all clips by definition. If two or more animations' global start times are numerically equal, the clips will start in perfect synchronization. If their playback rates are also equal, then they will remain in sync with no drift. It no longer matters *when* the code that plays each animation executes. Even if the AI code that plays the hit reaction ends up running a frame later than the player's punch code, it is still trivial to keep the two clips in sync by simply noting the global start time of the punch and setting the global start time of the reaction animation to match it. This is shown in Figure 12.20.

Of course, we do need to ensure that the two characters' global clocks match, but this is trivial to do. We can either adjust the global start times to

图 12.20。全局时钟方法可以缓解动画同步问题。

考虑到角色时钟的任何差异，或者我们可以简单地让游戏中的所有角色共享一个主时钟。

12.4.4 简单的动画数据格式

通常，动画数据是从 Maya 场景文件中提取的，方法是以每秒 30 或 60 个样本的速率离散地对骨骼的姿势进行采样。一个样本包含骨骼中每个关节的完整姿势。姿势通常以 SRT 格式存储：对于每个关节 j ，缩放分量是单个浮点标量 S j 或三元素向量 S j = [S jx S jy S jz] 。旋转分量当然是一个四元素四元数 Q j = [Q jx Q jy Q jz Q jw] 。平移分量是一个三元素向量 T j =

图 12.21.未压缩的动画剪辑每个样本、每个关节包含 10 个浮点数据通道。

Figure 12.20. A global clock approach can alleviate animation synchronization problems.

take account of any differences in the characters' clocks, or we can simply have all characters in the game share a single master clock.

12.4.4 A Simple Animation Data Format

Typically, animation data is extracted from a Maya scene file by sampling the pose of the skeleton discretely at a rate of 30 or 60 samples per second. A sample comprises a full pose for each joint in the skeleton. The poses are usually stored in SRT format: For each joint j, the scale component is either a single floating-point scalar S_j or a three-element vector $\mathbf{S}_j = \begin{bmatrix} S_{jx} & S_{jy} & S_{jz} \end{bmatrix}$. The rotational component is of course a four-element quaternion $\mathrm{Q}_j = [Q_{jx} \quad Q_{jy} \quad Q_{jz} \quad Q_{jw}]$. And the translational component is a three-element vector $\mathbf{T}_j =$

Figure 12.21. An uncompressed animation clip contains 10 channels of floating-point data per sample, per joint.

我们有时会说，一个动画每个关节最多由 10 个通道组成，参考 S j 、 Q j 和 T j 的 10 个分量。如图 12.21 所示。

在 C++ 中，动画剪辑可以用多种不同的方式表示。以下是其中一种可能：

```
struct JointPose { ... }; // SRT, defined as above

struct AnimationSample
{
    JointPose*       m_aJointPose; // array of joint
                                   // poses
};

struct AnimationClip
{
    Skeleton*        m_pSkeleton;
    F32              m_framesPerSecond;
    U32              m_frameCount;
    AnimationSample* m_aSamples; // array of samples
    bool             m_isLooping;
};
```

动画剪辑是为特定骨架编写的，通常不适用于其他骨架。因此，我们的示例 AnimationClip 数据结构包含对其骨架的引用 m_pSkeleton 。（在实际引擎中，这可能是一个唯一的骨架 ID，而不是 Skeleton* 指针。在这种情况下，引擎可能会提供一种通过唯一 ID 快速便捷地查找骨架的方法。）

假定每个样本中 m_aJointPose 数组中的 JointPose 数量与骨架中的关节数量匹配。m_aSamples 数组中的样本数量取决于帧数以及剪辑是否循环播放。对于非循环动画，样本数量为 (m_frameCount + 1)。但是，如果动画循环播放，则最后一个样本与第一个样本相同，通常会被省略。在这种情况下，样本数量等于 m_frameCount。

需要注意的是，在实际的游戏引擎中，动画数据并非以这种简单的格式存储。正如我们将在 12.8 节中看到的那样，数据通常会以各种方式进行压缩以节省内存。

12.4.5 连续通道函数

动画剪辑的样本实际上只是随时间变化的连续函数的定义。你可以把它们想象成 10 个标量值时间函数

$\begin{bmatrix} T_{jx} & T_{jy} & T_{jz} \end{bmatrix}$. We sometimes say that an animation consists of up to 10 *channels* per joint, in reference to the 10 components of $\mathbf{S}_j$, Q_j, and $\mathbf{T}_j$. This is illustrated in Figure 12.21.

In C++, an animation clip can be represented in many different ways. Here is one possibility:

```
struct JointPose { ... }; // SRT, defined as above

struct AnimationSample
{
    JointPose*       m_aJointPose; // array of joint
                                   // poses
};

struct AnimationClip
{
    Skeleton*        m_pSkeleton;
    F32              m_framesPerSecond;
    U32              m_frameCount;
    AnimationSample* m_aSamples; // array of samples
    bool             m_isLooping;
};
```

An animation clip is authored for a specific skeleton and generally won't work on any other skeleton. As such, our example `AnimationClip` data structure contains a reference to its skeleton, `m_pSkeleton`. (In a real engine, this might be a unique skeleton id rather than a `Skeleton*` pointer. In this case, the engine would presumably provide a way to quickly and conveniently look up a skeleton by its unique id.)

The number of `JointPoses` in the `m_aJointPose` array within each sample is presumed to match the number of joints in the skeleton. The number of samples in the `m_aSamples` array is dictated by the frame count and by whether or not the clip is intended to loop. For a non-looping animation, the number of samples is `(m_frameCount + 1)`. However, if the animation loops, then the last sample is identical to the first sample and is usually omitted. In this case, the sample count is equal to `m_frameCount`.

It's important to realize that in a real game engine, animation data isn't actually stored in this simplistic format. As we'll see in Section 12.8, the data is usually *compressed* in various ways to save memory.

12.4.5 Continuous Channel Functions

The samples of an animation clip are really just definitions of continuous functions over time. You can think of these as 10 scalar-valued functions of time

每个关节，或者每个关节两个向量值函数和一个四元值函数。理论上，这些通道函数在整个剪辑的局部时间轴上是平滑连续的，如图 12.22 所示（除了明确设计的不连续性，例如镜头切换）。然而，实际上，许多游戏引擎会在样本之间进行线性插值，在这种情况下，实际使用的函数是对底层连续函数的分段线性近似。如图 12.23 所示。

12.4.6 元通道

许多游戏允许为动画定义额外的数据“元通道”。这些通道可以编码游戏特定的信息，这些信息与骨骼姿势无直接关系，但需要与动画同步。

定义一个包含不同时间索引的事件触发器的特殊通道是很常见的，如图 12.24 所示。每当动画的本地时间索引经过其中一个触发器时，就会向游戏引擎发送一个事件，游戏引擎可以根据需要做出响应。（我们将在第 16 章详细讨论事件。）事件触发器的一个常见用途是指示动画过程中的哪些时间点

图 12.22。剪辑中的动画样本定义了随时间推移的连续函数。

图 12.23。许多游戏引擎在插入通道函数时使用分段线性近似。

per joint, or as two vector-valued functions and one quaternion-valued function per joint. Theoretically, these *channel functions* are smooth and continuous across the entire clip's local timeline, as shown in Figure 12.22 (with the exception of explicitly authored discontinuities like camera cuts). In practice, however, many game engines interpolate *linearly* between the samples, in which case the functions actually used are *piecewise linear approximations* to the underlying continuous functions. This is depicted in Figure 12.23.

12.4.6 Metachannels

Many games permit additional "metachannels" of data to be defined for an animation. These channels can encode game-specific information that doesn't have to do directly with posing the skeleton but which needs to be synchronized with the animation.

It is quite common to define a special channel that contains *event triggers* at various time indices, as shown in Figure 12.24. Whenever the animation's local time index passes one of these triggers, an *event* is sent to the game engine, which can respond as it sees fit. (We'll discuss events in detail in Chapter 16.) One common use of event triggers is to denote at which points during the

Figure 12.22. The animation samples in a clip define continuous functions over time.

Figure 12.23. Many game engines use a piecewise linear approximation when interpolating channel functions.

图 12.24. 可以向动画剪辑添加特殊事件触发通道，以便将声音效果、粒子效果和其他游戏事件与动画同步。

动画应该播放特定的声音或粒子效果。例如，当左脚或右脚接触地面时，可以触发脚步声和“尘土飞扬”的粒子效果。

另一种常见做法是允许特殊关节（在 Maya 中称为定位器）与骨架本身的关节一起动画。由于关节或定位器只是一种仿射变换，因此这些特殊关节可以用来编码游戏中几乎任何物体的位置和方向。

动画定位器的一个典型应用是指定游戏摄像机在动画过程中的定位和方向。在 Maya 中，定位器被约束到摄像机，然后该摄像机会随着场景中角色的关节而移动。摄像机的定位器会被导出并在游戏中使用，以便在动画过程中移动游戏摄像机。还可以通过将相关数据放入一个或多个额外的浮点通道中，来为摄像机的视野（焦距）以及其他可能的摄像机属性设置动画。

非联合动画通道的其他示例包括：

- 纹理坐标滚动，
- 纹理动画（纹理坐标滚动的一种特殊情况，其中帧在纹理内线性排列，并且每次迭代时纹理都会滚动一个完整的帧），

Figure 12.24. A special event trigger channel can be added to an animation clip in order to synchronize sound effects, particle effects and other game events with an animation.

animation certain sound or particle effects should be played. For example, when the left or right foot touches the ground, a footstep sound and a “cloud of dust” particle effect could be initiated.

Another common practice is to permit special joints, known in Maya as *locators*, to be animated along with the joints of the skeleton itself. Because a joint or locator is just an affine transform, these special joints can be used to encode the position and orientation of virtually any object in the game.

A typical application of animated locators is to specify how the game’s camera should be positioned and oriented during an animation. In Maya, a locator is constrained to a camera, and the camera is then animated along with the joints of the character(s) in the scene. The camera’s locator is exported and used in-game to move the game’s camera around during the animation. The field of view (focal length) of the camera, and possibly other camera attributes, can also be animated by placing the relevant data into one or more additional *floating-point channels*.

Other examples of non-joint animation channels include:

- texture coordinate scrolling,
- texture animation (a special case of texture coordinate scrolling in which frames are arranged linearly within a texture, and the texture is scrolled by one complete frame at each iteration),

- 动画材质参数（颜色、镜面反射、透明度等），
- 动画照明参数（半径、锥角、强度、颜色等）以及
- 任何其他需要随时间变化并以某种方式与动画同步的参数。

12.4.7 网格、骨架和剪辑之间的关系

图 12.25 中的 UML 图显示了动画剪辑数据如何与游戏引擎中的骨架、姿势、网格和其他数据交互。要特别注意这些类之间关系的基数和方向。基数显示在类之间关系箭头的尖端或尾部旁边 - 1 表示该类的单个实例，而星号表示多个实例。对于任何一种类型的角色，都会有一个骨架、一个或多个网格和一个或多个动画剪辑。骨架是统一的中心元素 - 皮肤附着在骨架上，但与动画剪辑没有任何关系。同样，剪辑针对特定的骨架，但它们对皮肤网格没有任何“了解”。图 12.26 说明了这些关系。

游戏设计师通常会尽量减少游戏中独特骨架的数量，因为每个新骨架通常都需要一整套新的动画剪辑。为了营造出多种不同类型角色的视觉效果，通常最好尽可能创建多个蒙皮到同一骨架的网格，以便所有角色可以共享同一套动画。

12.4.7.1 动画重定向

我们上面提到，一个动画通常只兼容单个骨架。这个限制可以通过动画重定向技术来克服。

重定向是指使用为一个骨架创作的动画来为另一个骨架制作动画。如果两个骨架在形态上完全相同，重定向可能归结为简单的关节索引重映射。但是，当两个骨架不完全匹配时，重定向问题就会变得更加复杂。在顽皮狗，动画师定义了一个称为重定向姿势的特殊姿势。该姿势捕捉了源骨架和目标骨架绑定姿势之间的本质差异，从而允许运行时重定向系统调整源姿势，使其在目标角色上更自然地发挥作用。

还有其他更高级的技术，可以将为某个骨架创作的动画重定向到其他骨架上。更多信息，请参见

- animated material parameters (color, specularity, transparency, etc.),
- animated lighting parameters (radius, cone angle, intensity, color, etc.), and
- any other parameters that need to change over time and are in some way synchronized with an animation.

12.4.7 Relationship between Meshes, Skeletons and Clips

The UML diagram in Figure 12.25 shows how animation clip data interfaces with the skeletons, poses, meshes and other data in a game engine. Pay particular attention to the *cardinality* and *direction* of the relationships between these classes. The cardinality is shown just beside the tip or tail of the relationship arrow between classes—a one represents a single instance of the class, while an asterisk indicates many instances. For any one type of character, there will be one skeleton, one or more meshes and one or more animation clips. The skeleton is the central unifying element—the skins are attached to the skeleton but don't have any relationship with the animation clips. Likewise, the clips are targeted at a particular skeleton, but they have no "knowledge" of the skin meshes. Figure 12.26 illustrates these relationships.

Game designers often try to reduce the number of unique skeletons in the game to a minimum, because each new skeleton generally requires a whole new set of animation clips. To provide the illusion of many different types of characters, it is usually better to create multiple meshes skinned to the same skeleton when possible, so that all of the characters can share a single set of animations.

12.4.7.1 Animation Retargeting

We said above that an animation is typically only compatible with a single skeleton. This limitation can be overcome via *animation retargeting* techniques.

Retargeting means using an animation authored for one skeleton to animate a different skeleton. If the two skeletons are morphologically identical, retargeting may boil down to a simple matter of joint index remapping. But when the two skeletons don't match exactly, the retargeting problem becomes more complex. At Naughty Dog, the animators define a special pose known as the *retarget pose*. This pose captures the essential differences between the bind poses of the source and target skeletons, allowing the runtime retargeting system to adjust source poses so they will work more naturally on the target character.

Other more-advanced techniques exist for retargeting animations authored for one skeleton so that they work on a different skeleton. For more infor-

图 12.25.共享动画资源的 UML 图。

信息，请参阅 Ludovic Dutreve 等人撰写的“基于特征点的面部动画重定向”（https://bit.ly/2HL9Cdr）和 Chris Hecker 等人撰写的“实时运动重定向到高度多样化的用户创建形态”（https://bit.ly/2vviG3x）。

图 12.26.许多动画剪辑和一个或多个网格针对单个骨架。

Figure 12.25. UML diagram of shared animation resources.

mation, see “Feature Points Based Facial Animation Retargeting” by Ludovic Dutreve et al. (https://bit.ly/2HL9Cdr) and “Real-time Motion Retargeting to Highly Varied User-Created Morphologies” by Chris Hecker et al. (https://bit.ly/2vviG3x).

Figure 12.26. Many animation clips and one or more meshes target a single skeleton.

12.5 皮肤和矩阵调色板生成

我们已经了解了如何通过旋转、平移以及可能的缩放关节来调整骨架的姿势。

通常将其表示为一组局部 P j → p(j) 或全局 (P j → M) 关节姿态变换，每个关节 j 对应一个。接下来，我们将探索将 3D 网格的顶点附加到已摆好姿势的骨架的过程。此过程称为蒙皮。

12.5.1 每个顶点的蒙皮信息

蒙皮网格通过其顶点附着到骨架上。每个顶点可以绑定到一个或多个关节。如果绑定到单个关节，顶点会精确跟踪该关节的运动。如果绑定到两个或多个关节，顶点的位置将成为其单独绑定到每个关节时所采用位置的加权平均值。

要将网格蒙皮到骨架上，3D 艺术家必须在每个顶点提供以下附加信息：

- **与其绑定的关节的索引，以及**
- **对于每个关节，都有一个加权因子描述该关节对最终顶点位置的影响有多大。**

假设加权因子加起来为一，这是计算任何加权平均值时的惯例。

通常，游戏引擎会对单个顶点可绑定的关节数量设置上限。出于多种原因，通常限制为四个关节。首先，四个 8 位关节索引可以打包成一个 32 位字，这很方便。此外，虽然很容易看出每个顶点有两个、三个甚至四个关节的模型之间的质量差异，但当每个顶点的关节数量超过四个时，大多数人都无法察觉质量差异。

由于关节权重之和必须为 1，因此最后一个权重可以省略，而且通常情况下确实如此。（它可以在运行时计算为 w 3 = 1 − (w 0 + w 1 + w 2) 。）因此，典型的蒙皮顶点数据结构可能如下所示：

```
struct SkinnedVertex
{
    float  m_position[3];    // (Px, Py, Pz)
    float  m_normal[3];      // (Nx, Ny, Nz)
    float  m_u, m_v;         // texture coords (u,v)
    U8     m_jointIndex[4];  // joint indices
    float  m_jointWeight[3]; // joint weights (last
                             // weight omitted)
};
```


12.5 Skinning and Matrix Palette Generation

We've seen how to pose a skeleton by rotating, translating and possibly scaling its joints. And we know that any skeletal pose can be represented mathematically as a set of local $\left(\mathbf{P}_{j\to p(j)}\right)$ or global $\left(\mathbf{P}_{j\to \mathrm{M}}\right)$ joint pose transformations, one for each joint j. Next, we will explore the process of attaching the vertices of a 3D mesh to a posed skeleton. This process is known as *skinning*.

12.5.1 Per-Vertex Skinning Information

A skinned mesh is attached to a skeleton by means of its vertices. Each vertex can be *bound* to one or more joints. If bound to a single joint, the vertex tracks that joint's movement exactly. If bound to two or more joints, the vertex's position becomes a *weighted average* of the positions it would have assumed had it been bound to each joint independently.

To skin a mesh to a skeleton, a 3D artist must supply the following additional information at each vertex:

- the *index* or *indices* of the joint(s) to which it is bound, and
- for each joint, a *weighting factor* describing how much influence that joint should have on the final vertex position.

The weighting factors are assumed to add to one, as is customary when calculating any weighted average.

Usually a game engine imposes an upper limit on the number of joints to which a single vertex can be bound. A four-joint limit is typical for a number of reasons. First, four 8-bit joint indices can be packed into a 32-bit word, which is convenient. Also, while it's pretty easy to see a difference in quality between a two-, three- and even a four-joint-per-vertex model, most people cannot see a quality difference as the number of joints per vertex is increased beyond four.

Because the joint weights must sum to one, the last weight can be omitted and often is. (It can be calculated at runtime as $w_3 = 1 - (w_0 + w_1 + w_2)$.) As such, a typical skinned vertex data structure might look as follows:

```
struct SkinnedVertex
{
    float  m_position[3];    // (Px, Py, Pz)
    float  m_normal[3];      // (Nx, Ny, Nz)
    float  m_u, m_v;         // texture coords (u,v)
    U8     m_jointIndex[4];  // joint indices
    float  m_jointWeight[3]; // joint weights (last
                             // weight omitted)
};
```

12.5.2 剥皮的数学

蒙皮网格的顶点会跟踪其绑定关节的运动。为了在数学上实现这一点，我们需要找到一个矩阵，能够将网格顶点从其原始位置（绑定姿势）变换到与骨架当前姿势相对应的新位置。我们将这样的矩阵称为蒙皮矩阵。

与所有网格顶点一样，蒙皮顶点的位置是在模型空间中指定的。无论其骨架处于绑定姿势还是其他姿势，这一点都是正确的。因此，我们寻求的矩阵将把顶点从模型空间（绑定姿势）变换到模型空间（当前姿势）。与我们迄今为止见过的其他变换（例如模型到世界变换或世界到视图变换）不同，蒙皮矩阵并非改变基础变换。它将顶点变形到新的位置，但顶点在变换前后都位于模型空间中。

12.5.2.1 简单示例：单关节骨架

让我们推导蒙皮矩阵的基本方程。为了简单起见，我们首先使用一个由单个关节组成的骨架。因此，我们需要两个坐标空间：模型空间（我们用下标 M 表示）以及我们唯一关节的关节空间（用下标 J 表示）。关节的坐标轴从绑定姿势开始，我们用上标 B 表示。在动画的任何给定时刻，关节的坐标轴都会移动到模型空间中的新位置和方向——我们用上标 C 表示当前姿势。

现在考虑一个被蒙皮到关节上的顶点。在绑定姿势下，它的模型空间位置是 v BM 。蒙皮过程计算该顶点在当前姿势下新的模型空间位置 v CM 。如图 12.27 所示。

查找给定关节的蒙皮矩阵的“技巧”在于，绑定到关节的顶点的位置在该关节的坐标空间中是恒定的。因此，我们获取模型空间中顶点的绑定姿势位置，将其转□□换到关节空间，将关节移动到其当前姿势，最后将顶点转换回模型空间。这种从模型空间到关节空间再返回的往返操作的最终效果是将顶点从绑定姿势“变形”到当前姿势。

参考图 12.28，假设顶点 v BM 在模型空间中的坐标为 (4, 6)（骨架处于绑定姿势时）。我们将此顶点转换为其等效的关节空间坐标 vj ，大致为 (1, 3)，如图所示。由于顶点处于绑定姿势

12.5.2 The Mathematics of Skinning

The vertices of a skinned mesh track the movements of the joint(s) to which they are bound. To make this happen mathematically, we would like to find a matrix that can transform the vertices of the mesh from their original positions (in bind pose) into new positions that correspond to the current pose of the skeleton. We shall call such a matrix a *skinning matrix*.

Like all mesh vertices, the position of a skinned vertex is specified in model space. This is true whether its skeleton is in bind pose or in any other pose. So the matrix we seek will transform vertices from model space (bind pose) to model space (current pose). Unlike the other transforms we've seen thus far, such as the model-to-world transform or the world-to-view transform, a skinning matrix is *not* a change of basis transform. It morphs vertices into new positions, but the vertices are in model space both before and after the transformation.

12.5.2.1 Simple Example: One-Jointed Skeleton

Let us derive the basic equation for a skinning matrix. To keep things simple at first, we'll work with a skeleton consisting of a single joint. We therefore have two coordinate spaces to work with: model space, which we'll denote with the subscript M, and the joint space of our one and only joint, which will be indicated by the subscript J. The joint's coordinate axes start out in bind pose, which we'll denote with the superscript B. At any given moment during an animation, the joint's axes move to a new position and orientation in model space—we'll indicate this *current pose* with the superscript C.

Now consider a single vertex that is skinned to our joint. In bind pose, its model-space position is $\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}$. The skinning process calculates the vertex's new model-space position in the current pose, $\mathbf{v}_{\mathrm{M}}^{\mathrm{C}}$. This is illustrated in Figure 12.27.

The "trick" to finding the skinning matrix for a given joint is to realize that the position of a vertex bound to a joint is *constant* when expressed in *that joint's coordinate space*. So we take the bind-pose position of the vertex in model space, convert it into joint space, move the joint into its current pose, and finally convert the vertex back into model space. The net effect of this round trip from model space to joint space and back again is to "morph" the vertex from bind pose into the current pose.

Referring to the illustration in Figure 12.28, let's assume that the coordinates of the vertex $\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}$ are (4, 6) in model space (when the skeleton is in bind pose). We convert this vertex into its equivalent joint-space coordinates $\mathbf{v}_j$, which are roughly (1, 3) as shown in the diagram. Because the vertex is bound

图 12.27. 绑定简单的单关节骨架的姿势和当前姿势以及绑定到该关节的单个顶点。

图 12.28. 通过将顶点的位置转换到关节空间，可以使其“跟踪”关节的运动。

对于关节来说，无论关节如何移动，它的关节空间坐标始终为 (1, 3)。一旦关节处于所需的当前姿势，我们就将顶点的坐标转换回模型空间，用符号 v CM 表示。在我们的图中，这些坐标大致为 (18, 2)。因此，蒙皮变换将顶点在模型空间中的坐标从 (4, 6) 变为 (18, 2)，这完全是由于关节从绑定姿势移动到图中所示的当前姿势所致。

从数学角度来看这个问题，我们可以用矩阵 Bj→M 表示关节 j 在模型空间中的绑定姿势。该矩阵将关节 j 空间中坐标表示的点或向量转换为一组等效的模型空间坐标。现在，考虑一个顶点，其坐标在模型空间中表示为绑定姿势的骨架。为了将这些顶点坐标转换到关节 j 的空间，我们只需将其乘以逆

Figure 12.27. Bind pose and current pose of a simple, one-joint skeleton and a single vertex bound to that joint.

Figure 12.28. By transforming a vertex's position into joint space, it can be made to "track" the joint's movements.

to the joint, its joint-space coordinates will *always* be (1, 3) no matter how the joint may move. Once we have the joint in the desired current pose, we convert the vertex's coordinates back into model space, which we'll denote with the symbol $\mathbf{v}_{\mathrm{M}}^{\mathrm{C}}$. In our diagram, these coordinates are roughly (18, 2). So the skinning transformation has morphed our vertex from (4, 6) to (18, 2) in model space, due entirely to the motion of the joint from its bind pose to the current pose shown in the diagram.

Looking at the problem mathematically, we can denote the *bind pose* of the joint j in model space by the matrix $\mathbf{B}_{j\to\mathrm{M}}$. This matrix transforms a point or vector whose coordinates are expressed in joint j's space into an equivalent set of model-space coordinates. Now, consider a vertex whose coordinates are expressed in model space with the skeleton in bind pose. To convert these vertex coordinates into the space of joint j, we simply multiply it by the *inverse*

bind pose matrix, $\mathbf{B}_{\mathrm{M}\to j} = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}$:

$$\mathbf{v}_j = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{B}_{\mathrm{M}\to j} = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}. \tag{12.3}$$

同样，我们可以用矩阵 C j → M 表示关节的当前姿势（即任何非绑定姿势的姿势）。为了将 vj 从关节空间转换回模型空间，我们只需将其乘以当前姿势矩阵即可，如下所示：

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}.$$

如果我们使用公式 (12.3) 展开 vj，我们会得到一个公式，该公式将我们的顶点直接从其在绑定姿势中的位置移到其在当前姿势中的位置：

$$\begin{aligned}\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} &= \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}\\ &= \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}\\ &= \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{K}_j.\end{aligned} \tag{12.4}$$

组合矩阵 K j = (B j → M) − 1 C j → M 称为蒙皮矩阵。

12.5.2.2 扩展到多关节骨架

在上面的例子中，我们只考虑了单个关节。然而，我们上面推导的数学公式实际上适用于任何你能想到的骨架中的任何关节，因为我们用全局姿势（即用关节空间来模拟空间变换）来表示所有内容。为了将上述公式扩展到包含多个关节的骨架，我们只需做两处细微的调整：

1. 我们必须确保使用公式 (12.1) 正确计算出针对相关关节的 B j → M 和 C j → M 矩阵。B j → M 和 C j → M 分别对应于该公式中使用的矩阵 P j → M 的绑定位姿和当前位姿。

2. 我们必须计算蒙皮矩阵 K j 的数组，每个关节 j 对应一个矩阵。
   这个数组被称为 矩阵调色板 。在渲染蒙皮网格时，矩阵调色板会被传递给渲染引擎。对于每个顶点，渲染器会在调色板中查找相应关节的蒙皮矩阵，并使用它将该顶点从绑定姿势转换为当前姿势。

这里需要注意的是，当前姿势矩阵 Cj→M 会随着角色的姿势变化而每帧发生变化。然而，逆绑定姿势矩阵在整个游戏中保持不变，因为骨架的绑定姿势在模型创建时是固定的。因此，

bind pose matrix, $\mathbf{B}_{\mathrm{M}\to j} = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}$:

$$\mathbf{v}_j = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{B}_{\mathrm{M}\to j} = \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}. \tag{12.3}$$

Likewise, we can denote the joint's *current pose* (i.e., any pose that is *not* bind pose) by the matrix $\mathbf{C}_{j\to\mathrm{M}}$. To convert $\mathbf{v}_j$ from joint space back into model space, we simply multiply it by the current pose matrix as follows:

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}.$$

If we expand $\mathbf{v}_j$ using Equation (12.3), we obtain an equation that takes our vertex directly from its position in bind pose to its position in the current pose:

$$\begin{aligned}\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} &= \mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}\\ &= \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}\\ &= \mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{K}_j.\end{aligned} \tag{12.4}$$

The combined matrix $\mathbf{K}_j = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}$ is known as a *skinning matrix*.

12.5.2.2 Extension to Multijointed Skeletons

In the example above, we considered only a single joint. However, the math we derived above actually applies to any joint in any skeleton imaginable, because we formulated everything in terms of global poses (i.e., joint space to model space transforms). To extend the above formulation to a skeleton containing multiple joints, we therefore need to make only two minor adjustments:

1. We must make sure that our $\mathbf{B}_{j\to\mathrm{M}}$ and $\mathbf{C}_{j\to\mathrm{M}}$ matrices are calculated properly for the joint in question, using Equation (12.1). $\mathbf{B}_{j\to\mathrm{M}}$ and $\mathbf{C}_{j\to\mathrm{M}}$ are just the bind pose and current pose equivalents, respectively, of the matrix $\mathbf{P}_{j\to\mathrm{M}}$ used in that equation.

2. We must calculate an array of skinning matrices $\mathbf{K}_j$, one for each joint j. This array is known as a *matrix palette*. The matrix palette is passed to the rendering engine when rendering a skinned mesh. For each vertex, the renderer looks up the appropriate joint's skinning matrix in the palette and uses it to transform the vertex from bind pose into current pose.

We should note here that the current pose matrix $\mathbf{C}_{j\to\mathrm{M}}$ changes every frame as the character assumes different poses over time. However, the inverse bind-pose matrix is constant throughout the entire game, because the bind pose of the skeleton is fixed when the model is created. Therefore, the

矩阵 Bj→M ）－1 通常与骨架一起缓存，不需要计算。
在运行时延迟。动画引擎通常会计算每个
关节 C j → p（j），然后使用公式 (12.1) 将它们转换为全局姿势（C j → M），最后将每个全局姿势乘以相应的缓存逆绑定姿势矩阵（B j → M）－1，以便为每个关节生成一个蒙皮矩阵（K j）。

12.5.2.3 结合模型到世界的变换

每个顶点最终都必须从模型空间变换到世界空间。因此，有些引擎会将蒙皮矩阵的调色板预乘以对象的模型到世界变换。这是一种非常有效的优化，因为它可以在渲染蒙皮几何体时，为每个顶点节省一次矩阵乘法运算。（由于需要处理数十万个顶点，这些节省的运算量累积起来可谓非常可观！）

为了将模型到世界的变换纳入我们的蒙皮矩阵，我们只需将其连接到常规蒙皮矩阵方程，如下所示：

$$\left(\mathbf{K}_j\right)_{\mathrm{W}} = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}\mathbf{M}_{\mathrm{M}\to\mathrm{W}}.$$

有些引擎会像这样将模型到世界的变换烘焙到蒙皮矩阵中，而有些则不会。选择权完全取决于工程团队，并受各种因素影响。例如，我们绝对不想这样做的一种情况是，将单个动画同时应用于多个角色——这种技术称为动画实例化，有时用于为大量角色制作动画。在这种情况下，我们需要将模型到世界的变换分开，以便我们可以在群体中的所有角色之间共享一个矩阵调色板。

12.5.2.4 将顶点蒙皮到多个关节

当一个顶点被蒙皮到多个关节时，我们会假设它被分别蒙皮到每个关节，然后计算每个关节的模型空间位置，最后对结果位置取加权平均值，以此来计算其最终位置。权重由角色绑定美术师提供，且总和必须始终为 1。（如果总和不为 1，则应由工具管线进行重新归一化。）

N 个数量 a 0 到 a N － 1 的加权平均值的一般公式，其中权重为 w 0 到 w N － 1 ，∑wi = 1 为：

$$a = \sum_{i=0}^{N-1} w_i a_i.$$

matrix $\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}$ is generally cached with the skeleton, and needn't be calculated at runtime. Animation engines generally calculate local poses for each joint $\left(\mathbf{C}_{j\to p(j)}\right)$, then use Equation (12.1) to convert these into global poses $\left(\mathbf{C}_{j\to\mathrm{M}}\right)$, and finally multiply each global pose by the corresponding cached inverse bind pose matrix $\left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}$ in order to generate a skinning matrix $\left(\mathbf{K}_j\right)$ for each joint.

12.5.2.3 Incorporating the Model-to-World Transform

Every vertex must eventually be transformed from model space into world space. Some engines therefore premultiply the palette of skinning matrices by the object's model-to-world transform. This can be a useful optimization, as it saves the rendering engine one matrix multiply per vertex when rendering skinned geometry. (With hundreds of thousands of vertices to process, these savings can really add up!)

To incorporate the model-to-world transform into our skinning matrices, we simply concatenate it to the regular skinning matrix equation, as follows:

$$\left(\mathbf{K}_j\right)_{\mathrm{W}} = \left(\mathbf{B}_{j\to\mathrm{M}}\right)^{-1}\mathbf{C}_{j\to\mathrm{M}}\mathbf{M}_{\mathrm{M}\to\mathrm{W}}.$$

Some engines bake the model-to-world transform into the skinning matrices like this, while others don't. The choice is entirely up to the engineering team and is driven by all sorts of factors. For example, one situation in which we would definitely *not* want to do this is when a single animation is being applied to multiple characters simultaneously—a technique known as *animation instancing* that is sometimes used for animating large crowds of characters. In this case we need to keep the model-to-world transforms separate so that we can share a single matrix palette across all characters in the crowd.

12.5.2.4 Skinning a Vertex to Multiple Joints

When a vertex is skinned to more than one joint, we calculate its final position by assuming it is skinned to each joint individually, calculating a model-space position for each joint and then taking a *weighted average* of the resulting positions. The weights are provided by the character rigging artist, and they must always sum to one. (If they do not sum to one, they should be renormalized by the tools pipeline.)

The general formula for a weighted average of N quantities a_0 through a_{N-1}, with weights w_0 through w_{N-1} and with $\sum w_i = 1$ is:

$$a = \sum_{i=0}^{N-1} w_i a_i.$$

这对于矢量 ai 也同样适用。因此，对于一个蒙皮到 N 个关节的顶点，其索引为 j 0 到 j N − 1，权重为 w 0 到 w N − 1，我们可以将公式 (12.4) 扩展如下：

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_{\mathrm{M}}^{\mathrm{B}} \mathbf{K}_{j_i},$$

其中 K ji 是关节 ji 的蒙皮矩阵。

12.6 动画混合

动画混合是指允许多个动画剪辑共同构成角色最终姿势的任何技术。更准确地说，混合是指将两个或多个输入姿势组合起来，从而生成骨架的输出姿势。

混合通常将两个或多个姿势在某个时间点组合起来，并在同一时刻生成输出。在这种情况下，混合用于将两个或多个动画组合成一系列新动画，而无需手动创建它们。例如，通过将受伤行走动画与未受伤行走动画混合，我们可以为角色在行走时生成各种中等程度的明显受伤。再举一个例子，我们可以混合角色向左瞄准的动画和向右瞄准的动画，以使角色沿着两个极端之间的任意角度瞄准。混合可用于在极端面部表情、身体姿势、运动模式等之间进行插值。

混合也可用于在不同时间点找到两个已知姿势之间的中间姿势。当我们想要找到角色在某个时间点的姿势，而该姿势与动画数据中可用的采样帧之一并不完全对应时，就会使用这种方法。我们还可以使用时间动画混合，通过在短时间内逐渐从源动画混合到目标动画，实现从一个动画到另一个动画的平滑过渡。

12.6.1 LERP 混合

并且 P skel B = { (PB) j } | N − 1 j = 0 ，我们希望在这两个极端之间找到一个中间姿态 P skel LERP。这可以通过在两个极端中每个关节的局部姿态之间进行线性插值（LERP）来实现。

This works equally well for vector quantities $\mathbf{a}_i$. So, for a vertex skinned to N joints with indices j_0 through j_{N-1} and weights w_0 through w_{N-1}, we can extend Equation (12.4) as follows:

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_{\mathrm{M}}^{\mathrm{B}} \mathbf{K}_{j_i},$$

where $\mathbf{K}_{j_i}$ is the skinning matrix for the joint j_i.

12.6 Animation Blending

The term *animation blending* refers to any technique that allows more than one animation clip to contribute to the final pose of the character. To be more precise, blending combines two or more *input poses* to produce an *output pose* for the skeleton.

Blending usually combines two or more poses at a single point in time, and generates an output at that same moment in time. In this context, blending is used to combine two or more animations into a host of new animations, without having to create them manually. For example, by blending an injured walk animation with an uninjured walk, we can generate various intermediate levels of apparent injury for our character while he is walking. As another example, we can blend between an animation in which the character is aiming to the left and one in which he's aiming to the right, in order to make the character aim along any desired angle between the two extremes. Blending can be used to interpolate between extreme facial expressions, body stances, locomotion modes and so on.

Blending can also be used to find an intermediate pose between two known poses at *different* points in time. This is used when we want to find the pose of a character at a point in time that does not correspond exactly to one of the sampled frames available in the animation data. We can also use temporal animation blending to smoothly transition from one animation to another, by gradually blending from the source animation to the destination over a short period of time.

12.6.1 LERP Blending

Given a skeleton with N joints, and two skeletal poses $\mathbf{P}_A^{\mathrm{skel}} = \left\{(\mathbf{P}_A)_j\right\}|_{j=0}^{N-1}$ and $\mathbf{P}_B^{\mathrm{skel}} = \left\{(\mathbf{P}_B)_j\right\}|_{j=0}^{N-1}$, we wish to find an intermediate pose $\mathbf{P}_{\mathrm{LERP}}^{\mathrm{skel}}$ between these two extremes. This can be done by performing a *linear interpolation* (LERP) between the local poses of each individual joint in each of the two

源姿势。可以写成如下形式：

$$
\begin{aligned}
(\mathbf{P}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{P}_A)_j, (\mathbf{P}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{P}_A)_j + \beta(\mathbf{P}_B)_j.
\end{aligned} \tag{12.5}
$$

整个骨架的插值姿势只是所有关节的插值姿势的集合：

$$
\mathbf{P}_{\text{LERP}}^{\text{skel}} = \left\{(\mathbf{P}_{\text{LERP}})_j\right\}\Big|_{j=0}^{N-1}. \tag{12.6}
$$

在这些方程中，β 称为混合百分比或混合因子。当 β = 0 时，骨架的最终姿势将与 P skel A 完全匹配；当 β = 1 时，最终姿势将与 P skel B 匹配。当 β 介于 0 和 1 之间时，最终姿势介于两个极端之间。此效果如图 12.11 所示。

我们在这里忽略了一个小细节：我们对关节姿势进行线性插值，这意味着对 4 ×4 变换矩阵进行插值。但是，正如我们在第 5 章中看到的，直接插值矩阵是不切实际的。这就是为什么局部姿势通常以 SRT 格式表示的原因之一——这样做使我们能够将第 5.2.5 节中定义的 LERP 操作分别应用于 SRT 的每个组件。SRT 的平移分量 T 的线性插值只是一个简单的向量 LERP：

$$
\begin{aligned}
(\mathbf{T}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{T}_A)_j, (\mathbf{T}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{T}_A)_j + \beta(\mathbf{T}_B)_j.
\end{aligned} \tag{12.7}
$$

旋转分量的线性插值是四元数LERP或SLERP（球面线性插值）：

$$
\begin{aligned}
(\mathrm{Q}_{\text{LERP}})_j &= \text{normalize}\left(\text{LERP}\left((\mathrm{Q}_A)_j, (\mathrm{Q}_B)_j, \beta\right)\right) \\
&= \text{normalize}\left((1-\beta)(\mathrm{Q}_A)_j + \beta(\mathrm{Q}_B)_j\right).
\end{aligned} \tag{12.8}
$$

or

$$
\begin{aligned}
(\mathrm{Q}_{\text{SLERP}})_j &= \text{SLERP}\left((\mathrm{Q}_A)_j, (\mathrm{Q}_B)_j, \beta\right) \\
&= \frac{\sin\left((1-\beta)\theta\right)}{\sin(\theta)}(\mathrm{Q}_A)_j + \frac{\sin(\beta\theta)}{\sin(\theta)}(\mathrm{Q}_B)_j.
\end{aligned} \tag{12.9}
$$

最后，比例分量的线性插值是标量或矢量 LERP，具体取决于引擎支持的比例类型（均匀或非均匀比例）：

$$
\begin{aligned}
(\mathbf{S}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{S}_A)_j, (\mathbf{S}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{S}_A)_j + \beta(\mathbf{S}_B)_j.
\end{aligned} \tag{12.10}
$$

source poses. This can be written as follows:

$$
\begin{aligned}
(\mathbf{P}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{P}_A)_j, (\mathbf{P}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{P}_A)_j + \beta(\mathbf{P}_B)_j.
\end{aligned} \tag{12.5}
$$

The interpolated pose of the whole skeleton is simply the set of interpolated poses for all of the joints:

$$
\mathbf{P}_{\text{LERP}}^{\text{skel}} = \left\{(\mathbf{P}_{\text{LERP}})_j\right\}\Big|_{j=0}^{N-1}. \tag{12.6}
$$

In these equations, β is called the *blend percentage* or *blend factor*. When $\beta = 0$, the final pose of the skeleton will exactly match $\mathbf{P}_A^{\text{skel}}$; when $\beta = 1$, the final pose will match $\mathbf{P}_B^{\text{skel}}$. When β is between zero and one, the final pose is an intermediate between the two extremes. This effect is illustrated in Figure 12.11.

We've glossed over one small detail here: We are linearly interpolating *joint poses*, which means interpolating 4×4 *transformation matrices*. But, as we saw in Chapter 5, interpolating matrices directly is not practical. This is one of the reasons why local poses are usually expressed in SRT format—doing so allows us to apply the LERP operation defined in Section 5.2.5 to each component of the SRT individually. The linear interpolation of the translation component **T** of an SRT is just a straightforward vector LERP:

$$
\begin{aligned}
(\mathbf{T}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{T}_A)_j, (\mathbf{T}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{T}_A)_j + \beta(\mathbf{T}_B)_j.
\end{aligned} \tag{12.7}
$$

The linear interpolation of the rotation component is a quaternion LERP or SLERP (spherical linear interpolation):

$$
\begin{aligned}
(\mathrm{Q}_{\text{LERP}})_j &= \text{normalize}\left(\text{LERP}\left((\mathrm{Q}_A)_j, (\mathrm{Q}_B)_j, \beta\right)\right) \\
&= \text{normalize}\left((1-\beta)(\mathrm{Q}_A)_j + \beta(\mathrm{Q}_B)_j\right).
\end{aligned} \tag{12.8}
$$

or

$$
\begin{aligned}
(\mathrm{Q}_{\text{SLERP}})_j &= \text{SLERP}\left((\mathrm{Q}_A)_j, (\mathrm{Q}_B)_j, \beta\right) \\
&= \frac{\sin\left((1-\beta)\theta\right)}{\sin(\theta)}(\mathrm{Q}_A)_j + \frac{\sin(\beta\theta)}{\sin(\theta)}(\mathrm{Q}_B)_j.
\end{aligned} \tag{12.9}
$$

Finally, the linear interpolation of the scale component is either a scalar or vector LERP, depending on the type of scale (uniform or nonuniform scale) supported by the engine:

$$
\begin{aligned}
(\mathbf{S}_{\text{LERP}})_j &= \text{LERP}\left((\mathbf{S}_A)_j, (\mathbf{S}_B)_j, \beta\right) \\
&= (1-\beta)(\mathbf{S}_A)_j + \beta(\mathbf{S}_B)_j.
\end{aligned} \tag{12.10}
$$

or

$$(S_{\text{LERP}})_j = \text{LERP}\left((S_A)_j, (S_B)_j, \beta\right) \tag{12.11}$$
$$= (1-\beta)(S_A)_j + \beta(S_B)_j.$$

在两个骨骼姿势之间进行线性插值时，看起来最自然的中间姿势通常是每个关节姿势在其直接父关节的空间中独立于其他姿势进行插值。换句话说，姿势混合通常是在局部姿势上进行的。如果我们直接在模型空间中混合全局姿势，结果往往在生物力学上看起来不合理。

由于姿势混合是在局部姿势上进行的，因此任何一个关节姿势的线性插值都完全独立于骨架中其他关节的插值。这意味着线性姿势插值可以在多处理器架构上完全并行执行。

12.6.2 LERP 混合的应用

现在我们了解了 LERP 混合的基础知识，让我们看一些典型的游戏应用。

12.6.2.1 时间插值

正如我们在 12.4.1.1 节中提到的，游戏动画几乎从来不会精确地在整数帧索引上采样。由于帧速率可变，玩家实际上可能会看到第 0.9、1.85 和 3.02 帧，而不是预期的第 1、2 和 3 帧。此外，一些动画压缩技术涉及仅存储不同的关键帧，这些关键帧在剪辑的本地时间轴上以不均匀的间隔分布。无论哪种情况，我们都需要一种机制来查找动画剪辑中实际存在的采样姿势之间的中间姿势。

LERP 混合通常用于查找这些中间姿势。例如，假设我们的动画剪辑包含时间 0、Δt、2Δt、3Δt 等位置均匀分布的姿势样本。为了找到时间 t = 2.18Δt 处的姿势，我们只需找到时间 2Δt 和 3Δt 处的姿势之间的线性插值，并使用混合百分比 β = 0.18。

一般来说，我们可以通过给定任意两个时间 t 1 和 t 2 处的姿势样本（包含 t ）来找到时间 t 处的姿势，如下所示：

$$\mathbf{P}_j(t) = \text{LERP}\left(\mathbf{P}_j(t_1), \mathbf{P}_j(t_2), \beta(t)\right) \tag{12.12}$$
$$= (1-\beta(t))\,\mathbf{P}_j(t_1) + \beta(t)\mathbf{P}_j(t_2), \tag{12.13}$$

其中混合因子 β (t) 可以通过以下比率确定

$$\beta(t) = \frac{t-t_1}{t_2-t_1}. \tag{12.14}$$

or

$$(S_{\text{LERP}})_j = \text{LERP}\left((S_A)_j, (S_B)_j, \beta\right) \tag{12.11}$$
$$= (1-\beta)(S_A)_j + \beta(S_B)_j.$$

When linearly interpolating between two skeletal poses, the most natural-looking intermediate pose is generally one in which each joint pose is interpolated independently of the others, in the space of that joint's immediate parent. In other words, pose blending is generally performed on *local poses*. If we were to blend global poses directly in model space, the results would tend to look biomechanically implausible.

Because pose blending is done on local poses, the linear interpolation of any one joint's pose is totally independent of the interpolations of the other joints in the skeleton. This means that linear pose interpolation can be performed entirely in parallel on multiprocessor architectures.

12.6.2 Applications of LERP Blending

Now that we understand the basics of LERP blending, let's have a look at some typical gaming applications.

12.6.2.1 Temporal Interpolation

As we mentioned in Section 12.4.1.1, game animations are almost never sampled exactly on integer frame indices. Because of variable frame rate, the player might actually see frames 0.9, 1.85 and 3.02, rather than frames 1, 2 and 3 as one might expect. In addition, some animation compression techniques involve storing only disparate key frames, spaced at uneven intervals across the clip's local timeline. In either case, we need a mechanism for finding intermediate poses between the sampled poses that are actually present in the animation clip.

LERP blending is typically used to find these intermediate poses. As an example, let's imagine that our animation clip contains evenly spaced pose samples at times 0, Δt, $2\Delta t$, $3\Delta t$ and so on. To find a pose at time $t = 2.18\Delta t$, we simply find the linear interpolation between the poses at times $2\Delta t$ and $3\Delta t$, using a blend percentage of $\beta = 0.18$.

In general, we can find the pose at time t given pose samples at any two times t_1 and t_2 that bracket t, as follows:

$$\mathbf{P}_j(t) = \text{LERP}\left(\mathbf{P}_j(t_1), \mathbf{P}_j(t_2), \beta(t)\right) \tag{12.12}$$
$$= (1-\beta(t))\,\mathbf{P}_j(t_1) + \beta(t)\mathbf{P}_j(t_2), \tag{12.13}$$

where the blend factor $\beta(t)$ can be determined by the ratio

$$\beta(t) = \frac{t-t_1}{t_2-t_1}. \tag{12.14}$$

12.6.2.2 运动连续性：交叉淡入淡出

游戏角色的动画制作是通过拼凑大量精细的动画剪辑来实现的。如果动画师足够优秀，角色在每个剪辑中都能以自然且符合物理规律的方式移动。然而，众所周知，在剪辑之间过渡时，要达到同样的质量水平非常困难。我们在游戏动画中看到的绝大多数“卡顿”都发生在角色从一个剪辑过渡到另一个剪辑时。

理想情况下，我们希望角色身体各部位的运动能够完美流畅，即使在过渡过程中也是如此。换句话说，骨骼中每个关节在运动时描绘出的三维路径不应包含任何突然的“跳跃”。我们称之为 C0 连续性；如图 12.29 所示。

不仅路径本身应该连续，其一阶导数（速度）也应该连续。这被称为 C1 连续性（或速度和动量连续性）。随着我们向更高阶的连续性迈进，动画角色运动的感知质量和真实感会得到提升。例如，我们可能希望实现 C2 连续性，其中运动路径（加速度曲线）的二阶导数也是连续的。

严格达到 C1 或更高级别的数学连续性通常难以实现。然而，基于 LERP 的动画混合可以实现相当令人满意的 C0 运动连续性。它通常也能很好地近似 C1 连续性。当以这种方式应用于剪辑之间的过渡时，LERP 混合有时被称为交叉淡入淡出。LERP 混合可能会引入不必要的瑕疵，例如令人讨厌的“脚部滑动”问题，因此必须谨慎使用。

为了在两个动画之间实现淡入淡出，我们将两个片段的时间线重叠一定量，然后将两个片段混合在一起。混合百分比 β 在时间 t 开始处从零开始，这意味着我们只能看到片段 A

图 12.29。左侧的通道函数具有 C0 连续性，而右侧的路径则不具有。

12.6.2.2 Motion Continuity: Cross-Fading

Game characters are animated by piecing together a large number of fine-grained animation clips. If your animators are any good, the character will appear to move in a natural and physically plausible way *within* each individual clip. However, it is notoriously difficult to achieve the same level of quality when transitioning from one clip to the next. The vast majority of the “pops” we see in game animations occur when the character transitions from one clip to the next.

Ideally, we would like the movements of each part of a character’s body to be perfectly smooth, even during transitions. In other words, the three-dimensional paths traced out by each joint in the skeleton as it moves should contain no sudden “jumps.” We call this *C0 continuity*; it is illustrated in Figure 12.29.

Not only should the paths themselves be continuous, but their first derivatives (velocity) should be continuous as well. This is called *C1 continuity* (or continuity of velocity and momentum). The perceived quality and realism of an animated character’s movement improves as we move to higher- and higher-order continuity. For example, we might want to achieve *C2 continuity*, in which the second derivatives of the motion paths (acceleration curves) are also continuous.

Strict mathematical continuity up to C1 or higher is often infeasible to achieve. However, LERP-based animation blending can be applied to achieve a reasonably pleasing form of C0 motion continuity. It usually also does a pretty good job of approximating C1 continuity. When applied to transitions between clips in this manner, LERP blending is sometimes called *cross-fading*. LERP blending can introduce unwanted artifacts, such as the dreaded “sliding feet” problem, so it must be applied judiciously.

To cross-fade between two animations, we overlap the timelines of the two clips by some reasonable amount, and then blend the two clips together. The blend percentage β starts at zero at time t_{start}, meaning that we see only clip A

Figure 12.29. The channel function on the left has C0 continuity, while the path on the right does not.

当交叉淡入淡出开始时。我们逐渐增加 β 值，直到时间 t end 达到 1。此时，只有片段 B 可见，我们可以完全退出片段 A。交叉淡入淡出的时间间隔 (Δ t bl end = t end − t start) 有时被称为混合时间。

交叉淡入淡出的类型

执行交叉混合过渡有两种常见方法：

- 平滑过渡。当 β 从 0 增加到 1 时，片段 A 和 B 同时播放。为了实现平滑过渡，两个片段必须是循环动画，并且它们的时间线必须同步，以便其中一个片段中腿部和手臂的位置与另一个片段中的位置大致匹配。（如果不这样做，交叉淡入淡出通常会看起来非常不自然。）图 12.30 演示了此技术。

- 冻结过渡。片段 A 的本地时钟在片段 B 开始播放时停止。因此，片段 A 中的骨架姿势被冻结，而片段 B 逐渐接管动作。当两个片段互不相关且无法进行时间同步（而进行平滑过渡时必须进行时间同步）时，这种过渡混合效果很好。图 12.31 展示了这种方法。

我们还可以控制混合因子 β 在过渡过程中如何变化。在图 12.30 和图 12.31 中，混合因子随时间线性变化。为了实现更平滑的过渡，我们可以根据时间的三次函数（例如一维贝塞尔曲线）改变 β。当这样的曲线应用于当前正在混合的剪辑时，它被称为缓出曲线；当这样的曲线应用于正在混合的新剪辑时，它被称为缓入曲线。如图 12.32 所示。

图 12.30. 平滑过渡，其中两个剪辑的本地时钟在过渡期间保持运行。

when the cross-fade begins. We gradually increase β until it reaches a value of one at time t_{end}. At this point only clip B will be visible, and we can retire clip A altogether. The time interval over which the cross-fade occurs ($\Delta t_{\text{blend}} = t_{\text{end}} - t_{\text{start}}$) is sometimes called the *blend time*.

Types of Cross-Fades

There are two common ways to perform a cross-blended transition:

- *Smooth transition*. Clips A and B both play simultaneously as β increases from zero to one. For this to work well, the two clips must be looping animations, and their timelines must be synchronized so that the positions of the legs and arms in one clip match up roughly with their positions in the other clip. (If this is not done, the cross-fade will often look totally unnatural.) This technique is illustrated in Figure 12.30.
- *Frozen transition*. The local clock of clip A is stopped at the moment clip B starts playing. Thus, the pose of the skeleton from clip A is frozen while clip B gradually takes over the movement. This kind of transitional blend works well when the two clips are unrelated and cannot be time-synchronized, as they must be when performing a *smooth* transition. This approach is depicted in Figure 12.31.

We can also control how the blend factor β varies during the transition. In Figure 12.30 and Figure 12.31, the blend factor varied linearly with time. To achieve an even smoother transition, we could vary β according to a cubic function of time, such as a one-dimensional Bézier. When such a curve is applied to a currently running clip that is being blended out, it is known as an *ease-out curve*; when it is applied to a new clip that is being blended in, it is known as an *ease-in curve*. This is shown in Figure 12.32.

Figure 12.30. A smooth transition, in which the local clocks of both clips keep running during the transition.

图 12.31. 冻结的过渡，其中剪辑 A 的本地时钟在过渡期间停止。

贝塞尔缓入/缓出曲线的公式如下所示。它返回混合间隔内任意时刻 t 的 β 值。β start 是混合间隔 t start 起始处的混合因子，β end 是时刻 t end 的最终混合因子。参数 u 是 t start 和 t end 之间的归一化时间，为了方便起见，我们还定义 v = 1 − u（归一化时间的倒数）。请注意，贝塞尔切线 T start 和 T end 被认为等于相应的混合因子 β start 和 β end，因为这会生成一条符合我们目的的良好曲线：

$$
\begin{aligned}
\text{let } u &= \left(\frac{t - t_{\text{start}}}{t_{\text{end}} - t_{\text{start}}}\right) \\
\text{and } v &= 1 - u. \\
\beta(t) &= (v^3)\beta_{\text{start}} + (3v^2u)T_{\text{start}} + (3vu^2)T_{\text{end}} + (u^3)\beta_{\text{end}} \\
&= (v^3 + 3v^2u)\beta_{\text{start}} + (3vu^2 + u^3)\beta_{\text{end}}.
\end{aligned}
$$

核心姿势

现在是时候提及运动连续性实际上可以

图 12.32. 平滑过渡，将三次缓入/缓出曲线应用于混合因子。

Figure 12.31. A frozen transition, in which clip A's local clock is stopped during the transition.

The equation for a Bézier ease-in/ease-out curve is given below. It returns the value of β at any time t within the blend interval. β_{start} is the blend factor at the start of the blend interval t_{start}, and β_{end} is the final blend factor at time t_{end}. The parameter u is the *normalized time* between t_{start} and t_{end}, and for convenience we'll also define $v = 1 - u$ (the *inverse* normalized time). Note that the Bézier tangents T_{start} and T_{end} are taken to be equal to the corresponding blend factors β_{start} and β_{end}, because this yields a well-behaved curve for our purposes:

$$
\begin{aligned}
\text{let } u &= \left(\frac{t - t_{\text{start}}}{t_{\text{end}} - t_{\text{start}}}\right) \\
\text{and } v &= 1 - u. \\
\beta(t) &= (v^3)\beta_{\text{start}} + (3v^2u)T_{\text{start}} + (3vu^2)T_{\text{end}} + (u^3)\beta_{\text{end}} \\
&= (v^3 + 3v^2u)\beta_{\text{start}} + (3vu^2 + u^3)\beta_{\text{end}}.
\end{aligned}
$$

Core Poses

This is an appropriate time to mention that motion continuity can actually be

Figure 12.32. A smooth transition, with a cubic ease-in/ease-out curve applied to the blend factor.

图 12.33。在枢轴运动中，角色面向移动方向，并绕垂直轴旋转。在定向运动中，运动方向无需与面向方向一致。

如果动画师确保任何给定剪辑中的最后一个姿势与其后一个剪辑的第一个姿势相匹配，则可以无需混合实现。在实践中，动画师通常会确定一组核心姿势 - 例如，我们可能有一个直立的核心姿势，一个蹲下的核心姿势，一个俯卧的核心姿势等等。通过确保角色在每个剪辑开始时都以这些核心姿势之一开始，并在结束时返回核心姿势，只需确保在动画拼接在一起时核心姿势匹配即可实现 C0 连续性。通过确保角色在一个剪辑结束时的动作平滑过渡到下一个剪辑开始时的动作，还可以实现 C1 或更高阶的运动连续性。这可以通过创作一个平滑的动画然后将其分成两个或多个剪辑来实现。

12.6.2.3 定向运动

基于 LERP 的动画混合通常应用于角色运动。当真人走路或跑步时，他可以通过两种基本方式改变移动的方向：首先，他可以转动整个身体来改变方向，在这种情况下，他总是面朝移动的方向。我将此称为枢轴运动，因为人在转身时会绕垂直轴旋转。其次，他可以在向前、向后或向侧面行走（在游戏世界中称为扫射）时保持面朝一个方向，以便朝着与面朝方向无关的方向移动。我将此称为目标运动，因为它通常用于在移动时保持眼睛或武器瞄准目标。图 12.33 说明了这两种运动方式。

有针对性的运动

为了实现目标运动，动画师创作了三个独立的循环

Figure 12.33. In pivotal movement, the character faces the direction she is moving and pivots about her vertical axis to turn. In targeted movement, the movement direction need not match the facing direction.

achieved *without* blending if the animator ensures that the last pose in any given clip matches the first pose of the clip that follows it. In practice, animators often decide upon a set of *core poses*—for example, we might have a core pose for standing upright, one for crouching, one for lying prone and so on. By making sure that the character starts in one of these core poses at the beginning of every clip and returns to a core pose at the end, C0 continuity can be achieved by simply ensuring that the core poses match when animations are spliced together. C1 or higher-order motion continuity can also be achieved by ensuring that the character's movement at the end of one clip smoothly transitions into the motion at the start of the next clip. This can be achieved by authoring a single smooth animation and then breaking it into two or more clips.

12.6.2.3 Directional Locomotion

LERP-based animation blending is often applied to character locomotion. When a real human being walks or runs, he can change the direction in which he is moving in two basic ways: First, he can turn his entire body to change direction, in which case he always faces in the direction he's moving. I'll call this *pivotal movement*, because the person pivots about his vertical axis when he turns. Second, he can keep facing in one direction while walking forward, backward or sideways (known as *strafing* in the gaming world) in order to move in a direction that is independent of his facing direction. I'll call this *targeted movement*, because it is often used in order to keep one's eye—or one's weapon—trained on a target while moving. These two movement styles are illustrated in Figure 12.33.

Targeted Movement

To implement *targeted movement*, the animator authors three separate looping

动画剪辑——一个向前移动，一个向左扫射，一个向右扫射。我将这些称为定向运动剪辑。三个方向剪辑围绕半圆的圆周排列，向前为 0 度，向左为 90 度，向右为 -90 度。将角色的面朝方向固定在 0 度，我们在半圆上找到所需的移动方向，选择两个相邻的移动动画并通过基于 LERP 的混合将它们混合在一起。混合百分比 β 由移动角度与两个相邻剪辑的角度的接近程度决定。如图 12.34 所示。

请注意，为了实现完整的圆形混合，我们的混合中没有包含向后运动。这是因为侧向扫射和向后奔跑之间的混合通常看起来不太自然。问题在于，当角色向左扫射时，通常会将右脚交叉在左脚前方，以便与纯粹的向前奔跑动画的混合看起来正确。同样，向右扫射通常是左脚交叉在右脚前方。当我们尝试将此类扫射动画直接混合到向后奔跑中时，一条腿会开始穿过另一条腿，这看起来非常尴尬且不自然。有很多方法可以解决这个问题。一种可行的方法是定义两个半球形混合，一个用于向前运动，一个用于向后运动，每个半球形的扫射动画都经过精心设计，以便在与相应的直线奔跑混合时正常工作。当从一个半球移动到另一个半球时，我们可以播放某种明确的过渡动画，以便角色有机会适当地调整其步态和腿部交叉。

图 12.34. 可以通过混合在四个主要方向上移动的循环运动剪辑来实现目标运动。

animation clips—one moving forward, one strafing to the left, and one strafing to the right. I'll call these *directional locomotion clips*. The three directional clips are arranged around the circumference of a semicircle, with forward at 0 degrees, left at 90 degrees and right at −90 degrees. With the character's facing direction fixed at 0 degrees, we find the desired movement direction on the semicircle, select the two adjacent movement animations and blend them together via LERP-based blending. The blend percentage β is determined by how close the angle of movement is to the angles of two adjacent clips. This is illustrated in Figure 12.34.

Note that we did not include backward movement in our blend, for a full circular blend. This is because blending between a sideways strafe and a backward run cannot be made to look natural in general. The problem is that when strafing to the left, the character usually crosses its right foot in front of its left so that the blend into the pure forward run animation looks correct. Likewise, the right strafe is usually authored with the left foot crossing in front of the right. When we try to blend such strafe animations directly into a backward run, one leg will start to pass through the other, which looks extremely awkward and unnatural. There are a number of ways to solve this problem. One feasible approach is to define two hemispherical blends, one for forward motion and one for backward motion, each with strafe animations that have been crafted to work properly when blended with the corresponding straight run. When passing from one hemisphere to the other, we can play some kind of explicit transition animation so that the character has a chance to adjust its gait and leg crossing appropriately.

Figure 12.34. Targeted movement can be implemented by blending together looping locomotion clips that move in each of the four principal directions.

图 12.35.N 个动画剪辑之间的广义线性混合。

枢轴运动

为了实现枢轴运动，我们可以简单地播放向前的运动循环，同时围绕垂直轴旋转整个角色以使其转弯。如果角色的身体在转弯时没有保持直立，枢轴运动看起来会更自然 - 真实人类在转弯时往往会稍微倾斜。我们可以尝试稍微倾斜整个角色的垂直轴，但这会导致内脚陷入地面而外脚离开地面的问题。通过对基本的向前行走或跑步进行三种变化的动画制作可以获得更自然的效果 - 一种是完全笔直，一种是极左转弯，一种是极右转弯。然后，我们可以在直线剪辑和极左转弯剪辑之间进行 LERP 混合以实现任何所需的倾斜角度。

12.6.3 复杂 LERP 混合

在实际的游戏引擎中，角色会出于各种目的使用各种各样的复杂混合。为了方便使用，可以“预打包”某些常用的复杂混合类型。在接下来的章节中，我们将探讨几种常用的预打包复杂混合类型。

12.6.3.1 广义一维 LERP 混合

使用一种我称之为一维 LERP 混合的技术，LERP 混合可以轻松扩展到两个以上的动画剪辑。我们定义一个新的混合参数 b，它位于所需的任何线性范围内（例如，从 − 1 到 + 1，或从 0 到 1，甚至从 27 到 136）。可以将任意数量的剪辑放置在此范围内的任意点，如图 12.35 所示。对于任何给定的 b 值，我们选择与其紧邻的两个剪辑，并使用公式 (12.5) 将它们混合在一起。如果两个相邻的剪辑位于点 b 1 和 b 2 ，则可以使用类似于公式 (12.14) 中使用的技术确定混合百分比 β，如下所示：

$$\beta(t) = \frac{b - b_1}{b_2 - b_1}. \tag{12.15}$$

Figure 12.35. A generalized linear blend between N animation clips.

Pivotal Movement

To implement *pivotal movement*, we can simply play the forward locomotion loop while rotating the entire character about its vertical axis to make it turn. Pivotal movement looks more natural if the character's body doesn't remain bolt upright when it is turning—real humans tend to lean into their turns a little bit. We could try slightly tilting the vertical axis of the character as a whole, but that would cause problems with the inner foot sinking into the ground while the outer foot comes off the ground. A more natural-looking result can be achieved by animating three variations on the basic forward walk or run—one going perfectly straight, one making an extreme left turn and one making an extreme right turn. We can then LERP-blend between the straight clip and the extreme left turn clip to implement any desired lean angle.

12.6.3 Complex LERP Blends

In a real game engine, characters make use of a wide range of complex blends for various purposes. It can be convenient to "prepackage" certain commonly used types of complex blends for ease of use. In the following sections, we'll investigate a few popular types of prepackaged complex blends.

12.6.3.1 Generalized One-Dimensional LERP Blending

LERP blending can be easily extended to more than two animation clips, using a technique I call *one-dimensional LERP blending*. We define a new blend parameter b that lies in any linear range desired (e.g., from -1 to $+1$, or from 0 to 1, or even from 27 to 136). Any number of clips can be positioned at arbitrary points along this range, as shown in Figure 12.35. For any given value of b, we select the two clips immediately adjacent to it and blend them together using Equation (12.5). If the two adjacent clips lie at points b_1 and b_2, then the blend percentage β can be determined using a technique analogous to that used in Equation (12.14), as follows:

$$\beta(t) = \frac{b - b_1}{b_2 - b_1}. \tag{12.15}$$

图 12.36. 目标运动中使用的方向剪辑可以被认为是一维 LERP 混合的特例。

定向运动只是一维 LERP 混合的一个特例。我们只需将放置定向动画剪辑的圆拉直，并使用运动方向角度 θ 作为参数 b（范围为 -90 度到 90 度）。任意数量的动画剪辑都可以以任意角度放置在这个混合范围内。如图 12.36 所示。

12.6.3.2 简单的二维 LERP 混合

有时我们希望同时平滑地改变角色运动的两个方面。例如，我们可能希望角色能够垂直和水平地瞄准武器。或者，我们可能希望角色在移动时改变步幅和双脚间距。为了实现这些效果，我们可以将一维 LERP 混合扩展到二维。

如果我们知道二维混合只涉及四个动画剪辑，并且这些剪辑位于一个正方形区域的四个角，那么我们可以通过执行两次一维混合来找到混合姿势。我们的广义混合因子 b□□ 变成了二维混合向量 b = [bxby] 。如果 b 位于四个剪辑所围成的正方形区域内，我们可以按照以下步骤找到最终姿势：

1. 使用水平混合因子 b□□x ，找到两个中间姿势，一个位于顶部两个动画剪辑之间，另一个位于底部两个动画剪辑之间。这两个姿势可以通过执行两个简单的一阶运算来找到：

Figure 12.36. The directional clips used in targeted movement can be thought of as a special case of one-dimensional LERP blending.

Targeted movement is just a special case of one-dimensional LERP blending. We simply straighten out the circle on which the directional animation clips were placed and use the movement direction angle θ as the parameter b (with a range of -90 to 90 degrees). Any number of animation clips can be placed onto this blend range at arbitrary angles. This is shown in Figure 12.36.

12.6.3.2 Simple Two-Dimensional LERP Blending

Sometimes we would like to smoothly vary *two* aspects of a character's motion simultaneously. For example, we might want the character to be capable of aiming his weapon vertically and horizontally. Or we might want to allow our character to vary her pace length and the separation of her feet as she moves. We can extend one-dimensional LERP blending to two dimensions in order to achieve these kinds of effects.

If we know that our 2D blend involves only four animation clips, and if those clips are positioned at the four corners of a square region, then we can find a blended pose by performing two 1D blends. Our generalized blend factor b becomes a two-dimensional blend vector $\mathbf{b} = \begin{bmatrix} b_x & b_y \end{bmatrix}$. If $\mathbf{b}$ lies within the square region bounded by our four clips, we can find the resulting pose by following these steps:

1. Using the horizontal blend factor b_x, find two intermediate poses, one between the top two animation clips and one between the bottom two clips. These two poses can be found by performing two simple one-

图 12.37.方形区域角落处四个剪辑之间的 2D 动画混合的简单公式。

维度 LERP 混合。

2. 使用垂直混合因子，通过将两个中间姿势 LERP 混合在一起来找到最终姿势。

图 12.37 说明了该技术。

12.6.3.3 三角二维 LERP 混合

上一节中我们讨论的简单 2D 混合技术，仅当需要混合的动画剪辑位于矩形区域的角落时才有效。那么，如何在 2D 混合空间中，对位于任意位置的任意数量的剪辑进行混合呢？

假设我们有三个想要混合在一起的动画剪辑。每个剪辑（由索引 i 指定）对应于二维混合空间中的特定混合坐标 bi = [b ix b iy]；这三个混合坐标在混合空间内形成一个三角形。三个剪辑中的每一个都定义了一组关节姿势 { (P i) j } N − 1 j = 0 ，其中 (P i) j 是剪辑 i 定义的关节 j 的姿势，N 是骨架中的关节数。我们希望找到与三角形内任意点 b 相对应的骨架插值姿势，如图 12.38 所示。

但是我们如何计算三个动画剪辑之间的 LERP 混合？
值得庆幸的是，答案很简单：LERP 函数实际上可以对任意数量的输入进行运算，因为它实际上只是一个加权平均值。与任何加权平均值一样，权重之和必须等于 1。对于双输入 LERP 混合，我们使用权重 β 和 (1 − β)，它们的和当然等于 1。对于三输入 LERP，我们只需使用三个权重，即 α 、 β 和 γ = (1 − α − β)。

Figure 12.37. A simple formulation for 2D animation blending between four clips at the corners of a square region.

dimensional LERP blends.

2. Using the vertical blend factor b_y, find the final pose by LERP-blending the two intermediate poses together.

This technique is illustrated in Figure 12.37.

12.6.3.3 Triangular Two-Dimensional LERP Blending

The simple 2D blending technique we investigated in the previous section only works when the animation clips we wish to blend lie at the corners of a rectangular region. How can we blend between an arbitrary number of clips positioned at arbitrary locations in our 2D blend space?

Let's imagine that we have three animation clips that we wish to blend together. Each clip, designated by the index i, corresponds to a particular blend coordinate $\mathbf{b}_i = \begin{bmatrix} b_{ix} & b_{iy} \end{bmatrix}$ in our two-dimensional blend space; these three blend coordinates form a triangle within the blend space. Each of the three clips defines a set of joint poses $\left\{ (\mathbf{P}_i)_j \right\}\Big|_{j=0}^{N-1}$, where $(\mathbf{P}_i)_j$ is the pose of joint j as defined by clip i, and N is the number of joints in the skeleton. We wish to find the interpolated pose of the skeleton corresponding to an arbitrary point $\mathbf{b}$ within the triangle, as illustrated in Figure 12.38.

But how can we calculate a LERP blend between three animation clips? Thankfully, the answer is simple: the LERP function can actually operate on any number of inputs, because it is really just a *weighted average*. As with any weighted average, the weights must add to one. In the case of a two-input LERP blend, we used the weights β and $(1-\beta)$, which of course add to one. For a three-input LERP, we simply use three weights, α, β and $\gamma = (1-\alpha-\beta)$.

图 12.38. 三个动画剪辑之间的二维动画混合。

然后我们按如下方式计算 LERP：

$$(\text{PLERP})_j = \alpha\,(\text{P}\,0)_j + \beta\,(\text{P}\,1)_j + \gamma\,(\text{P}\,2)_j\,。 \tag{12.16}$$

给定二维混合向量 b ，我们通过在二维混合空间中查找点 b 相对于由三个剪辑形成的三角形的重心坐标 (http://en.wikipedia.org/wiki/Barycentric_coordinates_%28mathematics%29) 来找到混合权重 α 、 β 和 γ 。通常，在顶点为 b 1 、 b 2 和 b 3 的三角形中，点 b 的重心坐标是三个标量值（α 、 β 、 γ），它们满足以下关系

$$\mathbf{b} = \alpha\mathbf{b}_0 + \beta\mathbf{b}_1 + \gamma\mathbf{b}_2, \tag{12.17}$$

and

$$\alpha + \beta + \gamma = 1.$$

这些正是我们为三剪辑加权平均值所寻求的权重。
重心坐标如图 12.39 所示。
请注意，将重心坐标 (1, 0, 0) 代入公式 (12.17) 可得出 b 0 ，而（0, 1, 0）可得出 b 1 ，（0, 0, 1）可得出 b 2 。同样，将这些混合权重代入公式 (12.16) 可分别得出每个关节 j 的姿势（P 0）j 、（P 1）j 和（P 2）j 。此外，重心坐标（1 3 , 1 3 , 1 3）位于三角形的质心，可得出三个姿势之间的均等混合。这正是我们所期望的。

12.6.3.4 广义二维 LERP 混合

重心坐标技术可以扩展到二维混合空间中任意位置的任意数量的动画剪辑。我们不会在这里详细描述它，但其基本思想是使用一种称为 Delaunay 三角剖分的技术 (http://en.wikipedia.org/

Figure 12.38. Two-dimensional animation blending between three animation clips.

Then we calculate the LERP as follows:

$$(\mathbf{P}_{\text{LERP}})_j = \alpha\,(\mathbf{P}_0)_j + \beta\,(\mathbf{P}_1)_j + \gamma\,(\mathbf{P}_2)_j\,. \tag{12.16}$$

Given the two-dimensional blend vector $\mathbf{b}$, we find the blend weights α, β and γ by finding the *barycentric coordinates* of the point $\mathbf{b}$ relative to the triangle formed by the three clips in two-dimensional blend space (http://en.wikipedia.org/wiki/Barycentric_coordinates_%28mathematics%29). In general, the barycentric coordinates of a point $\mathbf{b}$ within a triangle with vertices $\mathbf{b}_1$, $\mathbf{b}_2$ and $\mathbf{b}_3$ are three scalar values (α, β, γ) that satisfy the relations

$$\mathbf{b} = \alpha\mathbf{b}_0 + \beta\mathbf{b}_1 + \gamma\mathbf{b}_2, \tag{12.17}$$

and

$$\alpha + \beta + \gamma = 1.$$

These are exactly the weights we seek for our three-clip weighted average. Barycentric coordinates are illustrated in Figure 12.39.

Note that plugging the barycentric coordinate (1, 0, 0) into Equation (12.17) yields $\mathbf{b}_0$, while (0, 1, 0) gives us $\mathbf{b}_1$ and (0, 0, 1) produces $\mathbf{b}_2$. Likewise, plugging these blend weights into Equation (12.16) gives us poses $(\mathbf{P}_0)_j$, $(\mathbf{P}_1)_j$ and $(\mathbf{P}_2)_j$ for each joint j, respectively. Furthermore, the barycentric coordinate $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ lies at the centroid of the triangle and gives us an *equal* blend between the three poses. This is exactly what we'd expect.

12.6.3.4 Generalized Two-Dimensional LERP Blending

The barycentric coordinate technique can be extended to an arbitrary number of animation clips positioned at arbitrary locations within the two-dimensional blend space. We won't describe it in its entirety here, but the basic idea is to use a technique known as *Delaunay triangulation* (http://en.wikipedia.org/

图 12.39. 三角形内的各种重心坐标。

使用 Delaunay 三角剖分 (wiki/Delaunay_triangulation) 算法，根据各个动画剪辑 bi 的位置，找到一组三角形。确定三角形后，我们可以找到包含所需点 b 的三角形，然后如上所述执行三剪辑 LERP 混合。温哥华的 EA Sports 在《FIFA 足球》中使用了这项技术，并在其专有的“ANT”动画框架中实现。如图 12.40 所示。

12.6.4 部分骨架混合

人类可以独立控制身体的不同部位。例如，我可以一边走路一边挥动右臂，用左臂指向某个物体。在

图 12.40.二维混合空间中任意位置的任意数量动画剪辑之间的 Delaunay 三角剖分。

Figure 12.39. Various barycentric coordinates within a triangle.

wiki/Delaunay_triangulation) to find a set of triangles given the positions of the various animation clips $\mathbf{b}_i$. Once the triangles have been determined, we can find the triangle that encloses the desired point **b** and then perform a three-clip LERP blend as described above. This technique was used in FIFA soccer by EA Sports in Vancouver, implemented within their proprietary “ANT” animation framework. It is shown in Figure 12.40.

12.6.4 Partial-Skeleton Blending

A human being can control different parts of his or her body independently. For example, I can wave my right arm while walking and pointing at something with my left arm. One way to implement this kind of movement in a

Figure 12.40. Delaunay triangulation between an arbitrary number of animation clips positioned at arbitrary locations in two-dimensional blend space.

游戏是通过一种称为部分骨架混合的技术来实现的。

回想一下公式 (12.5) 和 (12.6)，在进行常规 LERP 混合时，骨架中的每个关节都使用相同的混合百分比 β。部分骨架混合扩展了这一思路，允许混合百分比根据每个关节而变化。换句话说，对于每个关节 j ，我们定义一个单独的混合百分比 β j 。整个骨架的所有混合百分比集合 { β j } N − 1 j = 0 有时被称为混合遮罩，因为它可以通过将某些关节的混合百分比设置为零来“屏蔽”它们。

举个例子，假设我们希望角色用他的右臂和右手向某人挥手。此外，我们希望他无论在行走、奔跑还是站立时都能挥手。为了使用部分混合实现这一点，动画师定义了三个全身动画：行走、奔跑和站立。动画师还创建了一个挥手动画，挥手。然后创建一个混合蒙版，其中除右肩、肘、腕和手指关节外，其他所有关节的混合百分比均为零，而这些关节的混合百分比为一：

$$\beta_j = \begin{cases} 1 & \text{when } j \text{ within right arm,} \\ 0 & \text{otherwise.} \end{cases}$$

当使用此混合蒙版将行走、跑步或站立与挥动进行 LERP 混合时，结果就是角色看起来正在行走、跑步或站立，同时挥动右臂。

部分混合很有用，但它往往会使角色的动作看起来不自然。出现这种情况有两个基本原因：

- 关节混合因子的突然变化会导致身体某个部位的运动看起来与身体其他部位的运动脱节。在我们的示例中，混合因子在右肩关节处发生了突然变化。因此，上脊柱、颈部和头部的动画由一个动画驱动，而右肩和手臂关节则完全由另一个动画驱动。这看起来很奇怪。通过逐渐改变混合因子而不是突然改变，可以在一定程度上缓解这个问题。（在我们的示例中，我们可能选择右肩的混合百分比为 0.9，上脊柱为 0.5，颈部和中脊柱为 0.2。）

- 真实人体的运动从来都不是完全独立的。例如，人们会认为一个人在跑步时挥手的动作看起来比站立时更“有弹性”且不受控制。然而，使用部分混合，无论身体其他部位如何运动，右臂的动画都将保持一致。这个问题很难解决

game is via a technique known as *partial-skeleton blending*.

Recall from Equations (12.5) and (12.6) that when doing regular LERP blending, the same blend percentage β was used for every joint in the skeleton. Partial-skeleton blending extends this idea by permitting the blend percentage to vary on a per-joint basis. In other words, for each joint j, we define a separate blend percentage β_j. The set of all blend percentages for the entire skeleton $\{\beta_j\}\big|_{j=0}^{N-1}$ is sometimes called a *blend mask* because it can be used to "mask out" certain joints by setting their blend percentages to zero.

As an example, let's say we want our character to wave at someone using his right arm and hand. Moreover, we want him to be able to wave whether he's walking, running or standing still. To implement this using partial blending, the animator defines three full-body animations: *Walk*, *Run* and *Stand*. The animator also creates a single waving animation, *Wave*. A blend mask is created in which the blend percentages are zero everywhere except for the right shoulder, elbow, wrist and finger joints, where they are equal to one:

$$\beta_j = \begin{cases} 1 & \text{when } j \text{ within right arm,} \\ 0 & \text{otherwise.} \end{cases}$$

When *Walk*, *Run* or *Stand* is LERP-blended with *Wave* using this blend mask, the result is a character who appears to be walking, running or standing while waving his right arm.

Partial blending is useful, but it has a tendency to make a character's movements look unnatural. This occurs for two basic reasons:

- An abrupt change in the per-joint blend factors can cause the movements of one part of the body to appear disconnected from the rest of the body. In our example, the blend factors change abruptly at the right shoulder joint. Hence the animation of the upper spine, neck and head are being driven by one animation, while the right shoulder and arm joints are being entirely driven by a different animation. This can look odd. The problem can be mitigated somewhat by gradually changing the blend factors rather than doing it abruptly. (In our example, we might select a blend percentage of 0.9 at the right shoulder, 0.5 on the upper spine and 0.2 on the neck and mid-spine.)
- The movements of a real human body are never totally independent. For example, one would expect a person's wave to look more "bouncy" and out of control when he or she is running than when he or she is standing still. Yet with partial blending, the right arm's animation will be identical no matter what the rest of the body is doing. This problem is difficult to

部分混合无法解决这个问题。许多游戏开发者转而采用一种看起来更自然的技术，称为加法混合。

12.6.5 添加剂混合

加法混合以一种全新的方式解决了动画组合问题。它引入了一种称为差异剪辑的新动画，顾名思义，它表示两个常规动画剪辑之间的差异。差异剪辑可以添加到常规动画剪辑中，以便在角色的姿势和动作中产生有趣的变化。本质上，差异剪辑编码了将一个姿势转换为另一个姿势所需进行的更改。在游戏行业中，差异剪辑通常被称为加法动画剪辑。在本书中，我们将坚持使用“差异剪辑”一词，因为它更准确地描述了正在发生的事情。

考虑两个输入剪辑，分别称为源剪辑（S）和参考剪辑（R）。
从概念上讲，差异剪辑是 $D = S - R$。如果将差异剪辑 D 添加到其原始参考剪辑中，我们将得到源剪辑（$S = D + R$）。我们也可以通过将一定比例的 D 添加到 R 来生成介于 R 和 S 之间的动画，这与 LERP 混合在两个极端之间找到中间动画的方式非常相似。然而，加法混合技术的真正魅力在于，一旦创建了差异剪辑，它就可以添加到其他不相关的剪辑中，而不仅仅是原始参考剪辑。

我们将这些动画称为目标剪辑，并用符号 T 表示。
举个例子，如果参考剪辑中的角色正常奔跑，而源剪辑中的角色疲惫地奔跑，那么差异剪辑将只包含使角色在奔跑时看起来“疲惫”所需的更改。如果将此差异剪辑应用于角色行走的剪辑，则生成的动画可以使角色在行走时看起来很疲惫。通过将单个差异剪辑添加到多个“常规”动画剪辑上，可以创建大量有趣且非常自然的动画，或者可以创建一系列差异剪辑，每个剪辑添加到单个目标动画时都会产生不同的效果。

12.6.5.1 数学公式

差异动画 D 定义为某个源动画 S 与某个参考动画 R 之间的差异。因此，从概念上讲，差异姿势（在某个时间点）等于 $D = S - R$。当然，我们处理的是关节姿势，而不是标量，所以我们不能简单地将姿势相减。通常，关节姿势是一个 4×4 的仿射变换矩阵，它将

overcome using partial blending. Instead, many game developers have turned to a more natural-looking technique known as *additive blending*.

12.6.5 Additive Blending

Additive blending approaches the problem of combining animations in a totally new way. It introduces a new kind of animation called a *difference clip*, which, as its name implies, represents the difference between two regular animation clips. A difference clip can be added onto a regular animation clip in order to produce interesting variations in the pose and movement of the character. In essence, a difference clip encodes the *changes* that need to be made to one pose in order to transform it into another pose. Difference clips are often called *additive animation clips* in the game industry. We'll stick with the term *difference clip* in this book because it more accurately describes what is going on.

Consider two input clips called the *source clip* (S) and the *reference clip* (R). Conceptually, the difference clip is $D = S - R$. If a difference clip D is added to its original reference clip, we get back the source clip ($S = D + R$). We can also generate animations that are partway between R and S by adding a percentage of D to R, in much the same way that LERP blending finds intermediate animations between two extremes. However, the real beauty of the additive blending technique is that once a difference clip has been created, it can be added to other unrelated clips, not just to the original reference clip. We'll call these animations *target clips* and denote them with the symbol T.

As an example, if the reference clip has the character running normally and the source clip has him running in a tired manner, then the difference clip will contain only the changes necessary to make the character look "tired" while running. If this difference clip is now applied to a clip of the character walking, the resulting animation can make the character look tired while walking. A whole host of interesting and very natural-looking animations can be created by adding a single difference clip onto various "regular" animation clips, or a collection of difference clips can be created, each of which produces a different effect when added to a single target animation.

12.6.5.1 Mathematical Formulation

A difference animation D is defined as the difference between some source animation S and some reference animation R. So conceptually, the difference pose (at a single point in time) is $D = S - R$. Of course, we're dealing with joint poses, not scalar quantities, so we cannot simply subtract the poses. In general, a joint pose is a 4×4 affine transformation matrix that transforms

将子关节局部空间中的点和向量移动到其父关节空间。矩阵减法的等效操作是乘以逆矩阵。因此，给定骨架中任意关节 j 的源姿态 S j 和参考姿态 R j ，我们可以定义该关节的差姿态 D j 如下。（在本讨论中，我们将省略 C → P 或 j → p（j）下标，因为我们处理的是子关节到父关节的姿态矩阵。）

$$\mathbf{D}_j = \mathbf{S}_j \mathbf{R}_j^{-1}.$$

将差异姿态 D j "添加" 到目标姿态 T j 上，得到一个新的附加姿态 A j 。这可以通过简单地将差异变换和目标变换连接起来来实现，如下所示：

$$\mathbf{A}_j = \mathbf{D}_j \mathbf{T}_j = \left(\mathbf{S}_j \mathbf{R}_j^{-1}\right) \mathbf{T}_j. \tag{12.18}$$

我们可以通过观察将差异姿势"添加"回原始参考姿势时发生的情况来验证这是否正确：

$$\begin{aligned}\mathbf{A}_j &= \mathbf{D}_j \mathbf{R}_j \\ &= \mathbf{S}_j \mathbf{R}_j^{-1} \mathbf{R}_j \\ &= \mathbf{S}_j.\end{aligned}$$

换句话说，将差异动画 D 添加回原始参考动画 R 会产生源动画 S，正如我们所期望的那样。

差异片段的时间插值

正如我们在 12.4.1.1 节中了解到的，游戏动画几乎从不在整数帧索引上采样。为了找到任意时刻 t 的姿势，我们通常必须在时刻 t 1 和 t 2 的相邻姿势样本之间进行时间插值。值得庆幸的是，差分片段可以像非加性片段一样进行时间插值。我们可以将公式 (12.12) 和 (12.14) 直接应用于差分片段，就像它们是普通动画一样。

请注意，只有当输入剪辑 S 和 R 的时长相同时，才能找到差异动画。否则，在一段时间内，S 或 R 中任何一个片段都是未定义的，这意味着 D 也同样未定义。

添加剂混合百分比

在游戏中，我们通常希望只混合一定比例的差异动画，以实现不同程度的效果。例如，如果差异动画导致角色将头部向右旋转 80 度，则混合

points and vectors from the child joint's local space to the space of its parent joint. The matrix equivalent of subtraction is multiplication by the inverse matrix. So given the source pose $\mathbf{S}_j$ and the reference pose $\mathbf{R}_j$ for any joint j in the skeleton, we can define the difference pose $\mathbf{D}_j$ at that joint as follows. (For this discussion, we'll drop the C → P or $j \to p(j)$ subscript, as it is understood that we are dealing with child-to-parent pose matrices.)

$$\mathbf{D}_j = \mathbf{S}_j \mathbf{R}_j^{-1}.$$

"Adding" a difference pose $\mathbf{D}_j$ onto a target pose $\mathbf{T}_j$ yields a new additive pose $\mathbf{A}_j$. This is achieved by simply concatenating the difference transform and the target transform as follows:

$$\mathbf{A}_j = \mathbf{D}_j \mathbf{T}_j = \left(\mathbf{S}_j \mathbf{R}_j^{-1}\right) \mathbf{T}_j. \tag{12.18}$$

We can verify that this is correct by looking at what happens when the difference pose is "added" back onto the original reference pose:

$$\begin{aligned}\mathbf{A}_j &= \mathbf{D}_j \mathbf{R}_j \\ &= \mathbf{S}_j \mathbf{R}_j^{-1} \mathbf{R}_j \\ &= \mathbf{S}_j.\end{aligned}$$

In other words, adding the difference animation D back onto the original reference animation R yields the source animation S, as we'd expect.

Temporal Interpolation of Difference Clips

As we learned in Section 12.4.1.1, game animations are almost never sampled on integer frame indices. To find a pose at an arbitrary time t, we must often *temporally interpolate* between adjacent pose samples at times t_1 and t_2. Thankfully, difference clips can be temporally interpolated just like their non-additive counterparts. We can simply apply Equations (12.12) and (12.14) directly to our difference clips as if they were ordinary animations.

Note that a difference animation can only be found when the input clips S and R are of the same duration. Otherwise there would be a period of time during which either S or R is undefined, meaning D would be undefined as well.

Additive Blend Percentage

In games, we often wish to blend in only a percentage of a difference animation to achieve varying degrees of the effect it produces. For example, if a difference clip causes the character to turn his head 80 degrees to the right, blending in

50% 的差异剪辑应该会让他把头向右转 40 度。

为了实现这一点，我们再次求助于我们的老朋友 LERP。我们希望在未改变的目标动画和完全应用差异动画后产生的新动画之间进行插值。为此，我们将公式 (12.18) 扩展如下：

$$\begin{aligned}\mathbf{A}_j &= \text{LERP}\left(\mathbf{T}_j, \mathbf{D}_j\mathbf{T}_j, \beta\right) \\ &= (1-\beta)\left(\mathbf{T}_j\right) + \beta\left(\mathbf{D}_j\mathbf{T}_j\right).\end{aligned} \tag{12.19}$$

正如我们在第五章中看到的，我们不能直接使用 LERP 矩阵。因此，公式 (11.16) 必须分解为针对 S、Q 和 T 的三个独立插值，就像我们在公式 (12.7) 到 (12.11) 中所做的那样。

12.6.5.2 加法混合与部分混合

加法混合在某些方面与部分混合类似。例如，我们可以取站立剪辑和挥动右臂的站立剪辑之间的差异。结果几乎与使用部分混合来制作挥动右臂的动画相同。但是，加法混合较少受到通过部分混合组合的动画的“不连贯”外观的影响。这是因为，使用加法混合，我们不会替换关节子集的动画或在两个可能不相关的姿势之间进行插值。相反，我们会为原始动画添加动作 - 可能覆盖整个骨架。实际上，差异动画“知道”如何改变角色的姿势以使其执行某些特定操作，例如疲倦、将头朝向某个方向或挥动手臂。这些更改可以应用于各种各样的动画，并且结果通常看起来非常自然。

12.6.5.3 添加剂混合的局限性

当然，叠加动画并非灵丹妙药。由于它只是在现有动画上添加动作，因此容易导致骨骼关节过度旋转，尤其是在同时应用多个差异剪辑的情况下。举个简单的例子，假设一个目标动画中，角色的左臂弯曲90度。如果我们添加一个差异动画，同时将肘部也旋转90度，那么最终效果就是手臂旋转90度+90度=180度。这会导致小臂与上臂交叉——这对大多数人来说都不是一个舒适的姿势！

显然，我们在选择参考剪辑以及应用该剪辑的目标剪辑时必须小心谨慎。以下是一些简单的经验法则：

50% of the difference clip should make him turn his head only 40 degrees to the right.

To accomplish this, we turn once again to our old friend LERP. We wish to interpolate between the unaltered target animation and the new animation that would result from a full application of the difference animation. To do this, we extend Equation (12.18) as follows:

$$\begin{aligned}\mathbf{A}_j &= \text{LERP}\left(\mathbf{T}_j, \mathbf{D}_j\mathbf{T}_j, \beta\right) \\ &= (1-\beta)\left(\mathbf{T}_j\right) + \beta\left(\mathbf{D}_j\mathbf{T}_j\right).\end{aligned} \tag{12.19}$$

As we saw in Chapter 5, we cannot LERP matrices directly. So Equation (11.16) must be broken down into three separate interpolations for S, Q and T, just as we did in Equations (12.7) through (12.11).

12.6.5.2 Additive Blending versus Partial Blending

Additive blending is similar in some ways to partial blending. For example, we can take the difference between a standing clip and a clip of standing while waving the right arm. The result will be almost the same as using a partial blend to make the right arm wave. However, additive blends suffer less from the “disconnected” look of animations combined via partial blending. This is because, with an additive blend, we are not replacing the animation for a subset of joints or interpolating between two potentially unrelated poses. Rather, we are adding movement to the original animation—possibly across the entire skeleton. In effect, a difference animation “knows” how to change a character’s pose in order to get him to do something specific, like being tired, aiming his head in a certain direction, or waving his arm. These changes can be applied to a reasonably wide variety of animations, and the result often looks very natural.

12.6.5.3 Limitations of Additive Blending

Of course, additive animation is not a silver bullet. Because it adds movement to an existing animation, it can have a tendency to over-rotate the joints in the skeleton, especially when multiple difference clips are applied simultaneously. As a simple example, imagine a target animation in which the character’s left arm is bent at a 90 degree angle. If we add a difference animation that also rotates the elbow by 90 degrees, then the net effect would be to rotate the arm by $90 + 90 = 180$ degrees. This would cause the lower arm to interpenetrate the upper arm—not a comfortable position for most individuals!

Clearly we must be careful when selecting the reference clip and also when choosing the target clips to which to apply it. Here are some simple rules of thumb:

- 在参考剪辑中尽量减少臀部旋转。
- 在参考剪辑中，肩部和肘关节通常应处于中立姿势，以在将差异剪辑添加到其他目标时尽量减少手臂的过度旋转。
- 动画师应为每个核心姿势（例如，直立、蹲伏、俯卧等）创建一个新的差异动画。这有助于动画师模拟真实人类在每种姿势下的动作方式。

这些经验法则可以作为一个很好的起点，但真正学习如何创建和应用差异动画剪辑的唯一方法是不断尝试，或者跟随有创建和应用差异动画经验的动画师或工程师学习。如果您的团队以前没有使用过加法混合，那么学习加法混合的技巧需要花费大量时间。

12.6.6 添加剂混合的应用

12.6.6.1 站姿变化

加法混合的一个特别引人注目的应用是姿态变化。对于每个所需的姿态，动画师都会创建一个一帧差异动画。当其中一个单帧剪辑与基础动画进行加法混合时，它会导致角色的整体姿态发生剧烈变化，同时他仍然继续执行其应该执行的基本动作。图 12.41 演示了这一概念。

图 12.41。两个单帧差异动画 A 和 B 可能导致目标动画剪辑呈现两种截然不同的姿态。（《神秘海域：德雷克的宝藏》中的角色，© 2007/® SIE。由顽皮狗创作和开发。）

- Keep hip rotations to a minimum in the reference clip.
- The shoulder and elbow joints should usually be in neutral poses in the reference clip to minimize over-rotation of the arms when the difference clip is added to other targets.
- Animators should create a new difference animation for each core pose (e.g., standing upright, crouched down, lying prone, etc.). This allows the animator to account for the way in which a real human would move when in each of these stances.

These rules of thumb can be a helpful starting point, but the only way to really learn how to create and apply difference clips is by trial and error or by apprenticing with animators or engineers who have experience creating and applying difference animations. If your team hasn't used additive blending in the past, expect to spend a significant amount of time learning the art of additive blending.

12.6.6 Applications of Additive Blending

12.6.6.1 Stance Variation

One particularly striking application of additive blending is *stance variation*. For each desired stance, the animator creates a one-frame difference animation. When one of these single-frame clips is additively blended with a base animation, it causes the entire stance of the character to change drastically while he continues to perform the fundamental action he's supposed to perform. This idea is illustrated in Figure 12.41.

Figure 12.41. Two single-frame difference animations A and B can cause a target animation clip to assume two totally different stances. (Character from *Uncharted: Drake's Fortune*, © 2007/® SIE. Created and developed by Naughty Dog.)

图 12.42。加法混合可用于为重复的空闲动画添加变化。图片由顽皮狗公司提供，© 2014/TM SIE
。

12.6.6.2 运动噪音

真实人类的奔跑方式并非每次都完全相同——他们的运动会随着时间推移而变化。尤其是在人分心的情况下（例如，攻击敌人）。加法混合可用于在原本完全重复的运动周期上叠加随机性或对干扰的反应。如图 12.42 所示。

12.6.6.3 瞄准并观察

加法混合的另一个常见用途是允许角色环顾四周或使用武器瞄准。为此，首先要为角色制作一些动作的动画，例如奔跑，头部或武器朝向正前方。然后，动画师将头部的方向或武器的瞄准方向改变到最右端，并保存一个单帧或多帧的差异动画。对最左端、最上端和最下端重复此过程。然后，可以将这四个差异动画加法混合到原始的正前方动画剪辑中，使角色能够瞄准右、左、上、下或介于两者之间的任意方向。

瞄准的角度由每个剪辑的加法混合因子控制。
例如，混合 100% 的右侧添加剂会使角色尽可能向右瞄准。混合 50% 的左侧添加剂会使角色向右瞄准

Figure 12.42. Additive blends can be used to add variation to a repetitive idle animation. Images courtesy of Naughty Dog, Inc., © 2014/™ SIE.

12.6.6.2 Locomotion Noise

Real humans don't run exactly the same way with every footfall—there is variation in their movement over time. This is especially true if the person is distracted (for example, by attacking enemies). Additive blending can be used to layer randomness, or reactions to distractions, on top of an otherwise entirely repetitive locomotion cycle. This is illustrated in Figure 12.42.

12.6.6.3 Aim and Look-At

Another common use for additive blending is to permit the character to look around or to aim his weapon. To accomplish this, the character is first animated doing some action, such as running, with his head or weapon facing straight ahead. Then the animator changes the direction of the head or the aim of the weapon to the extreme right and saves off a one-frame or multi-frame difference animation. This process is repeated for the extreme left, up and down directions. These four difference animations can then be additively blended onto the original straight-ahead animation clip, causing the character to aim right, left, up, down or anywhere in between.

The angle of the aim is governed by the additive blend factor of each clip. For example, blending in 100% of the right additive causes the character to aim as far right as possible. Blending 50% of the left additive causes him to aim at

图 12.43。加法混合可用于武器瞄准。截图由顽皮狗公司提供，© 2014/TM SIE。

这个角度是他最左侧目标的一半。我们也可以将其与向上或向下的附加项结合起来，实现对角线瞄准。如图 12.43 所示。

12.6.6.4 时间轴超载

有趣的是，动画剪辑的时间轴不一定非要用来表示时间。例如，一个三帧动画剪辑可以用来向引擎提供三个瞄准姿势——第 1 帧的左瞄准姿势、第 2 帧的向前瞄准姿势和第 3 帧的右瞄准姿势。为了让角色向右瞄准，我们可以简单地将瞄准动画的本地时钟固定在第 1 帧上。

3. 为了实现向前瞄准和向右瞄准各 50% 的混合，我们可以在第 2.5 帧进行调整。这是一个利用引擎现有功能实现新用途的绝佳示例。

12.7 后处理

一旦骨架由一个或多个动画剪辑设定好姿势，并使用线性插值或加法混合将结果混合在一起，通常需要在渲染角色之前修改姿势。这称为动画后期处理 。在本节中，我们将介绍几种最常见的动画后期处理。

Figure 12.43. Additive blending can be used to aim a weapon. Screenshots courtesy of Naughty Dog, Inc., © 2014/™ SIE.

an angle that is one-half of his leftmost aim. We can also combine this with an up or down additive to aim diagonally. This is demonstrated in Figure 12.43.

12.6.6.4 Overloading the Time Axis

It's interesting to note that the time axis of an animation clip needn't be used to represent time. For example, a three-frame animation clip could be used to provide three aim poses to the engine—a left aim pose on frame 1, a forward aim pose on frame 2 and a right aim pose on frame 3. To make the character aim to the right, we can simply fix the local clock of the aim animation on frame 3. To perform a 50% blend between aiming forward and aiming right, we can dial in frame 2.5. This is a great example of leveraging existing features of the engine for new purposes.

12.7 Post-Processing

Once a skeleton has been posed by one or more animation clips and the results have been blended together using linear interpolation or additive blending, it is often necessary to modify the pose prior to rendering the character. This is called *animation post-processing*. In this section, we'll look at a few of the most common kinds of animation post-processing.

12.7.1 程序动画

程序动画是指任何在运行时生成的动画，而不是由从 Maya 等动画工具导出的数据驱动的动画。有时，最初使用手工动画片段来设定骨架姿势，然后在后期处理步骤中通过程序动画以某种方式修改姿势。程序动画也可以代替手工动画片段作为系统的输入。

例如，假设使用常规动画剪辑使车辆在移动时看起来像是在地形上上下弹跳。车辆的行驶方向由玩家控制。我们希望调整前轮和方向盘的旋转，使它们在车辆转弯时能够逼真地移动。这可以通过对动画生成的姿势进行后处理来实现。假设原始动画的前轮指向正前方，方向盘处于中立位置。我们可以使用当前的转弯角度创建一个绕垂直轴的四元数，该四元数将使前轮偏转所需的量。该四元数可以与前轮关节的 Q 通道相乘，以得出轮胎的最终姿势。同样，我们可以生成一个绕转向柱轴的四元数，并将其乘以方向盘关节的 Q 通道以使其偏转。这些调整是在全局姿势计算和矩阵调色板生成之前对局部姿势进行的（参见第 12.5 节）。

再举一个例子，假设我们希望游戏世界中的树木和灌木丛在风中自然摇曳，并在角色穿过时被拂开。我们可以将树木和灌木丛建模为带有简单骨架的蒙皮网格来实现这一点。可以使用程序动画来代替或补充手动动画剪辑，使关节以自然的方式运动。我们可以对各个关节的旋转应用一个或多个正弦曲线或Perlin噪声函数，使它们在微风中摇曳；当角色穿过包含灌木丛或草地的区域时，我们可以将其根关节四元数径向向外偏转，使其看起来像是被角色推倒了一样。

12.7.2 逆运动学

假设我们有一个动画片段，其中角色俯身从地上捡起一个物体。在 Maya 中，这个片段看起来很棒，但在我们制作的游戏关卡中，地面并非完全平坦，所以有时角色的手会错过物体，或者看起来像是穿过了物体。在这种情况下，我们希望调整骨架的最终姿势，使手与目标物体完全对齐。一种称为逆运动学 (IK) 的技术可以用来

12.7.1 Procedural Animations

A *procedural animation* is any animation generated at runtime rather than being driven by data exported from an animation tool such as Maya. Sometimes, hand-animated clips are used to pose the skeleton initially, and then the pose is modified in some way via procedural animation as a post-processing step. A procedural animation can also be used as an input to the system in place of a hand-animated clip.

For example, imagine that a regular animation clip is used to make a vehicle appear to be bouncing up and down on the terrain as it moves. The direction in which the vehicle travels is under player control. We would like to adjust the rotation of the front wheels and steering wheel so that they move convincingly when the vehicle is turning. This can be done by post-processing the pose generated by the animation. Let's assume that the original animation has the front tires pointing straight ahead and the steering wheel in a neutral position. We can use the current angle of turn to create a quaternion about the vertical axis that will deflect the front tires by the desired amount. This quaternion can be multiplied with the front tire joints' Q channel to produce the final pose of the tires. Likewise, we can generate a quaternion about the axis of the steering column and multiply it into the steering wheel joint's Q channel to deflect it. These adjustments are made to the *local pose*, prior to global pose calculation and matrix palette generation (see Section 12.5).

As another example, let's say that we wish to make the trees and bushes in our game world sway naturally in the wind and get brushed aside when characters move through them. We can do this by modeling the trees and bushes as skinned meshes with simple skeletons. Procedural animation can be used, in place of or in addition to hand-animated clips, to cause the joints to move in a natural-looking way. We might apply one or more sinusoids, or a Perlin noise function, to the rotation of various joints to make them sway in the breeze, and when a character moves through a region containing a bush or grass, we can deflect its root joint quaternion radially outward to make it appear to be pushed over by the character.

12.7.2 Inverse Kinematics

Let's say we have an animation clip in which a character leans over to pick up an object from the ground. In Maya, the clip looks great, but in our production game level, the ground is not perfectly flat, so sometimes the character's hand misses the object or appears to pass through it. In this case, we would like to adjust the final pose of the skeleton so that the hand lines up exactly with the target object. A technique known as *inverse kinematics* (IK) can be used to make

图 12.44 逆运动学试图通过最小化末端执行器关节之间的误差，使它们进入目标全局姿势。

這件事發生了。

常规动画剪辑是正向运动学 (FK) 的一个例子。在正向运动学中，输入是一组局部关节姿势，输出是全局姿势以及每个关节的蒙皮矩阵。逆向运动学则相反：输入是单个关节（称为末端执行器）所需的全局姿势。我们求解骨架中其他关节的局部姿势，使末端执行器到达所需位置。

从数学上讲，反向运动（IK）可以归结为一个误差最小化问题。与大多数最小化问题一样，它可能只有一个解，也可能有多个解，甚至可能没有解。这在直觉上是合理的：如果我试图够到房间另一边的门把手，我必须走到门把手前才能够到。当骨架的初始姿势与目标位置相当接近时，反向运动（IK）效果最佳。这有助□□于算法专注于“最接近”的解，并在合理的处理时间内完成。图 12.44 展示了 IK 的实际应用。

想象一个双关节骨架，每个关节只能绕一个轴旋转。这两个关节的旋转可以用二维角度向量 θ = [θ1 θ2] 来描述。这两个关节所有可能的角度集合构成一个二维空间，称为配置空间。显然，对于更复杂的骨架，每个关节的自由度更高，配置空间就变成了多维的，但无论有多少个维度，这里描述的概念都同样适用。

现在想象一下绘制一个三维图，其中对于每个关节旋转组合（即二维配置空间中的每个点），我们绘制从末端执行器到目标的距离。图 12.45 显示了此类图的一个示例。该三维曲面中的“谷”表示末端执行器尽可能接近目标的区域。当曲面高度为零时，末端执行器已到达目标。逆运动学则试图找到

Figure 12.44. Inverse kinematics attempts to bring an end effector joint into a target global pose by minimizing the error between them.

this happen.

A regular animation clip is an example of *forward kinematics* (FK). In forward kinematics, the input is a set of local joint poses, and the output is a global pose and a skinning matrix for each joint. Inverse kinematics goes in the other direction: The input is the desired global pose of a single joint, which is known as the *end effector*. We solve for the *local* poses of other joints in the skeleton that will bring the end effector to the desired location.

Mathematically, IK boils down to an *error minimization* problem. As with most minimization problems, there might be one solution, many or none at all. This makes intuitive sense: If I try to reach a doorknob that is on the other side of the room, I won't be able to reach it without walking over to it. IK works best when the skeleton starts out in a pose that is reasonably close to the desired target. This helps the algorithm to focus in on the "closest" solution and to do so in a reasonable amount of processing time. Figure 12.44 shows IK in action.

Imagine a two-joint skeleton, each of which can rotate only about a single axis. The rotation of these two joints can be described by a two-dimensional angle vector $\theta = \begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}$. The set of all possible angles for our two joints forms a two-dimensional space called *configuration space*. Obviously, for more complex skeletons with more degrees of freedom per joint, configuration space becomes multidimensional, but the concepts described here work equally well no matter how many dimensions we have.

Now imagine plotting a three-dimensional graph, where for each combination of joint rotations (i.e., for each point in our two-dimensional configuration space), we plot the distance from the end effector to the desired target. An example of this kind of plot is shown in Figure 12.45. The "valleys" in this three-dimensional surface represent regions in which the end effector is as close as possible to the target. When the height of the surface is zero, the end effector has reached its target. Inverse kinematics, then, attempts to find

图 12.45。二维配置空间中每个点从末端执行器到目标的距离的三维图。反向运动找到局部最小值。

该表面上的最小值（低点）。

我们不会在这里深入讨论如何解决 IK 最小化问题。你可以在 http://en.wikipedia.org/wiki/Inverse_kinematics 以及 Jason Weber 的文章《约束逆运动学》[47] 中了解更多关于 IK 的知识。

12.7.3 布娃娃

当角色死亡或失去意识时，他的身体会变得无力。在这种情况下，我们希望身体能够以物理上逼真的方式与周围环境做出反应。为此，我们可以使用布娃娃。布娃娃是物理模拟的刚体的集合，每个刚体代表角色身体的半刚性部分，例如他的小臂或大腿。刚体在角色的关节处相互约束，以产生看起来自然的“无生命”身体运动。刚体的位置和方向由物理系统确定，然后用于驱动角色骨架中某些关键关节的位置和方向。数据从物理系统到骨架的传输通常在后处理步骤中完成。

要真正理解布娃娃的物理原理，我们必须首先了解碰撞和物理系统的工作原理。第 13.4.8.7 节和第 13.5.3.8 节将更详细地介绍布娃娃的物理原理。

12.8 压缩技术

动画数据会占用大量内存。单个关节姿势可能由十个浮点通道组成（三个用于平移，四个用于旋转，四个用于

Figure 12.45. A three-dimensional plot of the distance from the end effector to the target for each point in two-dimensional configuration space. IK finds the local minimum.

minima (low points) on this surface.

We won't get into the details of solving the IK minimization problem here. You can read more about IK at http://en.wikipedia.org/wiki/Inverse_kinematics and in Jason Weber's article, "Constrained Inverse Kinematics" [47].

12.7.3 Rag Dolls

A character's body goes limp when he dies or becomes unconscious. In such situations, we want the body to react in a physically realistic way with its surroundings. To do this, we can use a *rag doll*. A rag doll is a collection of physically simulated rigid bodies, each one representing a semi-rigid part of the character's body, such as his lower arm or his upper leg. The rigid bodies are constrained to one another at the joints of the character in such a way as to produce natural-looking "lifeless" body movement. The positions and orientations of the rigid bodies are determined by the physics system and are then used to drive the positions and orientations of certain key joints in the character's skeleton. The transfer of data from the physics system to the skeleton is typically done as a post-processing step.

To really understand rag doll physics, we must first have an understanding of how the collision and physics systems work. Rag dolls are covered in more detail in Sections 13.4.8.7 and 13.5.3.8.

12.8 Compression Techniques

Animation data can take up a lot of memory. A single joint pose might be composed of ten floating-point channels (three for translation, four for rotation and

（为了便于扩展，最多可以增加三个）。假设每个通道包含一个 4 字节的浮点值，则以每秒 30 个样本的速度采样的一秒钟剪辑将占用 4 字节 × 10 个通道 × 30 个样本/秒 = 每秒每个关节 1200 字节，或者每秒每个关节约 1.17 KiB 的数据速率。对于 100 个关节的骨架（按今天的标准来说很小），未压缩的动画剪辑每秒每个关节将占用 117 KiB。如果我们的游戏包含 1,000 秒的动画（对于现代游戏来说，这偏低），则整个数据集将占用惊人的 114.4 MiB。考虑到 PlayStation 3 只有 256 MiB 的主 RAM 和 256 MiB 的视频 RAM，这个数字相当大。当然，PS4 有 8 GiB 的 RAM。但即便如此，我们宁愿拥有更丰富、更多样的动画，也不愿不必要地浪费内存。因此，游戏工程师投入了大量的精力来压缩动画数据，以便以最小的内存成本实现最大程度的丰富性和多样性的动作。

12.8.1 渠道省略

减少动画剪辑大小的一个简单方法是省略不相关的通道。许多角色不需要非均匀缩放，因此三个缩放通道可以简化为一个均匀缩放通道。在某些游戏中，实际上可以省略所有关节的缩放通道（面部关节可能除外）。人形角色的骨骼通常无法拉伸，因此通常可以省略除根、面部关节以及有时锁骨之外的所有关节的平移。最后，因为四元数总是标准化的，所以我们可以每个四元数只存储三个分量（例如 x 、 y 和 z ），并在运行时重建第四个分量（例如 w ）。

作为进一步的优化，可以将在整个动画过程中姿势不会发生变化的通道存储为时间 t = 0 时的单个样本，再加上一个位，该位指示该通道对于 t 的所有其他值都是恒定的。

减少通道可以显著减小动画剪辑的大小。一个拥有 100 个关节、无缩放和平移的角色仅需要 303 个通道——每个关节的四元数需要 3 个通道，加上根关节平移的 3 个通道。相比之下，如果所有 100 个关节都包含 10 个通道，则需要 1,000 个通道。

12.8.2 量化

减小动画大小的另一种方法是减小每个通道的大小。浮点值通常以 32 位 IEEE 格式存储。此格式提供 23 位尾数精度和 8 位指数精度。

up to three more for scale). Assuming each channel contains a 4-byte floating-point value, a one-second clip sampled at 30 samples per second would occupy 4 bytes × 10 channels × 30 samples/second = 1200 bytes per joint per second, or a data rate of about 1.17 KiB per joint per second. For a 100-joint skeleton (which is small by today's standards), an uncompressed animation clip would occupy 117 KiB per joint per second. If our game contained 1,000 seconds of animation (which is on the low side for a modern game), the entire dataset would occupy a whopping 114.4 MiB. That's quite a lot, considering that a PlayStation 3 has only 256 MiB of main RAM and 256 MiB of video RAM. Sure, the PS4 has 8 GiB of RAM. But even so—we would rather have much richer animations with a lot more variety than waste memory unnecessarily. Therefore, game engineers invest a significant amount of effort into compressing animation data in order to permit the maximum richness and variety of movement at the minimum memory cost.

12.8.1 Channel Omission

One simple way to reduce the size of an animation clip is to omit channels that are irrelevant. Many characters do not require nonuniform scaling, so the three scale channels can be reduced to a single uniform scale channel. In some games, the scale channel can actually be omitted altogether for all joints (except possibly the joints in the face). The bones of a humanoid character generally cannot stretch, so translation can often be omitted for all joints except the root, the facial joints and sometimes the collar bones. Finally, because quaternions are always normalized, we can store only three components per quat (e.g., x, y and z) and reconstruct the fourth component (e.g., w) at runtime.

As a further optimization, channels whose pose does not change over the course of the entire animation can be stored as a single sample at time $t = 0$ plus a single bit indicating that the channel is constant for all other values of t.

Channel omission can significantly reduce the size of an animation clip. A 100-joint character with no scale and no translation requires only 303 channels—three channels for the quaternions at each joint, plus three channels for the root joint's translation. Compare this to the 1,000 channels that would be required if all ten channels were included for all 100 joints.

12.8.2 Quantization

Another way to reduce the size of an animation is to reduce the size of each channel. A floating-point value is normally stored in 32-bit IEEE format. This format provides 23 bits of precision in the mantissa and an 8-bit exponent.

然而，在动画剪辑中通常不需要保留这种精度和范围。存储四元数时，通道值保证在 [− 1, 1] 范围内。当幅度为 1 时，32 位 IEEE 浮点数的指数为零，而 23 位精度可以精确到小数点后七位。经验表明，仅使用 16 位精度就可以很好地编码四元数，因此，如果我们使用 32 位浮点数存储四元数，实际上会浪费每个通道 16 位。

将 32 位 IEEE 浮点数转换为 n 位整数表示的过程称为量化。此操作实际上包含两个部分：编码是将原始浮点值转换为量化整数□□表示的过程。解码是从量化整数恢复原始浮点值的近似值的过程。（我们只能恢复原始数据的近似值——量化是一种有损压缩方法，因为它会有效地减少用于表示值的精度位数。）

要将浮点值编码为整数，我们首先将可能输入值的有效范围划分为 N 个大小相等的区间。然后，我们确定特定浮点值位于哪个区间内，并用其区间的整数索引表示该值。要解码这个量化值，我们只需将整数索引转换为浮点格式，然后将其平移并缩放回原始范围。通常选择 N 来对应于可以用 n 位整数表示的可能整数值的范围。例如，如果我们将 32 位浮点值编码为 16 位整数，则区间数将为 N = 2 16 = 65,536。

Jonathan Blow 在《游戏开发者杂志》的“内积”专栏中撰写了一篇关于浮点标量量化的精彩文章，网址为 https://bit.ly/2J92oiU。文章介绍了在编码过程中将浮点值映射到区间的两种方法：我们可以将浮点数截断到下一个最低区间边界（T 编码），也可以将浮点数四舍五入到封闭区间的中心（R 编码）。同样，它描述了两种从整数表示重建浮点值的方法：我们可以返回原始值映射到的区间左侧的值（L 重建），也可以返回区间中心的值（C 重建）。这给了我们四种可能的编码/解码方法：TL、TC、RL 和 RC。其中，应避免使用 TL 和 RC，因为它们往往会从数据集中删除或添加能量，这通常会带来灾难性的后果。TC 的优势在于它在带宽方面是最高效的，但它也存在一个严重的问题——无法精确表示零值。（如果你编码 0.0f，解码后它会变成一个小的正值。）因此，RL 通常

However, it's often not necessary to retain that kind of precision and range in an animation clip. When storing a quaternion, the channel values are guaranteed to lie in the range $[-1, 1]$. At a magnitude of 1, the exponent of a 32-bit IEEE float is zero, and 23 bits of precision give us accuracy down to the seventh decimal place. Experience shows that a quaternion can be encoded well with only 16 bits of precision, so we're really wasting 16 bits per channel if we store our quats using 32-bit floats.

Converting a 32-bit IEEE float into an n-bit integer representation is called *quantization*. There are actually two components to this operation: *Encoding* is the process of converting the original floating-point value to a quantized integer representation. *Decoding* is the process of recovering an approximation to the original floating-point value from the quantized integer. (We can only recover an *approximation* to the original data—quantization is a *lossy* compression method because it effectively reduces the number of bits of precision used to represent the value.)

To encode a floating-point value as an integer, we first divide the valid range of possible input values into N equally sized *intervals*. We then determine within which interval a particular floating-point value lies and represent that value by the *integer index* of its interval. To decode this quantized value, we simply convert the integer index into floating-point format and shift and scale it back into the original range. N is usually chosen to correspond to the range of possible integer values that can be represented by an n-bit integer. For example, if we're encoding a 32-bit floating-point value as a 16-bit integer, the number of intervals would be $N = 2^{16} = 65{,}536$.

Jonathan Blow wrote an excellent article on the topic of floating-point scalar quantization in the *Inner Product* column of Game Developer Magazine, available at https://bit.ly/2J92oiU. The article presents two ways to map a floating-point value to an interval during the encoding process: We can either *truncate* the float to the next lowest interval boundary (*T encoding*), or we can *round* the float to the center of the enclosing interval (*R encoding*). Likewise, it describes two approaches to reconstructing the floating-point value from its integer representation: We can either return the value of the *left-hand side* of the interval to which our original value was mapped (*L reconstruction*), or we can return the value of the center of the interval (*C reconstruction*). This gives us four possible encode/decode methods: TL, TC, RL and RC. Of these, TL and RC are to be avoided because they tend to remove or add energy to the dataset, which can often have disastrous effects. TC has the benefit of being the most efficient method in terms of bandwidth, but it suffers from a severe problem—there is no way to represent the value zero exactly. (If you encode 0.0f, it becomes a small positive value when decoded.) RL is therefore usually

最好的选择，也是我们将在这里演示的方法。

本文仅讨论了正浮点值的量化，并且在示例中，为了简单起见，假设输入范围为 [0, 1]。然而，我们始终可以将任何浮点范围平移和缩放到 [0, 1] 范围内。例如，四元数通道的范围是 [− 1, 1]，但我们可以通过加 1 然后除以 2 将其转换为 [0, 1] 范围。

以下两个例程根据 Jonathan Blow 的 RL 方法，将 [0, 1] 范围内的输入浮点值编码和解码为 n 位整数。量化值始终返回 32 位无符号整数（U32），但实际仅使用最低有效 n 位，如 nBits 参数所指定。例如，如果您传递 nBits==16 ，则可以安全地将结果转换为 U16 。

```
U32 CompressUnitFloatRL(F32 unitFloat, U32 nBits)
{
    // Determine the number of intervals based on the
    // number of output bits we've been asked to produce.
    U32 nIntervals = 1u << nBits;

    // Scale the input value from the range [0, 1] into
    // the range [0, nIntervals - 1]. We subtract one
    // interval because we want the largest output value
    // to fit into nBits bits.
    F32 scaled = unitFloat * (F32)(nIntervals - 1u);

    // Finally, round to the nearest interval center. We
    // do this by adding 0.5f and then truncating to the
    // next-lowest interval index (by casting to U32).
    U32 rounded = (U32)(scaled + 0.5f);

    // Guard against invalid input values.
    if (rounded > nIntervals - 1u)
    rounded = nIntervals - 1u;

    return rounded;
}

F32 DecompressUnitFloatRL(U32 quantized, U32 nBits)
{
    // Determine the number of intervals based on the
    // number of bits we used when we encoded the value.
    U32 nIntervals = 1u << nBits;

    // Decode by simply converting the U32 to an F32, and
    // scaling by the interval size.
```


the best choice and is the method we'll demonstrate here.

The article only talks about quantizing positive floating-point values, and in the examples, the input range is assumed to be $[0, 1]$ for simplicity. However, we can always shift and scale any floating-point range into the range $[0, 1]$. For example, the range of quaternion channels is $[-1, 1]$, but we can convert this to the range $[0, 1]$ by adding one and then dividing by two.

The following pair of routines encode and decode an input floating-point value lying in the range $[0, 1]$ into an n-bit integer, according to Jonathan Blow's RL method. The quantized value is always returned as a 32-bit unsigned integer (`U32`), but only the least-significant n bits are actually used, as specified by the `nBits` argument. For example, if you pass `nBits==16`, you can safely cast the result to a `U16`.

```
U32 CompressUnitFloatRL(F32 unitFloat, U32 nBits)
{
    // Determine the number of intervals based on the
    // number of output bits we've been asked to produce.
    U32 nIntervals = 1u << nBits;

    // Scale the input value from the range [0, 1] into
    // the range [0, nIntervals - 1]. We subtract one
    // interval because we want the largest output value
    // to fit into nBits bits.
    F32 scaled = unitFloat * (F32)(nIntervals - 1u);

    // Finally, round to the nearest interval center. We
    // do this by adding 0.5f and then truncating to the
    // next-lowest interval index (by casting to U32).
    U32 rounded = (U32)(scaled + 0.5f);

    // Guard against invalid input values.
    if (rounded > nIntervals - 1u)
    rounded = nIntervals - 1u;

    return rounded;
}

F32 DecompressUnitFloatRL(U32 quantized, U32 nBits)
{
    // Determine the number of intervals based on the
    // number of bits we used when we encoded the value.
    U32 nIntervals = 1u << nBits;

    // Decode by simply converting the U32 to an F32, and
    // scaling by the interval size.
```

```
    F32 intervalSize = 1.0f / (F32)(nIntervals - 1u);
    F32 approxUnitFloat = (F32)quantized * intervalSize;

    return approxUnitFloat;
}
```

为了处理 [min , max] 范围内的任意输入值，我们可以使用以下例程：

```
U32 CompressFloatRL(F32 value, F32 min, F32 max,
                    U32 nBits)
{
    F32 unitFloat = (value - min) / (max - min);
    U32 quantized = CompressUnitFloatRL(unitFloat,
                                        nBits);
    return quantized;
}

F32 DecompressFloatRL(U32 quantized, F32 min, F32 max,
                      U32 nBits)
{
    F32 unitFloat = DecompressUnitFloatRL(quantized,
                                          nBits);
    F32 value = min + (unitFloat * (max - min));
    return value;
}
```

让我们回到最初的动画通道压缩问题。为了将四元数的四个分量压缩和解压缩为每个通道 16 位，我们只需调用 CompressFloatRL() 和 DecompressFloatRL() 函数，其中 min = − 1 ， max = 1 ， n = 16 ：

```
inline U16 CompressRotationChannel(F32 qx)
{
    return (U16)CompressFloatRL(qx, -1.0f, 1.0f, 16u);
}

inline F32 DecompressRotationChannel(U16 qx)
{
    return DecompressFloatRL((U32)qx, -1.0f, 1.0f, 16u);
}
```

平移通道的压缩比旋转通道稍微棘手一些，因为与四元数通道不同，平移通道的范围理论上可以是无限的。值得庆幸的是，角色的关节在实际中移动幅度并不大，所以我们可以确定一个合理的运动范围，并标记一个


```
    F32 intervalSize = 1.0f / (F32)(nIntervals - 1u);
    F32 approxUnitFloat = (F32)quantized * intervalSize;

    return approxUnitFloat;
}
```

To handle arbitrary input values in the range $[min, max]$, we can use these routines:

```
U32 CompressFloatRL(F32 value, F32 min, F32 max,
                    U32 nBits)
{
    F32 unitFloat = (value - min) / (max - min);
    U32 quantized = CompressUnitFloatRL(unitFloat,
                                        nBits);
    return quantized;
}

F32 DecompressFloatRL(U32 quantized, F32 min, F32 max,
                      U32 nBits)
{
    F32 unitFloat = DecompressUnitFloatRL(quantized,
                                          nBits);
    F32 value = min + (unitFloat * (max - min));
    return value;
}
```

Let's return to our original problem of animation channel compression. To compress and decompress a quaternion's four components into 16 bits per channel, we simply call `CompressFloatRL()` and `DecompressFloatRL()` with $min = -1$, $max = 1$ and $n = 16$:

```
inline U16 CompressRotationChannel(F32 qx)
{
    return (U16)CompressFloatRL(qx, -1.0f, 1.0f, 16u);
}

inline F32 DecompressRotationChannel(U16 qx)
{
    return DecompressFloatRL((U32)qx, -1.0f, 1.0f, 16u);
}
```

Compression of translation channels is a bit trickier than rotations, because unlike quaternion channels, the range of a translation channel could theoretically be unbounded. Thankfully, the joints of a character don't move very far in practice, so we can decide upon a reasonable range of motion and flag an

如果我们发现动画包含有效范围之外的平移，就会发生错误。游戏内过场动画是此规则的一个例外——当 IGC 在世界空间中动画化时，角色根关节的平移可能会变得非常大。为了解决这个问题，我们可以根据每个剪辑中实际实现的最大平移量，按动画或按关节选择有效平移的范围。由于数据范围可能因动画或关节而异，我们必须将该范围与压缩的剪辑数据一起存储。这会给每个动画剪辑添加少量数据，但影响通常可以忽略不计。

```
// We'll use a 2 m range -- your mileage may vary.
F32 MAX_TRANSLATION = 2.0f;

inline U16 CompressTranslationChannel(F32 vx)
{
    // Clamp to valid range...
    if (vx < -MAX_TRANSLATION)
        vx = -MAX_TRANSLATION;
    if (vx > MAX_TRANSLATION)
        vx = MAX_TRANSLATION;

    return (U16)CompressFloatRL(vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}

inline F32 DecompressTranslationChannel(U16 vx)
{
    return DecompressFloatRL((U32)vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}
```

12.8.3 采样频率和关键遗漏

动画数据往往很大，原因有三：首先，每个关节的姿势可以包含十个以上的浮点数据通道；其次，一个骨架包含大量关节（PS3 或 Xbox 360 上的人形角色有 250 个或更多关节，而某些 PS4 和 Xbox One 游戏中则超过 800 个关节）；第三，角色的姿势通常以高速率采样（例如每秒 30 帧）。我们已经找到了一些解决第一个问题的方法。我们无法真正减少高分辨率角色的关节数量，所以我们只能解决第二个问题。为了解决第三个问题，我们可以做两件事：

- 降低整体采样率。导出时某些动画看起来不错

error if we ever see an animation that contains translations outside the valid range. In-game cinematics are an exception to this rule—when an IGC is animated in world space, the translations of the characters' root joints can grow very large. To address this, we can select the range of valid translations on a per-animation or per-joint basis, depending on the maximum translations actually achieved within each clip. Because the data range might differ from animation to animation, or from joint to joint, we must store the range with the compressed clip data. This will add a tiny amount of data to each animation clip, but the impact is generally negligible.

```
// We'll use a 2 m range -- your mileage may vary.
F32 MAX_TRANSLATION = 2.0f;

inline U16 CompressTranslationChannel(F32 vx)
{
    // Clamp to valid range...
    if (vx < -MAX_TRANSLATION)
        vx = -MAX_TRANSLATION;
    if (vx > MAX_TRANSLATION)
        vx = MAX_TRANSLATION;

    return (U16)CompressFloatRL(vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}

inline F32 DecompressTranslationChannel(U16 vx)
{
    return DecompressFloatRL((U32)vx,
                -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}
```

12.8.3 Sampling Frequency and Key Omission

Animation data tends to be large for three reasons: first, because the pose of each joint can contain upwards of ten channels of floating-point data; second, because a skeleton contains a large number of joints (250 or more for a humanoid character on PS3 or Xbox 360, and more than 800 on some PS4 and Xbox One games); third, because the pose of the character is typically sampled at a high rate (e.g., 30 frames per second). We've seen some ways to address the first problem. We can't really reduce the number of joints for our high-resolution characters, so we're stuck with the second problem. To attack the third problem, we can do two things:

- *Reduce the sample rate overall.* Some animations look fine when exported

以每秒 15 个样本的速度进行，这样做可以将动画数据大小减少一半。

- 省略部分样本。如果某个通道的数据在片段的某个时间间隔内以近似线性的方式变化，我们可以省略该时间间隔内除端点之外的所有样本。然后，在运行时，我们可以使用线性插值来恢复丢失的样本。

后一种技术有点复杂，它需要我们存储每个样本的时间信息。这些额外的数据可能会抵消我们最初通过省略样本所节省的时间。不过，一些游戏引擎已经成功地运用了这种技术。

12.8.4 基于曲线的压缩

我使用过的最强大、最易用、最周全的动画 API 之一是 Rad Game Tools 开发的 Granny。Granny 并非将动画存储为规则间隔的姿势样本序列，而是存储为一系列 n 阶非均匀非有理 B 样条曲线，用于描述关节的 S、Q 和 T 通道随时间变化的路径。使用 B 样条曲线可以仅使用少量数据点对曲率较大的通道进行编码。

Granny 通过定期采样关节姿势来导出动画，这与传统的动画数据非常相似。然后，Granny 会为每个通道使用一组 B 样条曲线拟合采样数据集，使其符合用户指定的容差范围。最终得到的动画剪辑通常比均匀采样、线性插值得到的动画剪辑要小得多。该过程如图 12.46 所示。

图 12.46. 动画压缩的一种形式是将 B 样条拟合到动画通道数据。

12.8.5 小波压缩

压缩动画数据的另一种方法是将信号处理理论应用于该问题，即通过一种称为小波压缩的技术。小波是一种函数，其振幅像波一样振荡，但持续时间非常

at 15 samples per second, and doing so cuts the animation data size in half.

- *Omit some of the samples.* If a channel's data varies in an approximately linear fashion during some interval of time within the clip, we can omit all of the samples in this interval except the endpoints. Then, at runtime, we can use linear interpolation to recover the dropped samples.

The latter technique is a bit involved, and it requires us to store information about the *time* of each sample. This additional data can erode the savings we achieved by omitting samples in the first place. However, some game engines have used this technique successfully.

12.8.4 Curve-Based Compression

One of the most powerful, easiest-to-use and best-thought-out animation APIs I've ever worked with is Granny, by Rad Game Tools. Granny stores animations not as a regularly spaced sequence of pose samples but as a collection of *n*th-order, nonuniform, nonrational B-splines, describing the paths of a joint's S, Q and T channels over time. Using B-splines allows channels with a lot of curvature to be encoded using only a few data points.

Granny exports an animation by sampling the joint poses at regular intervals, much like traditional animation data. For each channel, Granny then fits a set of B-splines to the sampled dataset to within a user-specified tolerance. The end result is an animation clip that is usually significantly smaller than its uniformly sampled, linearly interpolated counterpart. This process is illustrated in Figure 12.46.

Figure 12.46. One form of animation compression fits B-splines to the animation channel data.

12.8.5 Wavelet Compression

Another way to compress animation data is to apply *signal processing theory* to the problem, via a technique known as *wavelet compression*. A wavelet is a function whose amplitude oscillates like a wave but whose duration is very

短暂，就像池塘里短暂的涟漪。小波函数经过精心设计，赋予其在信号处理中所需的特性。

在小波压缩中，动画曲线被分解为一系列正交小波的和，其方式与任意信号可以表示为一系列δ函数或正弦函数之和的方式非常相似。我们将在第14.2节深入讨论信号处理和线性时不变系统；其中提出的概念构成了理解小波压缩的必要基础。基于小波的压缩技术的完整讨论超出了本书的范围，但您可以在线阅读更多相关内容。搜索“wavelet”（小波）可以找到关于该主题的介绍性文章，然后尝试在Nicholas Frechette的博客上搜索“动画压缩：信号处理”，您会找到一篇关于Eidos Montreal公司如何为《Thief》（2014）实现小波压缩的精彩文章。

12.8.6 选择性加载和流式传输

最廉价的动画剪辑是根本不占用内存的动画剪辑。大多数游戏不需要所有动画剪辑同时占用内存。有些动画剪辑仅适用于特定类型的角色，因此在从未遇到该类型角色的关卡中无需加载它们。其他动画剪辑适用于游戏中的一次性场景。这些动画剪辑可以在需要之前加载或流式传输到内存中，并在播放结束后从内存中转储。

大多数游戏在首次启动时会将一组核心动画剪辑加载到内存中，并在整个游戏过程中保留它们。这些剪辑包括玩家角色的核心动作集，以及应用于游戏过程中反复出现的对象的动画，例如武器或强化道具。所有其他动画通常根据需要加载。有些游戏引擎会单独加载动画剪辑，但许多引擎会将它们打包成逻辑组，以便作为一个单元进行加载和卸载。

12.9 动画管线

低级动画引擎执行的操作形成一个管道，将其输入（动画剪辑和混合规范）转换为所需的输出（局部和全局姿势，加上用于渲染的矩阵调色板）。

对于游戏中的每个动画角色和对象，动画管道会将一个或多个动画剪辑和相应的混合因子作为输入，将它们混合在一起，并生成单个局部骨骼姿势作为输出。它还会计算骨骼的全局姿势和蒙皮调色板

short, like a brief ripple in a pond. Wavelet functions are carefully crafted to give them desirable properties for use in signal processing.

In wavelet compression, an animation curve is decomposed into a sum of orthonormal wavelets, in much the same way that an arbitrary signal can be represented as a train of delta functions or a sum of sinusoids. We discuss signal processing and linear time-invariant systems in some depth in Section 14.2; the concepts presented there form the foundations necessary to understand wavelet compression. A full discussion of wavelet-based compression techniques is well beyond the scope of this book, but you can read more about it online. Search for “wavelet” to find introductory articles on the topic, and then try searching for “Animation Compression: Signal Processing” on Nicholas Frechette’s blog for a great article on how wavelet compression was implemented for *Thief* (2014) by Eidos Montreal.

12.8.6 Selective Loading and Streaming

The cheapest animation clip is the one that isn’t in memory at all. Most games don’t need every animation clip to be in memory simultaneously. Some clips apply only to certain classes of character, so they needn’t be loaded during levels in which that class of character is never encountered. Other clips apply to one-off moments in the game. These can be loaded or streamed into memory just before being needed and dumped from memory once they have played.

Most games load a core set of animation clips into memory when the game first boots and keep them there for the duration of the game. These include the player character’s core move set and animations that apply to objects that reappear over and over throughout the game, such as weapons or power-ups. All other animations are usually loaded on an as-needed basis. Some game engines load animation clips individually, but many package them together into logical groups that can be loaded and unloaded as a unit.

12.9 The Animation Pipeline

The operations performed by the low-level animation engine form a *pipeline* that transforms its inputs (animation clips and blend specifications) into the desired outputs (local and global poses, plus a matrix palette for rendering).

For each animating character and object in the game, the animation pipeline takes one or more animation clips and corresponding blend factors as input, blends them together, and generates a single local skeletal pose as output. It also calculates a global pose for the skeleton and a palette of skinning

供渲染引擎使用的矩阵。通常提供后处理钩子，允许在最终全局姿势和矩阵调色板生成之前修改局部姿势。逆运动学 (IK)、布娃娃物理和其他形式的程序化动画在此应用于骨架。

该管道的阶段如下：

1. 片段解压和姿势提取。在此阶段，每个片段的数据都会被解压，并提取出与时间点对应的静态姿势。此阶段的输出是每个输入片段的局部骨骼姿势。该姿势可能包含骨架中所有关节的信息（全身姿势），也可能仅包含部分关节的信息（部分姿势），或者可能是一个用于加法混合的差异姿势。

2. 姿势混合。在此阶段，输入姿势通过全身 LERP 混合、部分骨架 LERP 混合和/或加法混合进行组合。此阶段的输出是骨架中所有关节的单个局部姿势。当然，此阶段仅在混合多个动画剪辑时执行——否则，可以直接使用阶段 1 的输出姿势。

3. 全局姿势生成。在此阶段，遍历骨骼层次结构，并将局部关节姿势连接起来，以生成骨骼的全局姿势。

4. 后处理。在此可选阶段，可以在最终确定姿势之前修改骨架的局部和/或后期处理用于逆运动学、布娃娃物理和其他形式的程序动画调整。

5. 重新计算全局姿态。许多类型的后处理需要全局姿态信息作为输入，但生成局部姿态作为输出。此类后处理步骤运行后，我们必须根据修改后的局部姿态重新计算全局姿态。显然，可以在第 2 阶段和第 3 阶段之间进行不需要全局姿态信息的后处理操作，从而避免重新计算全局姿态。

6. 矩阵调色板生成。生成最终的全局姿势后，每个关节的全局姿势矩阵将乘以相应的逆绑定姿势矩阵。此阶段的输出是适合输入渲染引擎的蒙皮矩阵调色板。

图 12.47 描述了典型的动画管道。

matrices for use by the rendering engine. Post-processing hooks are usually provided, which permit the local pose to be modified prior to final global pose and matrix palette generation. This is where inverse kinematics (IK), rag doll physics and other forms of procedural animation are applied to the skeleton.

The stages of this pipeline are:

1. *Clip decompression and pose extraction*. In this stage, each individual clip's data is decompressed, and a static pose is extracted for the time index in question. The output of this phase is a local skeletal pose for each input clip. This pose might contain information for every joint in the skeleton (a *full-body pose*), for only a subset of joints (a *partial pose*), or it might be a *difference pose* for use in additive blending.

2. *Pose blending*. In this stage, the input poses are combined via full-body LERP blending, partial-skeleton LERP blending and/or additive blending. The output of this stage is a single local pose for all joints in the skeleton. This stage is of course only executed when blending more than one animation clip together—otherwise the output pose from stage 1 can be used directly.

3. *Global pose generation*. In this stage, the skeletal hierarchy is walked, and local joint poses are concatenated in order to generate a global pose for the skeleton.

4. *Post-processing*. In this optional stage, the local and/or global poses of the skeleton can be modified prior to finalization of the pose. Post-processing is used for inverse kinematics, rag doll physics and other forms of procedural animation adjustment.

5. *Recalculation of global poses*. Many types of post-processing require global pose information as input but generate local poses as output. After such a post-processing step has run, we must recalculate the global pose from the modified local pose. Obviously, a post-processing operation that does not require global pose information can be done between stages 2 and 3, thus avoiding the need for global pose recalculation.

6. *Matrix palette generation*. Once the final global pose has been generated, each joint's global pose matrix is multiplied by the corresponding inverse bind pose matrix. The output of this stage is a palette of skinning matrices suitable for input to the rendering engine.

A typical animation pipeline is depicted in Figure 12.47.

图 12.47.典型的动画管道。

12.10 动作状态机

游戏角色的动作（站立、行走、奔跑、跳跃等）通常最好通过有限状态机（通常称为动作状态机 (ASM)）来建模。ASM 子系统位于动画管线的顶层，提供状态驱动的动画接口，几乎所有高级游戏代码都可以使用。

ASM 中的每个状态都对应于一个任意复杂的同步动画剪辑混合。有些状态可能非常简单 - 例如，“空闲”状态可能由单个全身动画组成。其他状态可能更复杂。“奔跑”状态可能对应于半圆形混合，分别在 - 90 度、0 度和 + 90 度点向左扫射、向前奔跑和向右扫射。“边跑边射击”状态可能包含半圆形方向混合，以及用于上下左右瞄准角色武器的附加或部分骨架混合节点，以及允许角色用眼睛、头部和肩膀环顾四周的其他混合。可能会包含更多附加动画，以控制角色在运动时的整体姿势、步态和脚间距，并通过随机的运动变化提供一定程度的“人性化”。

Figure 12.47. A typical animation pipeline.

12.10 Action State Machines

The actions of a game character (standing, walking, running, jumping, etc.) are usually best modeled via a finite state machine, commonly known as the *action state machine* (ASM). The ASM subsystem sits atop the animation pipeline and provides a state-driven animation interface for use by virtually all higher-level game code.

Each state in an ASM corresponds to an arbitrarily complex blend of simultaneous animation clips. Some states might be very simple—for example, the “idle” state might be comprised of a single full-body animation. Other states might be more complex. A “running” state might correspond to a semicircular blend, with strafing left, running forward and strafing right at the −90 degree, 0 degree and +90 degree points, respectively. The “running while shooting” state might include a semicircular directional blend, plus additive or partial-skeleton blend nodes for aiming the character’s weapon up, down, left and right, and additional blends to permit the character to look around with its eyes, head and shoulders. More additive animations might be included to control the character’s overall stance, gait and foot spacing while locomoting and to provide a degree of “humanness” through random movement variations.

图 12.48。分层动作状态机，展示了各层的状态转换如何在时间上独立。在此示例中，基础层描述了角色的全身姿势和动作。变化层通过对角色的姿势应用附加剪辑来提供变化。最后，两个手势层（一个附加层和一个部分层）允许角色瞄准或指向周围世界中的物体。

角色的 ASM 还能确保角色在不同状态之间平滑过渡。从状态 A 过渡到状态 B 时，两个状态的最终输出姿势通常会混合在一起，以实现两者之间平滑的淡入淡出效果。

大多数高质量动画引擎还允许角色身体的不同部位同时执行不同的、独立或半独立的动作。例如，一个角色可能在奔跑，用手臂瞄准和射击武器，并用面部关节说一句台词。身体不同部位的运动通常也不是完全同步的——某些身体部位倾向于“引导”其他部位的运动（例如，头部引导转身，然后是肩膀、臀部，最后是腿部）。在传统动画中，这种众所周知的技术被称为预期 [51]。这种复杂的运动可以通过允许多个独立的状态机控制单个角色来实现。通常每个状态机存在于一个单独的状态层中，如图 12.48 所示。每个层的 ASM 的输出姿势混合在一起形成最终的复合姿势。

所有这些意味着在任何给定的时刻，多个动画

Figure 12.48. A layered action state machine, showing how each layer's state transitions are temporally independent. In this example, the base layer describes the character's full-body stance and movement. A variation layer provides variety by applying additive clips to the character's pose. Finally, two gesture layers, one additive and one partial, permit the character to aim or point at objects in the world around it.

A character's ASM also ensures that characters can *transition* smoothly from state to state. During a transition from state A to state B, the final output poses of both states are usually blended together to provide a smooth *cross-fade* between them.

Most high-quality animation engines also permit different parts of a character's body to be doing different, independent or semi-independent actions simultaneously. For instance, a character might be running, aiming and firing a weapon with its arms, and speaking a line of dialog with its facial joints. The movements of different parts of the body aren't generally in perfect sync either—certain parts of the body tend to "lead" the movements of other parts (e.g., the head leads a turn, followed by the shoulders, the hips and finally the legs). In traditional animation, this well-known technique is known as *anticipation* [51]. This kind of complex movement can be realized by allowing multiple independent state machines to control a single character. Usually each state machine exists in a separate *state layer*, as shown in Figure 12.48. The output poses from each layer's ASM are blended together into a final composite pose.

All of this means that at any given moment in time, multiple animation

片段会影响角色骨架的最终姿势。因此，对于每个角色，我们需要一种方法来追踪所有当前正在播放的片段，并描述如何将它们精确地混合在一起以产生角色的最终姿势。一般来说，有两种方法可以做到这一点：

1. 平面加权平均。在这种方法中，引擎会维护一个平面列表，其中包含当前构成角色最终姿势的所有动画剪辑，每个剪辑对应一个混合权重。所有动画将混合在一起，形成一个大的加权平均值，以生成最终姿势。

2. 混合树。在这种方法中，每个参与的剪辑都由树的叶节点表示。该树的内部节点表示正在对剪辑执行的各种混合操作。多个混合操作组合在一起形成动作状态。引入额外的混合节点来表示瞬态交叉淡入淡出。在分层的ASM中，从每一层的动作状态获得的输出姿势会混合在一起。因此，角色的最终姿势是在这棵可能很复杂的混合树的根节点生成的。

12.10.1 平面加权平均法

在平面加权平均方法中，给定角色当前正在播放的每个动画剪辑都与一个混合权重相关联，该权重指示该剪辑对最终姿势的贡献程度。系统会维护一个包含所有活动动画剪辑（即混合权重非零的剪辑）的平面列表。为了计算骨架的最终姿势，我们会为 N 个活动剪辑中的每个剪辑提取其在相应时间索引处的姿势。然后，对于骨架的每个关节，我们会计算从 N 个活动动画中提取的平移向量、旋转四元数和比例因子的简单 N 点加权平均值。由此即可得出骨架的最终姿势。

一组 N 个向量 { vi } 的加权平均值公式如下：

$$\mathbf{v}_{\text{avg}} = \frac{\sum_{i=0}^{N-1} w_i \mathbf{v}_i}{\sum_{i=0}^{N-1} w_i}.$$

如果权重被标准化，即它们的总和为 1，那么这个等式可以简化为如下形式：

$$\mathbf{v}_{\text{avg}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_i, \qquad \text{when } \sum_{i=0}^{N-1} w_i = 1.$$

clips are contributing to the final pose of a character's skeleton. For each character, then, we need a way to track all of the currently-playing clips, and to describe how exactly they should be blended together in order to produce the character's final pose. Generally speaking, there are two ways to do this:

1. *Flat weighted average*. In this approach, the engine maintains a flat list of all animation clips that are currently contributing to a character's final pose, with one blend weight per clip. The animations are blended together as one big weighted average to produce the final pose.
2. *Blend trees*. In this approach, each contributing clip is represented by the leaf nodes of a tree. The interior nodes of this tree represent various blending operations that are being performed on the clips. Multiple blend operations are composed to form action states. Additional blend nodes are introduced to represent transient cross-fades. And in a layered ASM, the output poses obtained from the action states in each layer are blended together. The final pose of the character is thus produced at the root of this potentially complex blend tree.

12.10.1 The Flat Weighted Average Approach

In the flat weighted average approach, every animation clip that is currently playing on a given character is associated with a blend weight indicating how much it should contribute to its final pose. A flat list of all *active* animation clips (i.e., clips whose blend weights are nonzero) is maintained. To calculate the final pose of the skeleton, we extract a pose at the appropriate time index for each of the N active clips. Then, for each joint of the skeleton, we calculate a simple N-point weighted average of the translation vectors, rotation quaternions and scale factors extracted from the N active animations. This yields the final pose of the skeleton.

The equation for the weighted average of a set of N vectors $\{\mathbf{v}_i\}$ is as follows:

$$\mathbf{v}_{\text{avg}} = \frac{\sum_{i=0}^{N-1} w_i \mathbf{v}_i}{\sum_{i=0}^{N-1} w_i}.$$

If the weights are *normalized*, meaning they sum to one, then this equation can be simplified to the following:

$$\mathbf{v}_{\text{avg}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_i, \qquad \text{when } \sum_{i=0}^{N-1} w_i = 1.$$

在 N = 2 的情况下，如果我们让 w 0 = (1 − β) 和 w 1 = β ，则加权平均值简化为两个向量之间的线性插值 (LERP) 的熟悉方程：

$$\begin{aligned}\mathbf{v}_{\text{avg}} &= w_0\mathbf{v}_A + w_1\mathbf{v}_B \\ &= (1-\beta)\,\mathbf{v}_A + \beta\mathbf{v}_B \\ &= \text{LERP}[\text{vA，vB，}\beta]\text{。}\end{aligned}$$

我们可以将相同的加权平均公式同样应用于四元数，只需将它们视为四元素向量即可。

12.10.1.1 例如：OGRE

OGRE 动画系统正是如此运作的。一个 Ogre::Entity 代表一个 3D 网格的实例（例如，一个在游戏世界中行走的特定角色）。该 Entity 聚合了一个名为 Ogre::AnimationStateSet 的对象，该对象又维护了一个 Ogre::AnimationState 对象列表，每个活动动画对应一个对象。Ogre::AnimationState 类如下面代码片段所示。（为了清晰起见，省略了一些不相关的细节。）

```
/** Represents the state of an animation clip and the
    weight of its influence on the overall pose of the
    character.
*/
class AnimationState
{
protected:
    String            mAnimationName; // reference to
                                      // clip
    Real              mTimePos;       // local clock
    Real              mWeight;        // blend weight
    bool              mEnabled;       // is this anim
                                      // running?
    bool              mLoop;          // should the
                                      // anim loop?

public:
    /// API functions...
};
```

每个动画状态都会跟踪一个动画剪辑的本地时钟及其混合权重。当计算某个 Ogre::Entity 的骨架最终姿势时，OGRE 的动画系统会简单地循环遍历每个

In the case of $N = 2$, if we let $w_0 = (1-\beta)$ and $w_1 = \beta$, the weighted average reduces to the familiar equation for the linear interpolation (LERP) between two vectors:

$$\begin{aligned}\mathbf{v}_{\text{avg}} &= w_0\mathbf{v}_A + w_1\mathbf{v}_B \\ &= (1-\beta)\,\mathbf{v}_A + \beta\mathbf{v}_B \\ &= \text{LERP}\left[\mathbf{v}_A, \mathbf{v}_B, \beta\right].\end{aligned}$$

We can apply this same weighted average formulation equally well to quaternions by simply treating them as four-element vectors.

12.10.1.1 Example: OGRE

The OGRE animation system works in exactly this way. An `Ogre::Entity` represents an instance of a 3D mesh (e.g., one particular character walking around in the game world). The `Entity` aggregates an object called an `Ogre::AnimationStateSet`, which in turn maintains a list of `Ogre::AnimationState` objects, one for each active animation. The `Ogre::AnimationState` class is shown in the code snippet below. (A few irrelevant details have been omitted for clarity.)

```
/** Represents the state of an animation clip and the
    weight of its influence on the overall pose of the
    character.
*/
class AnimationState
{
protected:
    String            mAnimationName; // reference to
                                      // clip
    Real              mTimePos;       // local clock
    Real              mWeight;        // blend weight
    bool              mEnabled;       // is this anim
                                      // running?
    bool              mLoop;          // should the
                                      // anim loop?

public:
    /// API functions...
};
```

Each `AnimationState` keeps track of one animation clip's local clock and its blend weight. When calculating the final pose of the skeleton for a particular `Ogre::Entity`, OGRE's animation system simply loops through each

在其 AnimationStateSet 中激活 AnimationState。从与每个状态对应的动画剪辑中提取骨骼姿势，该姿势由该状态的本地时钟指定的时间索引决定。然后，对于骨架中的每个关节，计算平移向量、旋转四元数和缩放的 N 点加权平均值，得出最终的骨骼姿势。

值得注意的是，OGRE 没有播放速率（R）的概念。如果有的话，我们本应在 Ogre::AnimationState 类中看到如下数据成员：

```
Real    mPlaybackRate;
```

当然，我们仍然可以通过简单地缩放传递给 addTime() 函数的时间量来使动画在 OGRE 中播放得更慢或更快，但不幸的是，OGRE 不支持开箱即用的动画时间缩放。

12.10.1.2 例如：奶奶

Rad Game Tools (http://www.radgame tool.com/granny.html) 开发的 Granny 动画系统提供了一个类似于 OGRE 的扁平化加权平均动画混合系统。Granny 允许在单个角色上同时播放任意数量的动画。每个活动动画的状态都保存在一个名为 granny_control 的数据结构中。Granny 会计算加权平均值来确定最终姿势，并自动对所有活动动画片段的权重进行归一化。从这个意义上讲，它的架构与 OGRE 的动画系统几乎完全相同。

Granny 真正出彩的地方在于它对时间的处理。Granny 使用了 12.4.3 节中讨论过的全局时钟方法。它允许每个剪辑循环播放任意次数或无限次。剪辑还可以进行时间缩放；负时间缩放允许动画反向播放。

12.10.1.3 具有平坦加权平均值的交叉淡入淡出

在采用平面加权平均架构的动画引擎中，交叉淡入淡出是通过调整片段本身的权重来实现的。回想一下，任何权重 wi = 0 的片段都不会对角色的当前姿势产生影响，而权重非零的片段则会被平均以生成最终姿势。如果我们希望从片段 A 平滑过渡到片段 B，只需增加片段 B 的权重 wB，同时降低片段 A 的权重 w A。如图 12.49 所示。

当我们想要从一种复杂的混合过渡到另一种复杂的混合时，加权平均架构中的交叉淡入淡出会变得有点棘手。举个例子，假设我们希望将角色从行走过渡到跳跃。

active `AnimationState` in its `AnimationStateSet`. A skeletal pose is extracted from the animation clip corresponding to each state at the time index specified by that state's local clock. For each joint in the skeleton, an N-point weighted average is then calculated for the translation vectors, rotation quaternions and scales, yielding the final skeletal pose.

It is interesting to note that OGRE has no concept of a playback rate (R). If it did, we would have expected to see a data member like this in the `Ogre::AnimationState` class:

```
Real    mPlaybackRate;
```

Of course, we can still make animations play more slowly or more quickly in OGRE by simply scaling the amount of time we pass to the `addTime()` function, but unfortunately, OGRE does not support animation time scaling out of the box.

12.10.1.2 Example: Granny

The Granny animation system, by Rad Game Tools (http://www.radgame tool.com/granny.html), provides a flat, weighted average animation blending system similar to OGRE's. Granny permits any number of animations to be played on a single character simultaneously. The state of each active animation is maintained in a data structure known as a `granny_control`. Granny calculates a weighted average to determine the final pose, automatically normalizing the weights of all active clips. In this sense, its architecture is virtually identical to that of OGRE's animation system.

Where Granny really shines is in its handling of time. Granny uses the global clock approach discussed in Section 12.4.3. It allows each clip to be looped an arbitrary number of times or infinitely. Clips can also be time-scaled; a negative time scale allows an animation to be played in reverse.

12.10.1.3 Cross-Fades with a Flat Weighted Average

In an animation engine that employs the flat weighted average architecture, cross-fades are implemented by adjusting the weights of the clips themselves. Recall that any clip whose weight $w_i = 0$ will not contribute to the current pose of the character, while those whose weights are nonzero are averaged together to generate the final pose. If we wish to transition smoothly from clip A to clip B, we simply ramp up clip B's weight w_B, while simultaneously ramping down clip A's weight w_A. This is illustrated in Figure 12.49.

Cross-fading in a weighted average architecture becomes a bit trickier when we wish to transition from one complex blend to another. As an example, let's say we wish to transition the character from walking to jumping.

图 12.49.从剪辑 A 到剪辑 B 的简单交叉淡入淡出，采用加权平均动画架构实现。

假设行走动作是由剪辑 A、B 和 C 之间的三向平均产生的，而跳跃动作是由剪辑 D 和 E 之间的双向平均产生的。

我们希望角色看起来从行走到跳跃的过渡流畅，且不影响行走或跳跃动画的单独呈现效果。因此，在过渡过程中，我们希望降低 ABC 剪辑的权重，并增加 DE 剪辑的权重，同时保持 ABC 和 DE 剪辑组的相对权重不变。如果交叉淡入淡出的混合因子用 λ 表示，我们只需将两个剪辑组的权重设置为所需值，然后将源组的权重乘以 (1 − λ)，将目标组的权重乘以 λ 即可满足此要求。

让我们看一个具体的例子来验证一下这种方法是否有效。假设在从 ABC 到 DE 的转换之前，非零权重分别为：wA = 0.2、wB = 0.3 和 wC = 0.5。转换之后，我们希望非零权重分别为 wD = 0.33 和 wE = 0.66。因此，我们将权重设置如下：

$$\begin{aligned} w_A &= (1-\lambda)(0.2), & w_D &= \lambda(0.33), \\ w_B &= (1-\lambda)(0.3), & w_E &= \lambda(0.66). \\ w_C &= (1-\lambda)(0.5), \end{aligned} \tag{12.20}$$

根据公式（12.20），您应该能够确信以下内容：

1. 当λ= 0时，输出姿态是剪辑A、B和C的正确混合，剪辑D和E的贡献为零。

2. 当λ= 1时，输出姿势是剪辑D和E的正确混合，不受A、B或C的影响。

3. 当 0 < λ < 1 时，ABC 组和 DE 组的相对权重仍然正确，尽管它们的和不再等于 1。（实际上，ABC 组的权重加起来等于（1 − λ），DE 组的权重加起来等于 λ。）

Figure 12.49. A simple cross-fade from clip A to clip B, as implemented in a weighted average animation architecture.

Let's assume that the walk movement is produced by a three-way average between clips A, B and C, and that the jump movement is produced by a two-way average between clips D and E.

We want the character to look like he's smoothly transitioning from walking to jumping, without affecting how the walk or jump animations look individually. So during the transition, we want to ramp down the ABC clips and ramp up the DE clips while keeping the *relative weights* of the ABC and DE clip groups constant. If the cross-fade's blend factor is denoted by λ, we can meet this requirement by simply setting the weights of *both* clip groups to their desired values and then multiplying the weights of the source group by $(1-\lambda)$ and the weights of the destination group by λ.

Let's look at a concrete example to convince ourselves that this will work properly. Imagine that before the transition from ABC to DE, the nonzero weights are as follows: $w_A = 0.2$, $w_B = 0.3$ and $w_C = 0.5$. After the transition, we want the nonzero weights to be $w_D = 0.33$ and $w_E = 0.66$. So, we set the weights as follows:

$$\begin{aligned} w_A &= (1-\lambda)(0.2), & w_D &= \lambda(0.33), \\ w_B &= (1-\lambda)(0.3), & w_E &= \lambda(0.66). \\ w_C &= (1-\lambda)(0.5), \end{aligned} \tag{12.20}$$

From Equations (12.20), you should be able to convince yourself of the following:

1. When $\lambda = 0$, the output pose is the correct blend of clips A, B and C, with zero contribution from clips D and E.

2. When $\lambda = 1$, the output pose is the correct blend of clips D and E, with no contribution from A, B or C.

3. When $0 < \lambda < 1$, the *relative* weights of both the ABC group and the DE group remain correct, although they no longer add to one. (In fact, group ABC's weights add to $(1-\lambda)$, and group DE's weights add to λ.)

图 12.50。二进制 LERP 混合，由二进制表达式树表示。

为了使这种方法有效，实现必须跟踪剪辑之间的逻辑分组（即使在最底层，所有剪辑的状态都维护在一个大的扁平数组中——例如，OGRE 中的 Ogre::AnimationStateSet）。在上面的例子中，系统必须“知道” A、B 和 C 组成一个组，D 和 E 组成另一个组，并且我们希望从组 ABC 过渡到组 DE。这需要在扁平的剪辑状态数组之上维护额外的元数据。

12.10.2 混合树

一些动画引擎不是将角色的剪辑状态表示为平面加权平均值，而是表示为混合操作树。动画混合树是编译器理论中称为表达式树或语法树的示例。这种树的内部节点是运算符，叶节点用作这些运算符的输入。（更准确地说，内部节点表示语法的非终结符，而叶节点表示终结符。）在以下各节中，我们将简要回顾在 12.6.3 和 12.6.5 节中学习到的各种动画混合，并了解每种动画混合如何用表达式树表示。

12.10.2.1 二叉 LERP 混合树

正如我们在 12.6.1 节中看到的，二元线性插值 (LERP) 混合算法将两个输入姿态混合成一个输出姿态。混合权重 β 控制第二个输入姿态在输出中出现的百分比，而 (1 − β) 指定第一个输入姿态的百分比。这可以用图 12.50 所示的二元表达式树来表示。

12.10.2.2 广义一维混合树

在 12.6.3.1 节中，我们了解到，通过沿线性尺度放置任意数量的剪辑，可以方便地定义广义的一维 LERP 混合。混合因子 b□□ 指定了沿该尺度所需的混合。这种混合可以表示为一个 n 输入算子，如图 12.51 所示。

给定 b 的特定值，这种线性混合总是可以转换为二进制 LERP 混合。我们只需使用紧邻的两个剪辑

Figure 12.50. A binary LERP blend, represented by a binary expression tree.

For this approach to work, the implementation must keep track of the logical groupings between clips (even though, at the lowest level, all of the clips' states are maintained in one big, flat array—for example, the `Ogre::AnimationStateSet` in OGRE). In our example above, the system must "know" that A, B and C form a group, that D and E form another group, and that we wish to transition from group ABC to group DE. This requires additional metadata to be maintained, on top of the flat array of clip states.

12.10.2 Blend Trees

Some animation engines represent a character's clip state not as a flat weighted average but rather as a tree of blend operations. An animation blend tree is an example of what is known in compiler theory as an *expression tree* or a *syntax tree*. The interior nodes of such a tree are operators, and the leaf nodes serve as the inputs to those operators. (More correctly, the interior nodes represent the *nonterminals* of the grammar, while the leaf nodes represent the *terminals*.)

In the following sections, we'll briefly revisit the various kinds of animation blends we learned about in Sections 12.6.3 and 12.6.5 and see how each can be represented by an expression tree.

12.10.2.1 Binary LERP Blend Trees

As we saw in Section 12.6.1, a binary linear interpolation (LERP) blend takes two input poses and blends them together into a single output pose. A blend weight β controls the percentage of the second input pose that should appear at the output, while $(1 - \beta)$ specifies the percentage of the first input pose. This can be represented by the binary expression tree shown in Figure 12.50.

12.10.2.2 Generalized One-Dimensional Blend Trees

In Section 12.6.3.1, we learned that it can be convenient to define a generalized one-dimensional LERP blend by placing an arbitrary number of clips along a linear scale. A blend factor b specifies the desired blend along this scale. Such a blend can be pictured as an n-input operator, as shown in Figure 12.51.

Given a specific value for b, such a linear blend can always be transformed into a binary LERP blend. We simply use the two clips immediately adjacent to

图 12.51。多输入表达式树可用于表示广义的一维混合。对于任意特定的混合因子 b□□ 值，这样的树总是可以转换为二叉表达式树。

图 12.52.一个简单的 2D LERP 混合，以级联二进制混合的形式实现。

b 作为二元混合的输入，并计算混合权重 β，如公式 (12.15) 所示

12.10.2.3 二维 LERP 混合树

在 12.6.3.2 节中，我们了解了如何通过简单地级联两个二元 LERP 混合的结果来实现二维 LERP 混合。给定一个所需的二维混合点 b = [bxby] ，图 12.52 展示了如何以树形形式表示这种混合。

12.10.2.4 加法混合树

12.6.5 节介绍了加法混合。这是一个二元运算，因此可以用二叉树节点表示，如图 12.53 所示。单个混合权重 β 控制应在

Figure 12.51. A multi-input expression tree can be used to represent a generalized 1D blend. Such a tree can always be transformed into a binary expression tree for any specific value of the blend factor b.

Figure 12.52. A simple 2D LERP blend, implemented as cascaded binary blends.

b as the inputs to the binary blend and calculate the blend weight β as specified in Equation (12.15)

12.10.2.3 Two-Dimensional LERP Blend Trees

In Section 12.6.3.2, we saw how a two-dimensional LERP blend can be realized by simply cascading the results of two binary LERP blends. Given a desired two-dimensional blend point $\mathbf{b} = \begin{bmatrix} b_x & b_y \end{bmatrix}$, Figure 12.52 shows how this kind of blend can be represented in tree form.

12.10.2.4 Additive Blend Trees

Section 12.6.5 described additive blending. This is a binary operation, so it can be represented by a binary tree node, as shown in Figure 12.53. A single blend weight β controls the amount of the additive animation that should appear at

图 12.53. 以二叉树表示的加法混合。

图 12.54.为了将多个差异姿势混合到常规的“基础”姿势上，必须使用级联二叉表达式树。

输出——当 β = 0 时，加性剪辑根本不影响输出，而当 β = 1 时，加性剪辑对输出的影响最大。

必须小心处理加法混合节点，因为输入是不可互换的（与大多数类型的混合运算符一样）。两个输入之一是常规骨骼姿势，而另一个输入是一种称为差异姿势（也称为加法姿势）的特殊姿势。差异姿势只能应用于常规姿势，而加法混合的结果是另一个常规姿势。这意味着混合节点的加法输入必须始终是叶节点，而常规输入可以是叶节点或内部节点。如果我们想对我们的角色应用多个加法动画，我们必须使用级联二叉树，并将加法剪辑始终应用于加法输入，如图 12.54 所示。

12.10.2.5 分层混合树

我们在 12.10 节的开头提到，可以通过将多个独立的状态机排列成状态层来实现复杂的角色运动。每个状态机层 ASM 的输出姿势会被混合在一起，形成最终的复合姿势。当使用混合树实现这一点时，最终效果是将每个活动状态的混合树组合成一棵超级树，如图 12.55 所示。

Figure 12.53. An additive blend represented as a binary tree.

Figure 12.54. In order to additively blend more than one difference pose onto a regular “base” pose, a cascaded binary expression tree must be used.

the output—when $\beta = 0$, the additive clip does not affect the output at all, while when $\beta = 1$, the additive clip has its maximum effect on the output.

Additive blend nodes must be handled carefully, because the inputs are not interchangeable (as they are with most types of blend operators). One of the two inputs is a regular skeletal pose, while the other is a special kind of pose known as a *difference pose* (also known as an *additive pose*). A difference pose may *only* be applied to a regular pose, and the result of an additive blend is another regular pose. This implies that the additive input of a blend node must always be a leaf node, while the regular input may be a leaf or an interior node. If we want to apply more than one additive animation to our character, we must use a cascaded binary tree with the additive clips always applied to the additive inputs, as shown in Figure 12.54.

12.10.2.5 Layered Blend Trees

We said at the beginning of Section 12.10 that complex character movement can be produced by arranging multiple independent state machines into *state layers*. The output poses from each layer's ASM are blended together into a final composite pose. When this is implemented using blend trees, the net effect is to combine the blend trees of each active state together into one über tree, as illustrated in Figure 12.55.

图 12.55 分层状态机将多个状态的混合树转换为单个统一的树。

12.10.2.6 使用混合树实现交叉淡入淡出

当角色在分层 ASM 的每一层内从一个状态过渡到另一个状态时，我们通常希望在状态之间提供平滑的交叉淡入淡出效果。在基于表达式树的 ASM 中实现交叉淡入淡出比在加权平均架构中更直观一些。无论是从一个剪辑过渡到另一个剪辑，还是从一个复杂混合过渡到另一个复杂混合，方法始终相同：我们只需在每个状态的混合树的根节点之间引入一个瞬态二进制 LERP 节点来处理交叉淡入淡出效果。

我们将像之前一样用符号 λ 表示交叉淡入淡出节点的混合因子。其顶部输入是源状态的混合树（可以是单个剪辑或复杂混合），其底部输入是目标状态的树（同样是剪辑或复杂混合）。在过渡过程中，λ 从 0 递增到 1。一旦 $\lambda = 1$，过渡就完成了，交叉淡入淡出 LERP 节点及其顶部输入树就可以退出了。这使得其底部输入树成为

Figure 12.55. A layered state machine converts the blend trees from multiple states into a single, unified tree.

12.10.2.6 Cross-Fades with Blend Trees

As a character transitions from state to state within each layer of a layered ASM, we often wish to provide a smooth cross-fade between states. Implementing a cross-fade in an expression tree based ASM is a bit more intuitive than it is in a weighted average architecture. Whether we're transitioning from one clip to another or from one complex blend to another, the approach is always the same: We simply introduce a transient binary LERP node between the roots of the blend trees of each state to handle the cross-fade.

We'll denote the blend factor of the cross-fade node with the symbol λ as before. Its top input is the source state's blend tree (which can be a single clip or a complex blend), and its bottom input is the destination state's tree (again a clip or a complex blend). During the transition, λ is ramped from zero to one. Once $\lambda = 1$, the transition is complete, and the cross-fade LERP node and its top input tree can be retired. This leaves its bottom input tree as the root of

图 12.56。任意两个混合树 A 和 B 之间的交叉淡入淡出。

给定状态层的整体混合树，从而完成转换。
图 12.56 说明了这个过程。

12.10.3 状态和混合树规范

动画师、游戏设计师和程序员通常会合作为游戏中的核心角色创建动画和控制系统。这些开发人员需要一种方法来指定构成角色 ASM 的状态、布置每个混合树的树形结构，并选择将作为其输入的剪辑。虽然状态和混合树可以硬编码，但大多数现代游戏引擎都提供了一种数据驱动的方式来定义动画状态。数据驱动方法的目标是允许用户创建新的动画状态、删除不需要的状态、微调现有状态，然后相当快速地查看其更改的效果。换句话说，数据驱动动画引擎的核心目标是实现快速迭代。

要构建任意复杂的混合树，我们实际上只需要四种原子类型的混合节点：剪辑、二元 LERP 混合、二元加法混合以及可能的三元（三角形）LERP 混合。几乎任何可以想象到的混合树都可以由这些原子节点组合而成。

Figure 12.56. A cross-fade between two arbitrary blend trees A and B.

the overall blend tree for the given state layer, thus completing the transition. This process is illustrated in Figure 12.56.

12.10.3 State and Blend Tree Specifications

Animators, game designers and programmers usually cooperate to create the animation and control systems for the central characters in a game. These developers need a way to specify the states that make up a character's ASM, to lay out the tree structure of each blend tree, and to select the clips that will serve as their inputs. Although the states and blend trees could be hard-coded, most modern game engines provide a *data-driven* means of defining animation states. The goal of a data-driven approach is to permit a user to create new animation states, remove unwanted states, fine-tune existing states and then see the effects of his or her changes reasonably quickly. In other words, the central goal of a data-driven animation engine is to enable *rapid iteration*.

To build an arbitrarily complex blend tree, we really only require four atomic types of blend nodes: clips, binary LERP blends, binary additive blends and possibly ternary (triangular) LERP blends. Virtually any blend tree imaginable can be created as compositions of these atomic nodes.

仅由原子节点构建的混合树很快就会变得庞大而难以处理。因此，许多游戏引擎允许预定义自定义复合节点类型以方便使用。第 12.6.3.4 节和 12.10.2.2 节讨论的 N 维线性混合节点就是复合节点的一个例子。我们可以想象无数种复杂的混合节点类型，每一种都针对特定游戏的特定问题。足球游戏可以定义一个允许角色带球的节点。战争游戏可以定义一个特殊节点来处理武器的瞄准和射击。格斗游戏可以为角色可以执行的每个战斗动作定义自定义节点。一旦我们能够定义自定义节点类型，一切皆有可能。

用户输入动画状态数据的方式多种多样。
一些游戏引擎采用简单、精简的方法，允许在文本文件中使用简单的语法指定动画状态。另一些引擎则提供流畅的图形编辑器，允许通过将原子组件（例如剪辑和混合节点）拖放到画布上并以任意方式链接在一起来构建动画状态。此类编辑器通常提供角色的实时预览，以便用户可以立即看到角色在最终游戏中的外观。在我看来，选择的具体方法对最终游戏的质量影响不大——最重要的是用户能够快速轻松地进行更改并查看更改的结果。

12.10.3.1 示例：顽皮狗引擎

顽皮狗的《神秘海域》和《最后生还者》系列中使用的动画引擎采用了一种简单的基于文本的方式来指定动画状态。由于顽皮狗与 Lisp 语言的悠久历史（参见第 16.9.5.1 节），顽皮狗引擎中的状态规范是用 Scheme 编程语言（其本身是 Lisp 的一个变体）的定制版本编写的。可以使用两种基本状态类型：简单 (simple) 和复杂 (complex)。

简单状态

简单状态包含单个动画剪辑。例如：

```
(define-state simple
    :name "pirate-b-bump-back"
    :clip "pirate-b-bump-back"
    :flags (anim-state-flag no-adjust-to-ground)
)
```


A blend tree built exclusively from atomic nodes can quickly become large and unwieldy. As a result, many game engines permit custom compound node types to be predefined for convenience. The N-dimensional linear blend node discussed in Sections 12.6.3.4 and 12.10.2.2 is an example of a compound node. One can imagine myriad complex blend node types, each one addressing a particular problem specific to the particular game being made. A soccer game might define a node that allows the character to dribble the ball. A war game could define a special node that handles aiming and firing a weapon. A brawler could define custom nodes for each fight move the characters can perform. Once we have the ability to define custom node types, the sky's the limit.

The means by which the users enter animation state data varies widely. Some game engines employ a simple, bare-bones approach, allowing animation states to be specified in a text file with a simple syntax. Other engines provide a slick, graphical editor that permits animation states to be constructed by dragging atomic components such as clips and blend nodes onto a canvas and linking them together in arbitrary ways. Such editors usually provide a live preview of the character so that the user can see immediately how the character will look in the final game. In my opinion, the specific method chosen has little bearing on the quality of the final game—what matters most is that the user can make changes and see the results of those changes reasonably quickly and easily.

12.10.3.1 Example: The Naughty Dog Engine

The animation engine used in Naughty Dog's *Uncharted* and *The Last of Us* franchises employs a simple, text-based approach to specifying animation states. For reasons related to Naughty Dog's rich history with the Lisp language (see Section 16.9.5.1), state specifications in the Naughty Dog engine are written in a customized version of the Scheme programming language (which itself is a Lisp variant). Two basic state types can be used: *simple* and *complex*.

Simple States

A *simple* state contains a single animation clip. For example:

```
(define-state simple
    :name "pirate-b-bump-back"
    :clip "pirate-b-bump-back"
    :flags (anim-state-flag no-adjust-to-ground)
)
```

别让 Lisp 风格的语法搞糊涂了。这段代码的作用是定义一个名为“pirate-b-bump-back”的状态，它的动画剪辑也恰好名为“pirate-b-bump-back”。:flags 参数允许用户为该状态指定各种布尔选项。

复杂状态

复杂状态包含任意 LERP 或加法混合树。例如，以下状态定义了一棵包含单个二元 LERP 混合节点的树，该节点包含两个片段（“walk-l-to-r”和“run-l-to-r”）作为输入：

```
(define-state complex
    :name "move-l-to-r"
    :tree
        (anim-node-lerp
            (anim-node-clip "walk-l-to-r")
            (anim-node-clip "run-l-to-r")
        )
)
```

:tree 参数允许用户指定任意混合树，由 LERP 或加法混合节点和播放单个动画剪辑的节点组成。

由此，我们可以看到上面显示的（定义状态简单...）示例实际上如何在后台工作 - 它可能定义了一个包含单个“剪辑”节点的复杂混合树，如下所示：

```
(define-state complex
    :name "pirate-b-unimog-bump-back"
    :tree (anim-node-clip "pirate-b-unimog-bump-back")
    :flags (anim-state-flag no-adjust-to-ground)
)
```

以下复杂状态显示了如何将混合节点级联到任意深度的混合树中：

```
(define-state complex
    :name "move-b-to-f"
    :tree
        (anim-node-lerp
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-f")
                    (anim-node-clip "move-f-look-lr")
                )
```


Don't let the Lisp-style syntax throw you. All this block of code does is to define a state named “pirate-b-bump-back” whose animation clip also happens to be named “pirate-b-bump-back.” The `:flags` parameter allows users to specify various Boolean options on the state.

Complex States

A *complex* state contains an arbitrary tree of LERP or additive blends. For example, the following state defines a tree that contains a single binary LERP blend node, with two clips (“walk-l-to-r” and “run-l-to-r”) as its inputs:

```
(define-state complex
    :name "move-l-to-r"
    :tree
        (anim-node-lerp
            (anim-node-clip "walk-l-to-r")
            (anim-node-clip "run-l-to-r")
        )
)
```

The `:tree` argument allows the user to specify an arbitrary blend tree, composed of LERP or additive blend nodes and nodes that play individual animation clips.

From this, we can see how the `(define-state simple ...)` example shown above might really work under the hood—it probably defines a complex blend tree containing a single “clip” node, like this:

```
(define-state complex
    :name "pirate-b-unimog-bump-back"
    :tree (anim-node-clip "pirate-b-unimog-bump-back")
    :flags (anim-state-flag no-adjust-to-ground)
)
```

The following complex state shows how blend nodes can be cascaded into arbitrarily deep blend trees:

```
(define-state complex
    :name "move-b-to-f"
    :tree
        (anim-node-lerp
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-f")
                    (anim-node-clip "move-f-look-lr")
                )
```

图 12.57。与示例状态“move-b-to-f”对应的混合树。

```
            (anim-node-clip "move-f-look-ud")
        )
        (anim-node-additive
            (anim-node-additive
                (anim-node-clip "move-b")
                (anim-node-clip "move-b-look-lr")
            )
            (anim-node-clip "move-b-look-ud")
        )
    )
)
```

这对应于图 12.57 所示的树。

快速迭代

顽皮狗的动画团队借助四个重要工具实现了快速迭代：

1. 游戏内动画查看器允许角色在游戏中生成，并通过游戏内菜单控制其动画。
2. 一个简单的命令行工具允许动画脚本在运行中重新编译并加载到正在运行的游戏中。为了调整角色的动画，用户可以修改包含动画状态规范的文本文件，快速重新加载动画状态，并立即看到更改对游戏中动画角色的影响。
3. 引擎会持续追踪每个角色在游戏最后几秒内的所有状态转换。这让我们可以暂停游戏，然后倒回动画来仔细检查，并调试游戏中发现的问题。

Figure 12.57. Blend tree corresponding to the example state “move-b-to-f.”

```
            (anim-node-clip "move-f-look-ud")
        )
        (anim-node-additive
            (anim-node-additive
                (anim-node-clip "move-b")
                (anim-node-clip "move-b-look-lr")
            )
            (anim-node-clip "move-b-look-ud")
        )
    )
)
```

This corresponds to the tree shown in Figure 12.57.

Rapid Iteration

Naughty Dog’s animation team achieves rapid iteration with the help of four important tools:

1. An in-game animation viewer allows a character to be spawned into the game and its animations controlled via an in-game menu.
2. A simple command-line tool allows animation scripts to be recompiled and reloaded into the running game on the fly. To tweak a character’s animations, the user can make changes to the text file containing the animation state specifications, quickly reload the animation states and immediately see the effects of his or her changes on an animating character in the game.
3. The engine continually keeps track of all state transitions performed by each character during the last few seconds of gameplay. This allows us to pause the game and then literally rewind the animations to scrutinize them and debug problems that are noticed while playing.

4. 顽皮狗引擎还提供了大量“实时更新”工具。例如，动画师可以在 Maya 中调整动画，并在游戏中看到几乎即时的更新。

12.10.3.2 示例：虚幻引擎 4

虚幻引擎 4 (UE4) 为用户提供了五种用于处理骨骼动画和骨骼网格的工具：骨骼编辑器、骨骼网格编辑器、动画编辑器、动画蓝图编辑器和物理编辑器。

- 骨架编辑器本质上是一个绑定工具。它允许用户查看和修改骨架，为关节添加插槽，并测试骨架的运动。在其他引擎中，插槽有时被称为附着点（参见第 12.11.1 节）。
- 骨架网格编辑器允许用户编辑蒙皮到动画骨架的网格的属性。
- 动画编辑器允许用户导入、创建和管理动画资源。在此编辑器中，可以调整动画剪辑（UE4 中称为“序列”）的压缩和时间。剪辑可以组合到预定义的混合空间中，并且可以通过创建动画蒙太奇来定义游戏内的过场动画。
- 动画蓝图编辑器允许用户运用虚幻引擎蓝图可视化脚本系统的强大功能来控制角色的动画状态机。该编辑器如图 12.58 所示。
- 物理编辑器允许用户对刚体的层次结构进行建模，当布娃娃物理处于活动状态时，这些刚体可以驱动骨骼的运动。

关于虚幻引擎动画工具的完整讨论超出了我们的范围，但您可以通过在线搜索“虚幻骨骼网格动画系统”来了解更多信息。

12.10.4 过渡

为了创建高质量的动画角色，我们必须精心管理动作状态机中状态之间的转换，以确保动画之间的衔接不会显得突兀粗糙。大多数现代动画引擎都提供了一种数据驱动的机制，用于精确指定如何处理转换。在本节中，我们将探讨这种机制的工作原理。

4. The Naughty Dog engine also offers a host of “live update” tools. For example, animators can tweak their animations in Maya and see them update virtually instantaneously in the game.

12.10.3.2 Example: Unreal Engine 4

Unreal Engine 4 (UE4) provides its users with five tools for working with skeletal animations and skeletal meshes: The Skeleton Editor, the Skeletal Mesh Editor, the Animation Editor, the Animation Blueprint Editor, and the Physics Editor.

- The Skeleton Editor is essentially a rigging tool. It allows users to view and modify skeletons, add *sockets* to joints, and test out the movement of the skeleton. A socket is sometimes called an *attach point* in other engines (see Section 12.11.1).
- The Skeletal Mesh Editor allows users to edit properties of the meshes that are skinned to animating skeletons.
- The Animation Editor allows users to import, create and manage animation assets. In this editor, the compression and timing of animation clips (which UE4 calls Sequences) can be adjusted. Clips can be combined into predefined Blend Spaces, and in-game cinematics can be defined by creating Animation Montages.
- The Animation Blueprint Editor allows users to apply the power of Unreal Engine’s Blueprints visual scripting system to controlling characters’ animation state machines. This editor is depicted in Figure 12.58.
- The Physics Editor allows users to model a hierarchy of rigid bodies that drive the skeleton’s motion when ragdoll physics is active.

A complete discussion of Unreal Engine’s animation tools is beyond our scope here, but you can read more about it by searching for “Unreal Skeletal Mesh Animation System” online.

12.10.4 Transitions

To create a high-quality animating character, we must carefully manage the *transitions* between states in the action state machine to ensure that the splices between animations do not have a jarring and unpolished appearance. Most modern animation engines provide a data-driven mechanism for specifying exactly how transitions should be handled. In this section, we’ll explore how this mechanism works.

图 12.58 虚幻引擎 4 动画蓝图编辑器。（参见彩色图 XXVI。）

12.10.4.1 转换种类

管理状态之间的过渡有很多不同的方法。如果我们知道源状态的最终姿势与目标状态的第一个姿势完全匹配，我们可以简单地从一个状态“弹出”到另一个状态。否则，我们可以从一个状态交叉淡入淡出到下一个状态。在状态间过渡时，交叉淡入淡出并不总是合适的选择。例如，交叉淡入淡出无法产生从躺在地上到直立的真实过渡。对于这种状态过渡，我们需要一个或多个自定义动画。这种过渡通常通过在状态机中引入特殊的过渡状态来实现。这些状态仅用于从一个状态转换到另一个状态——它们从不用作稳态节点。但由于它们是成熟的状态，因此它们可以由任意复杂的混合树组成。这在编写自定义动画过渡时提供了最大的灵活性。

12.10.4.2 过渡参数

在描述两个状态之间的特定转换时，我们通常需要指定各种参数，以精确控制转换的发生方式。这些参数包括但不限于以下内容。

- 源状态和目标状态。此转换适用于哪个（些）状态？

Figure 12.58. The Unreal Engine 4 animation Blueprints editor. (See Color Plate XXVI.)

12.10.4.1 Kinds of Transitions

There are many different ways to manage the transition between states. If we know that the final pose of the source state exactly matches the first pose of the destination state, we can simply “pop” from one state to another. Otherwise, we can cross-fade from one state to the next. Cross-fading is not always a suitable choice when transitioning from state to state. For example, there is no way that a cross-fade can produce a realistic transition from lying on the ground to standing upright. For this kind of state transition, we need one or more custom animations. This kind of transition is often implemented by introducing special *transitional states* into the state machine. These states are intended for use only when going from one state to another—they are never used as a steady-state node. But because they are full-fledged states, they can be comprised of arbitrarily complex blend trees. This provides maximum flexibility when authoring custom-animated transitions.

12.10.4.2 Transition Parameters

When describing a particular transition between two states, we generally need to specify various parameters, controlling exactly how the transition will occur. These include but are not limited to the following.

- *Source and destination states*. To which state(s) does this transition apply?

• 过渡类型。过渡是立即过渡、淡入淡出过渡还是通过过渡状态执行?

• 持续时间。对于交叉淡入淡出过渡，我们需要指定交叉淡入淡出需要多长时间。

• 缓入/缓出曲线类型。在交叉淡入淡出过渡中，我们可能希望指定缓入/缓出曲线的类型，以便在淡入淡出期间改变混合因子。

• 过渡窗口。某些过渡仅在源动画位于其本地时间轴的指定窗口内时才有效。例如，从拳击动画到冲击反应的过渡可能仅在手臂挥动的后半段才有意义。如果在挥动的前半段尝试执行过渡，则该过渡将被禁止（或者可能会选择其他过渡）。

12.10.4.3 过渡矩阵

指定状态之间的转换可能颇具挑战性，因为可能的转换数量通常非常庞大。在一个具有 n 个状态的状态机中，最坏情况下可能的转换数量为 n^2。我们可以想象一个二维方阵，其中所有可能的状态都列在纵轴和横轴上。这样的表格可以用来指定从纵轴上的任意状态到横轴上的任意其他状态的所有可能转换。

在实际游戏中，这个转换矩阵通常非常稀疏，因为并非所有状态间转换都是可行的。例如，通常不允许从死亡状态转换到任何其他状态。同样，从驾驶状态到游泳状态可能也根本不可能（除非经历至少一个导致角色跳出车辆的中间状态）。表中唯一转换的数量甚至可能比状态间有效转换的数量还要少得多。这是因为我们通常可以在许多不同的状态对之间重用单个转换规范。

12.10.4.4 实现转换矩阵

实现转换矩阵的方法有很多种。我们可以使用电子表格应用程序将所有转换以矩阵形式列出，或者允许在编写动作状态的同一文本文件中编写转换。如果提供了用于状态编辑的图形用户界面，也可以将转换添加到该 GUI 中。在接下来的章节中，我们将

- *Transition type*. Is the transition immediate, cross-faded or performed via a transitional state?
- *Duration*. For cross-faded transitions, we need to specify how long the cross-fade should take.
- *Ease-in/ease-out curve type*. In a cross-faded transition, we may wish to specify the type of ease-in/ease-out curve to use to vary the blend factor during the fade.
- *Transition window*. Certain transitions can only be taken when the source animation is within a specified window of its local timeline. For example, a transition from a punch animation to an impact reaction might only make sense when the arm is in the second half of its swing. If an attempt to perform the transition is made during the first half of the swing, the transition would be disallowed (or a different transition might be selected instead).

12.10.4.3 The Transition Matrix

Specifying transitions between states can be challenging, because the number of possible transitions is usually very large. In a state machine with n states, the worst-case number of possible transitions is n^2. We can imagine a two-dimensional square matrix with every possible state listed along both the vertical and horizontal axes. Such a table can be used to specify all of the possible transitions from any state along the vertical axis to any other state along the horizontal axis.

In a real game, this *transition matrix* is usually quite sparse, because not all state-to-state transitions are possible. For example, transitions are usually disallowed from a death state to any other state. Likewise, there is probably no way to go from a driving state to a swimming state (without going through at least one intermediate state that causes the character to jump out of his vehicle). The number of unique transitions in the table may be significantly less even than the number of valid transitions between states. This is because we can often reuse a single transition specification between many different pairs of states.

12.10.4.4 Implementing a Transition Matrix

There are all sorts of ways to implement a transition matrix. We could use a spreadsheet application to tabulate all the transitions in matrix form, or we might permit transitions to be authored in the same text file used to author our action states. If a graphical user interface is provided for state editing, transitions could be added to this GUI as well. In the following sections, we'll

简要了解一下来自真实游戏引擎的几个转换矩阵实现。

示例：《荣誉勋章：太平洋突袭》中的通配符转换

在《荣誉勋章：太平洋突袭》（MOHPA）中，我们充分利用了转换矩阵的稀疏性，支持通配符转换规范。对于每个转换规范，源状态和目标状态的名称都可以包含星号 (*) 作为通配符。这使我们能够指定从任意状态到任意其他状态的单个默认转换（通过语法 from="*" to="*"），然后轻松地针对所有状态类别优化此全局默认转换。必要时，这种优化可以一直延伸到特定状态对之间的自定义转换。MOHPA 转换矩阵如下所示：

```
<transitions>
    <!-- global default -->
    <trans from="*" to="*"
           type=frozen duration=0.2>

    <!-- default for any walk to any run -->
    <trans from="walk*" to="run*"
           type=smooth
           duration=0.15>

    <!-- special handling from any prone to any getting-up
      -- action (only valid from 2 sec to 7.5 sec on the
      -- local timeline) -->
    <trans from="*prone" to="*get-up"
           type=smooth
           duration=0.1
           window-start=2.0
           window-end=7.5>
    ...
</transitions>
```

示例：《神秘海域》中的一级转换

在某些动画引擎中，高级游戏代码会通过明确指定目标状态来请求从当前状态到新状态的转换。这种方法的问题在于，调用代码必须非常了解状态的名称，以及在特定状态下哪些转换是有效的。

take a brief look at a few transition matrix implementations from real game engines.

Example: Wildcarded Transitions in Medal of Honor: Pacific Assault

On *Medal of Honor: Pacific Assault* (*MOHPA*), we used the sparseness of the transition matrix to our advantage by supporting wildcarded transition specifications. For each transition specification, the names of both the source and destination states could contain asterisks (*) as a wildcard character. This allowed us to specify a single default transition from any state to any other state (via the syntax `from="*" to="*"`) and then refine this global default easily for entire categories of states. The refinement could be taken all the way down to custom transitions between specific state pairs when necessary. The *MOHPA* transition matrix looked something like this:

```
<transitions>
    <!-- global default -->
    <trans from="*" to="*"
           type=frozen duration=0.2>

    <!-- default for any walk to any run -->
    <trans from="walk*" to="run*"
           type=smooth
           duration=0.15>

    <!-- special handling from any prone to any getting-up
      -- action (only valid from 2 sec to 7.5 sec on the
      -- local timeline) -->
    <trans from="*prone" to="*get-up"
           type=smooth
           duration=0.1
           window-start=2.0
           window-end=7.5>
    ...
</transitions>
```

Example: First-Class Transitions in Uncharted

In some animation engines, high-level game code requests transitions from the current state to a new state by naming the destination state explicitly. The problem with this approach is that the calling code must have intimate knowledge of the names of the states and of which transitions are valid when in a particular state.

在顽皮狗的引擎中，这个问题通过将状态转换从次要的实现细节转变为首要的实体来克服。每个状态都提供了一个到其他状态的有效转换列表，并且每个转换都被赋予一个唯一的名称。转换的名称是标准化的，以使每个转换的效果可预测。例如，如果一个转换被称为“行走”，那么无论当前状态是什么，它总是从当前状态转到某种行走状态。每当高级动画控制代码想要从状态 A 转换到状态 B 时，它都会通过名称请求转换（而不是明确请求目标状态）。如果可以找到这样的转换并且有效，则采用它；否则，请求失败。

以下示例状态定义了四个转换，分别为“reload”、“step-left”、“step-right”和“fire”。(transition-group ...) 命令调用了一组先前定义的转换；当同一组转换用于多个状态时，它非常有用。(transition-end ...) 命令指定在到达状态本地时间线末尾时执行的转换（如果在此之前没有执行过其他转换）。

```
(define-state complex
    :name "s_turret-idle"
    :tree (aim-tree
            (anim-node-clip "turret-aim-all--base")
            "turret-aim-all--left-right"
            "turret-aim-all--up-down"
        )
    :transitions (
        (transition "reload" "s_turret-reload"
            (range - -) :fade-time 0.2)

        (transition "step-left" "s_turret-step-left"
            (range - -) :fade-time 0.2)

        (transition "step-right" "s_turret-step-right"
            (range - -) :fade-time 0.2)

        (transition "fire" "s_turret-fire"
            (range - -) :fade-time 0.1)

        (transition-group "combat-gunout-idle^move")

        (transition-end "s_turret-idle")
    )
)
```

这种方法的优点可能一开始很难看出。它的主要

In Naughty Dog’s engine, this problem is overcome by turning state transitions from secondary implementation details into first-class entities. Each state provides a list of valid transitions to other states, and each transition is given a unique name. The names of the transitions are standardized in order to make the *effect* of each transition predictable. For example, if a transition is called “walk,” then it *always* goes from the current state to a walking state of some kind, no matter what the current state is. Whenever the high-level animation control code wants to transition from state A to state B, it asks for a transition by name (rather than requesting the destination state explicitly). If such a transition can be found and is valid, it is taken; otherwise, the request fails.

The following example state defines four transitions named “reload,” “step-left,” “step-right” and “fire.” The `(transition-group ...)` line invokes a previously defined group of transitions; it is useful when the same set of transitions is to be used in multiple states. The `(transition-end ...)` command specifies a transition that is taken upon reaching the end of the state’s local timeline if no other transition has been taken before then.

```
(define-state complex
    :name "s_turret-idle"
    :tree (aim-tree
            (anim-node-clip "turret-aim-all--base")
            "turret-aim-all--left-right"
            "turret-aim-all--up-down"
        )
    :transitions (
        (transition "reload" "s_turret-reload"
            (range - -) :fade-time 0.2)

        (transition "step-left" "s_turret-step-left"
            (range - -) :fade-time 0.2)

        (transition "step-right" "s_turret-step-right"
            (range - -) :fade-time 0.2)

        (transition "fire" "s_turret-fire"
            (range - -) :fade-time 0.1)

        (transition-group "combat-gunout-idle^move")

        (transition-end "s_turret-idle")
    )
)
```

The beauty of this approach may be difficult to see at first. Its primary

目的是允许以数据驱动的方式修改转换和状态，在许多情况下无需更改 C++ 源代码。这种灵活性是通过屏蔽动画控制代码以获取状态图结构信息来实现的。例如，假设我们有十种不同的行走状态（正常、害怕、蹲伏、受伤等等）。所有这些状态都可以转换为跳跃状态，但不同类型的行走可能需要不同的跳跃动画（例如，正常跳跃、害怕跳跃、蹲伏跳跃、受伤跳跃等等）。对于这十种行走状态中的每一种，我们都定义一个简称为“跳跃”的转换。首先，我们可以将所有这些转换都指向一个通用的“跳跃”状态，以便让一切正常运行。之后，我们可以对其中一些转换进行微调，使它们指向自定义的跳跃状态。我们甚至可以在某些“行走”状态及其对应的“跳跃”状态之间引入过渡状态。可以对状态图的结构和转换的参数进行各种更改，而不会影响 C++ 源代码 - 只要转换的名称不变。

12.10.5 控制参数

从软件工程的角度来看，协调复杂动画角色的所有混合权重、播放速率和其他控制参数可能极具挑战性。不同的混合权重对角色动画的方式有不同的影响。例如，一个权重可能控制角色的移动方向，而其他权重则控制其移动速度、武器的水平和垂直瞄准、头部/眼睛的注视方向等等。我们需要某种方式将所有这些混合权重暴露给负责控制它们的代码。

在平面加权平均架构中，我们有一个平面列表，其中包含所有可能在角色上播放的动画剪辑。每个剪辑状态都有一个混合权重、一个播放速率以及可能的其他控制参数。控制角色的代码必须按名称查找各个剪辑状态，并相应地调整每个剪辑状态的混合权重。这使得界面变得简单，但却将控制混合权重的大部分责任转移到了角色控制系统。例如，要调整角色奔跑的方向，角色控制代码必须知道“奔跑”动作由一组动画剪辑组成，这些剪辑的名称类似于“左移”、“向前奔跑”、“右移”和“向后奔跑”。它必须按名称查找这些剪辑状态，并手动控制所有四个混合权重，才能实现特定角度的奔跑动画。毋庸置疑，以如此细粒度的方式控制动画参数可能非常繁琐，并且可能导致难以实现。

purpose is to allow transitions and states to be modified in a data-driven manner, without requiring changes to the C++ source code in many cases. This degree of flexibility is accomplished by shielding the animation control code from knowledge of the structure of the state graph. For example, let's say that we have ten different walking states (normal, scared, crouched, injured and so on). All of them can transition into a jumping state, but different kinds of walks might require different jump animations (e.g., normal jump, scared jump, jump from crouch, injured jump, etc.). For each of the ten walking states, we define a transition simply called "jump." At first, we can point all of these transitions to a single generic "jump" state, just to get things up and running. Later, we can fine-tune some of these transitions so that they point to custom jump states. We can even introduce transitional states between some of the "walk" states and their corresponding "jump" states. All sorts of changes can be made to the structure of the state graph and the parameters of the transitions without affecting the C++ source code—as long as the *names* of the transitions don't change.

12.10.5 Control Parameters

From a software engineering perspective, it can be challenging to orchestrate all of the blend weights, playback rates and other control parameters of a complex animating character. Different blend weights have different effects on the way the character animates. For example, one weight might control the character's movement direction, while others control its movement speed, horizontal and vertical weapon aim, head/eye look direction and so on. We need some way of exposing all of these blend weights to the code that is responsible for controlling them.

In a flat weighted average architecture, we have a flat list of all the animation clips that could possibly be played on the character. Each clip state has a blend weight, a playback rate and possibly other control parameters. The code that controls the character must look up individual clip states by name and adjust each one's blend weight appropriately. This makes for a simple interface, but it shifts most of the responsibility for controlling the blend weights to the character control system. For example, to adjust the direction in which a character is running, the character control code must know that the "run" action is comprised of a group of animation clips, named something like "StrafeLeft," "RunForward," "StrafeRight" and "RunBackward." It must look up these clip states by name and manually control all four blend weights in order to achieve a particular angled run animation. Needless to say, controlling animation parameters in such a fine-grained way can be tedious and can lead to difficult-

理解源代码。

在混合树中，出现了一系列不同的问题。得益于树形结构，剪辑可以自然地分组到功能单元中。自定义树节点可以封装复杂的角色动作。这些优点都优于平面加权平均方法。然而，控制参数隐藏在树中。想要控制头部和眼睛水平注视方向的代码需要事先了解混合树的结构，以便找到树中合适的节点来控制它们的参数。

不同的动画引擎以不同的方式解决这些问题。以下是一些示例：

- 节点搜索。一些引擎为高级代码提供了一种在树中查找混合节点的方法。例如，可以为树中的相关节点赋予特殊名称，例如将控制武器水平瞄准的节点命名为“HorizAim”。控制代码只需在树中搜索特定名称的节点即可；如果找到，我们就能知道调整其混合权重会产生什么效果。

- 命名变量。某些引擎允许为各个控制参数分配名称。控制代码可以通过名称查找控制参数，从而调整其值。

- 控制结构。在其他引擎中，一个简单的数据结构（例如浮点值数组或 C 结构体）包含整个角色的所有控制参数。混合树中的节点连接到特定的控制参数，可以通过硬编码使用某些结构体成员，或通过名称或索引查找参数来实现。

当然，还有很多其他的选择。每个动画引擎解决这个问题的方式略有不同，但最终效果大致相同。

12.11 约束

我们已经了解了如何使用动作状态机来指定复杂的混合树，以及如何使用过渡矩阵来控制状态之间的过渡。角色动画控制的另一个重要方面是以各种方式约束场景中角色和/或物体的移动。例如，我们可能希望约束一件武器，使其始终看起来像是在携带它的角色手中。

to-understand source code.

In a blend tree, a different set of problems arise. Thanks to the tree structure, the clips are grouped naturally into functional units. Custom tree nodes can encapsulate complex character motions. These are both helpful advantages over the flat weighted average approach. However, the control parameters are buried within the tree. Code that wishes to control the horizontal look-at direction of the head and eyes needs a priori knowledge of the structure of the blend tree so that it can find the appropriate nodes in the tree in order to control their parameters.

Different animation engines solve these problems in different ways. Here are some examples:

- *Node search*. Some engines provide a way for higher-level code to *find* blend nodes in the tree. For example, relevant nodes in the tree can be given special names, such as “HorizAim” for the node that controls horizontal weapon aiming. The control code can simply search the tree for a node of a particular name; if one is found, then we know what effect adjusting its blend weight will have.
- *Named variables*. Some engines allow names to be assigned to the individual control parameters. The controlling code can look up a control parameter by name in order to adjust its value.
- *Control structure*. In other engines, a simple data structure, such as an array of floating-point values or a C struct, contains all of the control parameters for the entire character. The nodes in the blend tree(s) are connected to particular control parameters, either by being hard-coded to use certain struct members or by looking up the parameters by name or index.

Of course, there are many other alternatives as well. Every animation engine tackles this problem in a slightly different way, but the net effect is always roughly the same.

12.11 Constraints

We’ve seen how action state machines can be used to specify complex blend trees and how a transition matrix can be used to control how transitions between states should work. Another important aspect of character animation control is to constrain the movement of the characters and/or objects in the scene in various ways. For example, we might want to constrain a weapon so that it always appears to be in the hand of the character who is carrying it.

图 12.59. 附件显示了父级的运动如何自动引起子级的运动，但反之则不然。

我们可能希望约束两个角色，使它们在握手时能够正确对齐。角色的脚通常被约束为与地板对齐，而其手则可能被约束为与梯子的横档或车辆的方向盘对齐。在本节中，我们将简要介绍在典型的动画系统中如何处理这些约束。

12.11.1 附件

几乎所有现代游戏引擎都允许对象相互连接。最简单的对象到对象的连接是指约束对象 A 骨架中特定关节 JA 的位置和/或方向，使其与对象 B 骨架中的关节 JB 重合。连接通常是一种父子关系。当父对象的骨架移动时，子对象会进行调整以满足约束条件。然而，当子对象移动时，父对象的骨架通常不受影响。如图 12.59 所示。

有时，在父关节和子关节之间引入偏移会很方便。例如，当将枪放到角色手中时，我们可以约束枪的“Grip”关节，使其与角色的“RightWrist”关节重合。然而，这可能无法使枪与手正确对齐。解决这个问题的一个方法是在两个骨架中的一个骨架中引入一个特殊关节。例如，我们可以在角色骨架中添加一个“RightGun”关节，使其成为“RightWrist”关节的子关节，并将其定位，以便当枪的“Grip”关节约束到它时，枪看起来就像角色自然地握着一样。然而，这种方法的问题在于，它增加了

Figure 12.59. An attachment, showing how movement of the parent automatically produces movement of the child but not vice versa.

We might wish to constrain two characters so that they line up properly when shaking hands. A character's feet are often constrained so that they line up with the floor, and its hands might be constrained to line up with the rungs on a ladder or the steering wheel of a vehicle. In this section, we'll take a brief look at how these constraints are handled in a typical animation system.

12.11.1 Attachments

Virtually all modern game engines permit objects to be attached to one another. At its simplest, object-to-object attachment involves constraining the position and/or orientation of a particular joint J_A within the skeleton of object A so that it coincides with a joint J_B in the skeleton of object B. An attachment is usually a parent-child relationship. When the parent's skeleton moves, the child object is adjusted to satisfy the constraint. However, when the child moves, the parent's skeleton is usually not affected. This is illustrated in Figure 12.59.

Sometimes it can be convenient to introduce an *offset* between the parent joint and the child joint. For example, when placing a gun into a character's hand, we could constrain the "Grip" joint of the gun so that it coincides with the "RightWrist" joint of the character. However, this might not produce the correct alignment of the gun with the hand. One solution to this problem is to introduce a special joint into one of the two skeletons. For example, we could add a "RightGun" joint to the character's skeleton, make it a child of the "RightWrist" joint, and position it so that when the "Grip" joint of the gun is constrained to it, the gun looks like it is being held naturally by the character. The problem with this approach, however, is that it increases the

图 12.60。附着点就像父级和子级之间的额外关节。

骨架中的关节数量。每个关节都会产生与动画混合和矩阵调色板计算相关的处理成本，以及存储其动画关键帧所需的内存成本。因此，添加新的关节通常不是一个可行的选择。

我们知道，为了附加目的而添加的额外关节不会影响角色的姿势——它只会在附加的父关节和子关节之间引入额外的变换。因此，我们真正想要的是一种标记某些关节的方法，以便它们可以被动画混合管线忽略，但仍可用于附加目的。这种特殊关节有时被称为 附加点 。它们如图 12.60 所示。

在 Maya 中，附着点的建模方式可能与常规关节或定位器类似，但许多游戏引擎会以更便捷的方式定义附着点。例如，它们可以作为动作状态机文本文件的一部分指定，或通过动画创作工具中的自定义 GUI 指定。这使得动画师可以专注于影响角色外观的关节，而附着点的控制权则方便地交给需要的人——游戏设计师和工程师。

12.11.2 对象间注册

随着每一款新游戏的推出，游戏角色与其环境之间的互动变得越来越复杂和微妙。因此，拥有一个能够在动画制作过程中使角色和物体相互对齐的系统至关重要。这样的系统可以用于游戏内过场动画和互动游戏元素。

想象一下，一位动画师正在使用 Maya 或其他动画工具，设置一个包含两个角色和一扇门的场景。两个角色握手，然后其中一个打开门，两人一起走了进去。动画师可以确保场景中的三个演员完美地排成一列。

Figure 12.60. An attach point acts like an extra joint between the parent and the child.

number of joints in the skeleton. Each joint has a processing cost associated with animation blending and matrix palette calculation and a memory cost for storing its animation keys. So adding new joints is often not a viable option.

We know that an additional joint added for attachment purposes will not contribute to the pose of the character—it merely introduces an additional transform between the parent and child joint in an attachment. What we really want, then, is a way to mark certain joints so that they can be ignored by the animation blending pipeline but can still be used for attachment purposes. Such special joints are sometimes called *attach points*. They are illustrated in Figure 12.60.

Attach points might be modeled in Maya just like regular joints or locators, although many game engines define attach points in a more convenient manner. For example, they might be specified as part of the action state machine text file or via a custom GUI within the animation authoring tool. This allows the animators to focus only on the joints that affect the look of the character, while the power to control attachments is put conveniently into the hands of the people who need it—the game designers and the engineers.

12.11.2 Interobject Registration

The interactions between game characters and their environments is growing ever more complex and nuanced with each new title. Hence, it is important to have a system that allows characters and objects to be aligned with one another when animating. Such a system can be used for in-game cinematics and interactive gameplay elements alike.

Imagine that an animator, working in Maya or some other animation tool, sets up a scene involving two characters and a door object. The two characters shake hands, and then one of them opens the door and they both walk through it. The animator can ensure that all three actors in the scene line up perfectly.

图 12.61.原始 Maya 场景包含三个演员和一个参考定位器。

图 12.62. 参考定位器被编码在每个演员的动画文件中。

然而，动画导出后会变成三个独立的片段，分别在游戏世界中的三个不同物体上播放。这两个角色在动画序列开始之前可能已经处于AI或玩家的控制之下。那么，我们如何确保这三个片段在游戏中播放时能够正确对齐呢？

12.11.2.1 参考定位器

一个好的解决方案是为所有三个动画剪辑引入一个公共的参考点。在 Maya 中，动画师可以将一个定位器（它只是一个 3D 变换，很像骨骼关节）放入场景中，放置在任何方便的位置。正如我们将看到的，它的位置和方向实际上无关紧要。定位器会以某种方式进行标记，以告知动画导出工具需要对其进行特殊处理。

导出三个动画剪辑时，这些工具会将引用定位器的位置和方向（以相对于每个角色的局部对象空间的坐标表示）存储到所有三个剪辑的数据文件中。之后，当在游戏中播放这三个剪辑时，动画引擎可以查找引用定位器在这三个剪辑中的相对位置和方向。然后，它可以变换三个对象的原点，使所有三个引用定位器在世界空间中重合。引用定位器的作用类似于附着点（第 12.11.1 节），实际上，可以将其实现为一个附着点。最终效果是，所有三个角色现在都彼此对齐，就像它们在原始 Maya 场景中对齐一样。

图 12.61 展示了如何在 Maya 场景中设置上述示例中的门和两个角色。如图 12.62 所示，引用定位器出现在每个导出的动画剪辑中（以该 Actor 的局部空间表示）。在游戏中，这些局部空间引用定位器会与固定的世界空间定位器对齐，以便重新对齐 Actor，如图 12.63 所示。

Figure 12.61. Original Maya scene containing three actors and a reference locator.

Figure 12.62. The reference locator is encoded in each actor's animation file.

However, when the animations are exported, they become three separate clips, to be played on three separate objects in the game world. The two characters might have been under AI or player control prior to the start of this animated sequence. How, then, can we ensure that the three objects line up correctly with one another when the three clips are played back in-game?

12.11.2.1 Reference Locators

One good solution is to introduce a common reference point into all three animation clips. In Maya, the animator can drop a *locator* (which is just a 3D transform, much like a skeletal joint) into the scene, placing it anywhere that seems convenient. Its location and orientation are actually irrelevant, as we'll see. The locator is tagged in some way to tell the animation export tools that it is to be treated specially.

When the three animation clips are exported, the tools store the position and orientation of the reference locator, expressed in coordinates that are relative to the local object space of each actor, into all three clip's data files. Later, when the three clips are played back in-game, the animation engine can look up the relative position and orientation of the reference locator in all three clips. It can then transform the origins of the three objects in such a way as to make all three reference locators coincide in world space. The reference locator acts much like an *attach point* (Section 12.11.1) and, in fact, could be implemented as one. The net effect—all three actors now line up with one another, exactly as they had been aligned in the original Maya scene.

Figure 12.61 illustrates how the door and the two characters from the above example might be set up in a Maya scene. As shown in Figure 12.62, the reference locator appears in each exported animation clip (expressed in that actor's local space). In-game, these local-space reference locators are aligned to a fixed world-space locator in order to realign the actors, as shown in Figure 12.63.

图 12.63. 在运行时，本地空间参考变换与世界空间参考定位器对齐，从而使演员正确排列。

12.11.2.2 查找世界空间参考位置

我们在这里忽略了一个重要的细节——谁来决定引用定位器在世界空间中的位置和方向？每个动画剪辑都会提供引用定位器在其 actor 坐标空间中的变换。但我们需要某种方法来定义该引用定位器在世界空间中的位置。

在我们这个关于门和两个角色握手的例子中，其中一个角色（门）在世界中是固定的。因此，一个可行的解决方案是向门询问参考定位器的位置，然后将两个角色与其对齐。实现此目的的命令可能类似于以下伪代码。

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB)
{
    // Find the world-space transform of the reference
    // locator as specified in the door's animation.
    Transform refLoc = getReferenceLocatorWs(door,
        "shake-hands-door");

    // Play the door's animation in-place. (It's
    // already in the correct place.)
    playAnimation("shake-hands-door", door);

    // Play the two characters' animations relative to
    // the world-space reference locator obtained from
    // the door.
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

Figure 12.63. At runtime, the local-space reference transforms are aligned to a world-space reference locator, causing the actors to line up properly.

12.11.2.2 Finding the World-Space Reference Location

We've glossed over one important detail here—who decides what the world-space position and orientation of the reference locator should be? Each animation clip provides the reference locator's transform in the coordinate space of *its actor*. But we need some way to define where that reference locator should be in world space.

In our example with the door and the two characters shaking hands, one of the actors is fixed in the world (the door). So one viable solution is to ask the door for the location of the reference locator and then align the two characters to it. The commands to accomplish this might look similar to the following pseudocode.

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB)
{
    // Find the world-space transform of the reference
    // locator as specified in the door's animation.
    Transform refLoc = getReferenceLocatorWs(door,
        "shake-hands-door");

    // Play the door's animation in-place. (It's
    // already in the correct place.)
    playAnimation("shake-hands-door", door);

    // Play the two characters' animations relative to
    // the world-space reference locator obtained from
    // the door.
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

另一种选择是独立于场景中的三个角色来定义参考定位器的世界空间变换。例如，我们可以使用我们的世界构建工具将参考定位器放置到世界中（参见第 15.3 节）。在这种情况下，上面的伪代码应该改为如下所示：

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB,
    Actor& refLocatorActor)
{
    // Find the world-space transform of the reference
    // locator by simply querying the transform of an
    // independent actor (presumably placed into the
    // world manually).
    Transform refLoc = getActorTransformWs(
        refLocatorActor);

    // Play all animations relative to the world-space
    // reference locator obtained above.
    playAnimationRelativeToReference("shake-hands-door",
        door, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

12.11.3 抓取和手部 IK

即使使用附件连接两个物体，我们有时也会发现游戏中的对齐效果并不完全一致。例如，一个角色可能用右手握着步枪，左手支撑着枪托。当角色用武器瞄准不同方向时，我们可能会注意到左手在某些瞄准角度下不再与枪托正确对齐。这种关节错位是由 LERP 混合引起的。即使相关关节在片段 A 和片段 B 中完美对齐，LERP 混合也不能保证在 A 和 B 混合在一起时这些关节也对齐。

解决这个问题的一个方法是使用逆运动学 (IK) 来校正左手的位置。基本方法是确定相关关节的期望目标位置。然后将 IK 应用于一小段关节链（通常是两、三或四个关节），从相关关节开始，然后

Another option is to define the world-space transform of the reference locator independently of the three actors in the scene. We could place the reference locator into the world using our world-building tool, for example (see Section 15.3). In this case, the pseudocode above should be changed to look something like this:

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB,
    Actor& refLocatorActor)
{
    // Find the world-space transform of the reference
    // locator by simply querying the transform of an
    // independent actor (presumably placed into the
    // world manually).
    Transform refLoc = getActorTransformWs(
        refLocatorActor);

    // Play all animations relative to the world-space
    // reference locator obtained above.
    playAnimationRelativeToReference("shake-hands-door",
        door, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);
    playAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

12.11.3 Grabbing and Hand IK

Even after using an attachment to connect two objects, we sometimes find that the alignment does not look exactly right in-game. For example, a character might be holding a rifle in her right hand, with her left hand supporting the stock. As the character aims the weapon in various directions, we may notice that the left hand no longer aligns properly with the stock at certain aim angles. This kind of joint misalignment is caused by LERP blending. Even if the joints in question are aligned perfectly in clip A and in clip B, LERP blending does not guarantee that those joints will be in alignment when A and B are blended together.

One solution to this problem is to use *inverse kinematics* (IK) to correct the position of the left hand. The basic approach is to determine the desired target position for the joint in question. IK is then applied to a short chain of joints (usually two, three or four joints), starting with the joint in question and

沿着层级向上，依次为父关节、祖父关节等等。我们试图校正其位置的关节称为末端执行器。IK 解算器会调整末端执行器父关节的方向，以使末端执行器尽可能靠近目标。

IK 系统的 API 通常采用请求的形式，用于在特定关节链上启用或禁用 IK，并指定所需的目标点。实际的 IK 计算通常由低级动画管线内部完成。这使得它能够在适当的时间进行计算——即在计算中间局部和全局骨骼姿势之后，但在最终矩阵调色板计算之前。

一些动画引擎允许预先定义 IK 链。例如，我们可能为左臂定义一条 IK 链，为右臂定义一条 IK 链，为两条腿定义两条。为了便于本例说明，我们假设某个 IK 链由其末端执行器关节的名称来标识。（其他引擎可能使用索引、句柄或其他唯一标识符，但概念相同。）启用 IK 计算的函数可能如下所示：

```
void enableIkChain(Actor& actor,
                   const char* endEffectorJointName,
                   const Vector3& targetLocationWs);
```

禁用 IK 链的函数可能如下所示：

```
void disableIkChain(Actor& actor,
                    const char* endEffectorJointName);
```

IK 的启用和禁用频率通常相对较低，但世界空间目标位置必须每帧都保持更新（如果目标正在移动）。因此，低级动画管线始终提供某种机制来更新活动的 IK 目标点。例如，管线可能允许我们多次调用 enableIkChain()。第一次调用时，IK 链会被启用，并设置其目标点。所有后续调用都只会更新目标点。保持 IK 目标最新的另一种方法是将它们链接到游戏中的动态对象。例如，IK 目标可以指定为刚体游戏对象的句柄，或动画对象内的关节。

当关节已经相当接近其目标时，IK 非常适合对关节对齐进行微调。当关节的期望位置与其实际位置之间的误差较大时，它的效果就差强人意了。另请注意，大多数 IK 算法仅求解关节的位置。您可以

progressing up the hierarchy to its parent, grandparent and so on. The joint whose position we are trying to correct is known as the *end effector*. The IK solver adjusts the orientations of the end effector's parent joint(s) in order to get the end effector as close as possible to the target.

The API for an IK system usually takes the form of a request to enable or disable IK on a particular chain of joints, plus a specification of the desired target point. The actual IK calculation is usually done internally by the low-level animation pipeline. This allows it to do the calculation at the proper time—namely, after intermediate local and global skeletal poses have been calculated but before the final matrix palette calculation.

Some animation engines allow IK chains to be defined a priori. For example, we might define one IK chain for the left arm, one for the right arm and two for the two legs. Let's assume for the purposes of this example that a particular IK chain is identified by the name of its end-effector joint. (Other engines might use an index or handle or some other unique identifier, but the concept remains the same.) The function to enable an IK calculation might look something like this:

```
void enableIkChain(Actor& actor,
                   const char* endEffectorJointName,
                   const Vector3& targetLocationWs);
```

and the function to disable an IK chain might look like this:

```
void disableIkChain(Actor& actor,
                    const char* endEffectorJointName);
```

IK is usually enabled and disabled relatively infrequently, but the world-space target location must be kept up-to-date every frame (if the target is moving). Therefore, the low-level animation pipeline always provides some mechanism for updating an active IK target point. For example, the pipeline might allow us to call `enableIkChain()` multiple times. The first time it is called, the IK chain is enabled, and its target point is set. All subsequent calls simply update the target point. Another way to keep IK targets up-to-date is to link them to dynamic objects in the game. For example, an IK target might be specified as a handle to a rigid game object, or a joint within an animated object.

IK is well-suited to making minor corrections to joint alignment when the joint is already reasonably close to its target. It does not work nearly as well when the error between a joint's desired location and its actual location is large. Note also that most IK algorithms solve only for the *position* of a joint. You may

图 12.64。在动画创作包中，角色在空间中向前移动，双脚看起来像是落地的。图片由顽皮狗公司提供。（《神秘海域：德雷克的宝藏》© 2007/® SIE。由顽皮狗创作和开发。）

需要编写额外的代码来确保末端执行器的方向也与其目标正确对齐。反向运动学并非万能药，而且可能会显著降低性能。因此，请务必谨慎使用。

12.11.4 运动提取和脚 IK

在游戏中，我们通常希望角色的运动动画看起来逼真且“接地气”。影响运动动画真实感的最大因素之一是脚部是否在地面上滑动。脚部滑动可以通过多种方式解决，其中最常见的是运动提取和脚部 IK。

12.11.4.1 运动提取

想象一下如何为一个沿直线向前行走的角色制作动画。在 Maya（或他或她选择的动画软件）中，动画师让角色向前迈出一整步，先是左脚，然后是右脚。生成的动画剪辑称为运动循环，因为它会无限循环，只要角色在游戏中向前行走。动画师会注意确保角色的脚看起来像是接地的，并且在移动时不会滑动。角色从第 0 帧的初始位置移动到循环结束时的新位置。如图 12.64 所示。

请注意，角色的局部空间原点在整个行走过程中保持不变。实际上，角色在向前迈步时“将其原点抛在身后”。现在想象一下循环播放此动画。我们

Figure 12.64. In the animation authoring package, the character moves forward in space, and its feet appear grounded. Image courtesy of Naughty Dog, Inc. (UNCHARTED: Drake's Fortune © 2007/® SIE. Created and developed by Naughty Dog.)

need to write additional code to ensure that the *orientation* of the end effector aligns properly with its target as well. IK is not a cure-all, and it may have significant performance costs. So always use it judiciously.

12.11.4 Motion Extraction and Foot IK

In games, we usually want the locomotion animations of our characters to look realistic and "grounded." One of the biggest factors contributing to the realism of a locomotion animation is whether or not the feet slide around on the ground. *Foot sliding* can be overcome in a number of ways, the most common of which are *motion extraction* and *foot IK*.

12.11.4.1 Motion Extraction

Let's imagine how we'd animate a character walking forward in a straight line. In Maya (or his or her animation package of choice), the animator makes the character take one complete step forward, first with the left foot and then with the right foot. The resulting animation clip is known as a *locomotion cycle*, because it is intended to be looped indefinitely, for as long as the character is walking forward in-game. The animator takes care to ensure that the feet of the character appear grounded and don't slide as it moves. The character moves from its initial location on frame 0 to a new location at the end of the cycle. This is shown in Figure 12.64.

Notice that the local-space origin of the character remains fixed during the entire walk cycle. In effect, the character is "leaving his origin behind him" as he takes his step forward. Now imagine playing this animation as a loop. We

图 12.65。将根关节的向前运动归零后的行走循环。图片由顽皮狗公司提供。（《神秘海域：德雷克的宝藏》© 2007/® SIE。由顽皮狗创作和开发。）

会看到角色向前迈出整整一步，然后弹回到动画第一帧的位置。显然这在游戏中是行不通的。

为了实现这一点，我们需要移除角色的向前运动，以便其局部空间原点始终大致位于角色的质心下方。我们可以通过将角色骨架根关节的前向平移归零来实现这一点。最终的动画剪辑将使角色看起来像是在“月球漫步”，如图 12.65 所示。

为了让双脚看起来像在原始 Maya 场景中那样“贴”在地面上，我们需要角色每帧向前移动适当的距离。我们可以查看角色移动的距离，除以他到达目的地所用的时间，从而得到他的平均移动速度。但是，角色行走时前进的速度并非恒定不变。这在角色跛行时尤其明显（受伤的腿快速向前移动，然后“好”的腿移动得更慢），但所有看起来自然的步行周期都是如此。

因此，在将根关节的向前运动清零之前，我们首先将动画数据保存在一个特殊的“提取运动”通道中。这些数据可以在游戏中用于将角色的局部空间原点向前移动，移动量与根关节在 Maya 中每帧移动的量完全相同。最终结果是，角色将完全按照预设向前行走，但现在他的局部空间原点也随之移动，使动画能够正常循环。如图 12.66 所示。

如果角色在动画中向前移动 4 英尺，并且动画

Figure 12.65. Walk cycle after zeroing out the root joint's forward motion. Image courtesy of Naughty Dog, Inc. (UNCHARTED: Drake's Fortune © 2007/® SIE. Created and developed by Naughty Dog.)

would see the character take one complete step forward, and then pop back to where he was on the first frame of the animation. Clearly this won't work in-game.

To make this work, we need to remove the forward motion of the character, so that his local-space origin remains roughly under the center of mass of the character at all times. We could do this by zeroing out the forward translation of the root joint of the character's skeleton. The resulting animation clip would make the character look like he's "moonwalking," as shown in Figure 12.65.

In order to get the feet to appear to "stick" to the ground the way they did in the original Maya scene, we need the character to move forward by just the right amount each frame. We could look at the distance the character moved, divide by the amount of time it took for him to get there, and hence find his average movement speed. But a character's forward speed is not constant when walking. This is especially evident when a character is limping (quick forward motion on the injured leg, followed by slower motion on the "good" leg), but it is true for all natural-looking walk cycles.

Therefore, before we zero out the forward motion of the root joint, we first save the animation data in a special "extracted motion" channel. This data can be used in-game to move the local-space origin of the character forward by the exact amount that the root joint had moved in Maya each frame. The net result is that the character will walk forward exactly as he was authored, but now his local-space origin comes along for the ride, allowing the animation to loop properly. This is shown in Figure 12.66.

If the character moves forward by 4 feet in the animation and the animation

图 12.66. 游戏中的行走循环，提取的根运动数据应用于角色的局部空间原点。图片由顽皮狗公司提供。（《神秘海域：德雷克的宝藏》© 2007/® SIE。由顽皮狗创作和开发。）

需要一秒钟才能完成，那么我们知道角色的平均移动速度为4英尺/秒。要让角色以不同的速度行走，我们只需调整行走循环动画的播放速率即可。例如，要让角色以2英尺/秒的速度行走，我们只需以一半的速度播放动画（R = 0.5）。

12.11.4.2 足部 IK

运动提取可以很好地使角色在直线移动时（或者更准确地说，当角色的移动路径与动画师绘制的路径完全匹配时）双脚看起来像是落地的。然而，真实的游戏角色必须以与原始手工绘制的运动路径不一致的方式进行旋转和移动（例如，在不平坦的地形上移动时）。这会导致额外的脚部滑动。

解决这个问题的一个方法是使用反向运动 (IK) 来校正脚部的任何滑动。其基本思路是分析动画，确定每只脚在哪些时间段内完全接触地面。当脚接触地面时，我们会记录它在世界空间中的位置。在脚接触地面的所有后续帧中，我们会使用反向运动 (IK) 来调整腿部的姿势，使脚部始终固定在正确的位置。这项技术听起来很简单，但要让它看起来和感觉起来都正确却非常具有挑战性。它需要大量的迭代和微调。而且，一些自然的人体动作——例如通过增加步幅来引导转弯——无法仅靠反向运动来实现。

此外，动画的外观和角色的体验之间存在很大的权衡，尤其是对于人类控制的角色而言。通常，玩家角色控制系统的响应速度更为重要。

Figure 12.66. Walk cycle in-game, with extracted root motion data applied to the local-space origin of the character. Image courtesy of Naughty Dog, Inc. (UNCHARTED: Drake's Fortune © 2007/® SIE. Created and developed by Naughty Dog.)

takes one second to complete, then we know that the character is moving at an average speed of 4 feet/second. To make the character walk at a different speed, we can simply scale the playback rate of the walk cycle animation. For example, to make the character walk at 2 feet/second, we can simply play the animation at half speed ($R = 0.5$).

12.11.4.2 Foot IK

Motion extraction does a good job of making a character's feet appear grounded when it is moving in a straight line (or, more correctly, when it moves in a path that exactly matches the path animated by the animator). However, a real game character must be turned and moved in ways that don't coincide with the original hand-animated path of motion (e.g., when moving over uneven terrain). This results in additional foot sliding.

One solution to this problem is to use IK to correct for any sliding in the feet. The basic idea is to analyze the animations to determine during which periods of time each foot is fully in contact with the ground. At the moment a foot contacts the ground, we note its world-space location. For all subsequent frames while that foot remains on the ground, we use IK to adjust the pose of the leg so that the foot remains fixed to the proper location. This technique sounds easy enough, but getting it to look and feel right can be very challenging. It requires a lot of iteration and fine-tuning. And some natural human motions—like leading into a turn by increasing your stride—cannot be produced by IK alone.

In addition, there is a big trade-off between the *look* of the animations and the *feel* of the character, particularly for a human-controlled character. It's generally more important for the player character control system to feel responsive

角色动画的完美呈现比游戏本身更有趣。关键在于：不要轻视在游戏中添加脚部反向运动或运动提取的任务。要预留充足的时间进行反复试验，并做好权衡取舍的准备，以确保你的玩家角色不仅外观精美，而且手感也同样出色。

12.11.5 其他类型的约束

还有许多其他类型的约束系统可以添加到游戏动画引擎中。一些示例包括：

• 注视。这是角色注视环境中兴趣点的能力。角色可以只用眼睛注视某个点，也可以用眼睛和头部注视，或者用眼睛、头部和整个上半身的扭转来注视。注视约束有时使用反向运动 (IK) 或程序化关节偏移来实现，但通常可以通过加法混合来实现更自然的外观。

• 掩体配准。这是指角色能够与作为掩体的物体完美对齐的能力。这通常通过上面描述的参考定位器技术实现。

• 掩体的进入和离开。如果角色可以寻找掩体，通常必须使用动画混合和自定义进入和离开动画来让角色进入和离开掩体。

• 穿越辅助。角色能够在环境中的障碍物上方、下方、周围或中间穿行，为游戏增添不少活力。这通常通过提供自定义动画并使用参考定位器来确保与要克服的障碍物正确匹配来实现。

and fun than it is for the character's animations to look perfect. The upshot is this: Do not take the task of adding foot IK or motion extraction to your game lightly. Budget time for a lot of trial and error, and be prepared to make trade-offs to ensure that your player character not only looks good but *feels* good as well.

12.11.5 Other Kinds of Constraints

There are plenty of other possible kinds of constraint systems that can be added to a game animation engine. Some examples include:

- *Look-at*. This is the ability for characters to look at points of interest in the environment. A character might look at a point with only his or her eyes, with eyes and head, or with eyes, head and a twist of the entire upper body. Look-at constraints are sometimes implemented using IK or procedural joint offsets, although a more natural look can often be achieved via additive blending.
- *Cover registration*. This is the ability for a character to align perfectly with an object that is serving as cover. This is often implemented via the reference locator technique described above.
- *Cover entry and departure*. If a character can take cover, animation blending and custom entry and departure animations must usually be used to get the character into and out of cover.
- *Traversal aids*. The ability for a character to navigate over, under, around or through obstacles in the environment can add a lot of life to a game. This is often done by providing custom animations and using a reference locator to ensure proper registration with the obstacle being overcome.

13
碰撞和刚体动力学

在现实世界中，固体物体本质上就是固体。它们通常会避免做出一些不可能的事情，比如独自穿过彼此。但在虚拟游戏世界中，除非我们指示，否则物体不会做任何事情，游戏程序员必须明确努力确保物体不会相互穿过。这就是任何游戏引擎的核心组件之一——碰撞检测系统——的作用。

游戏引擎的碰撞系统通常与物理引擎紧密结合。当然，物理学领域非常广阔，当今大多数游戏引擎所称的“物理学”更准确地说是刚体动力学模拟。刚体是一种理想化的、无限坚硬的、不可变形的固体物体。动力学一词指的是确定这些刚体在力的作用下如何随时间移动和相互作用的过程。刚体动力学模拟允许以高度交互和自然混乱的方式将运动传递给游戏中的物体——这种效果在使用固定动画剪辑移动物体时很难实现。

动态模拟大量使用碰撞检测系统，以便正确模拟模拟中物体的各种物理行为，包括相互弹跳、摩擦滑动、滚动和静止。当然，碰撞检测系统可以独立使用，无需动态模拟——许多游戏没有“物理”机制。

13
Collision and Rigid Body Dynamics

In the real world, solid objects are inherently, well…solid. They generally avoid doing impossible things, like passing through one another, all by themselves. But in a virtual game world, objects don't do anything unless we tell them to, and game programmers must make an explicit effort to ensure that objects do not pass through one another. This is the role of one of the central components of any game engine—the *collision detection system*.

A game engine's collision system is often closely integrated with a *physics engine*. Of course, the field of physics is vast, and what most of today's game engines call "physics" is more accurately described as a *rigid body dynamics* simulation. A *rigid body* is an idealized, infinitely hard, non-deformable solid object. The term *dynamics* refers to the process of determining how these rigid bodies *move* and *interact* over time under the influence of *forces*. A rigid body dynamics simulation allows motion to be imparted to objects in the game in a highly interactive and naturally chaotic manner—an effect that is much more difficult to achieve when using canned animation clips to move things about.

A dynamics simulation makes heavy use of the collision detection system in order to properly simulate various physical behaviors of the objects in the simulation, including bouncing off one another, sliding under friction, rolling and coming to rest. Of course, a collision detection system can be used standalone, without a dynamics simulation—many games do not have a "physics"

系统。但所有涉及物体在二维或三维空间中移动的游戏都具有某种形式的碰撞检测。

在本章中，我们将研究典型的碰撞检测系统和典型的物理（刚体动力学）系统的架构。在研究这两个紧密相关的系统的组件时，我们将了解它们背□□后的数学和理论。

13.1 你想在游戏中加入物理效果吗?

如今，大多数游戏引擎都具备某种物理模拟功能。一些物理效果，例如布娃娃的死亡，是玩家所期待的。其他效果，例如绳索、布料、头发或复杂的物理驱动机械，则可以增添一些难以言喻的特质，使游戏在竞争中脱颖而出。近年来，一些游戏工作室开始尝试高级物理模拟，包括近似实时的流体力学效果和可变形物体的模拟。但是，在游戏中添加物理效果并非没有代价，在我们致力于在游戏中实现详尽的物理驱动功能之前，我们应该（至少）了解其中的利弊。

13.1.1 物理系统的作用

以下仅列举了使用游戏物理系统可以实现的一些功能。

- 检测动态物体和静态世界几何之间的碰撞。
- 模拟受重力和其他力影响的自由刚体。
- 弹簧质量系统。
- 可破坏的建筑物和结构。
- 射线和形状投射（确定视线、子弹撞击等）。
- 触发体积（确定对象何时进入、离开或位于游戏世界中的预定义区域内）。
- 复杂机器（起重机、移动平台拼图等）。
- 陷阱（例如巨石崩落）。
- 可驾驶的车辆具有逼真的悬架。
- 布娃娃角色死亡。
- 动力布娃娃：传统动画和布娃娃物理之间的真实融合。

system at all. But all games that involve objects moving about in two- or three-dimensional space have some form of collision detection.

In this chapter, we'll investigate the architecture of both a typical collision detection system and a typical physics (rigid body dynamics) system. As we investigate the components of these two closely interrelated systems, we'll take a look at the mathematics and the theory that underlie them.

13.1 Do You Want Physics in Your Game?

Nowadays, most game engines have some kind of physical simulation capabilities. Some physical effects, like rag doll deaths, are simply expected by gamers. Other effects, like ropes, cloth, hair or complex physically driven machinery can add that *je ne sais quoi* that sets a game apart from its competitors. In recent years, some game studios have started experimenting with advanced physical simulations, including approximate real-time fluid mechanics effects and simulations of deformable bodies. But adding physics to a game is not without costs, and before we commit ourselves to implementing an exhaustive list of physics-driven features in our game, we should (at the very least) understand the trade-offs involved.

13.1.1 Things You Can Do with a Physics System

Here are just a few of the things you can do or have with a game physics system.

- Detect collisions between dynamic objects and static world geometry.
- Simulate free rigid bodies under the influence of gravity and other forces.
- Spring-mass systems.
- Destructible buildings and structures.
- Ray and shape casts (to determine line of sight, bullet impacts, etc.).
- Trigger volumes (determine when objects enter, leave or are inside predefined regions in the game world).
- Complex machines (cranes, moving platform puzzles and so on).
- Traps (such as an avalanche of boulders).
- Drivable vehicles with realistic suspensions.
- Rag doll character deaths.
- Powered rag doll: a realistic blend between traditional animation and rag doll physics.

- 悬挂道具（水壶、项链、剑）、半逼真的头发、服装动作。
- 布料模拟。
- 水面模拟和浮力。
- 音频传播。

并且这样的例子不胜枚举。

这里需要注意的是，除了在游戏中运行时运行物理模拟之外，我们还可以将其作为离线预处理步骤的一部分来运行，以生成动画剪辑。Maya 等动画工具提供了许多物理插件。NaturalMotion, Inc. 的 Endorphin 1 软件包 (http://www.naturalmotion.com/endorphin.htm) 也采用了这种方法。在本章中，我们将仅限于讨论运行时刚体动力学模拟，但离线工具也是一个强大的选项，我们在规划游戏项目时应该始终注意这一点。

13.1.2 物理有趣吗?

游戏中使用刚体动力学系统并不一定能提升游戏乐趣。物理模拟本身的混乱行为往往反而会降低游戏体验，而非提升其趣味性。物理带来的乐趣取决于诸多因素，包括模拟本身的质量、与其他引擎系统集成的细致程度、选择物理驱动的玩法元素而非更直接控制的元素、物理元素如何与玩家目标和玩家角色的能力互动，以及游戏类型。

让我们看一下几种常见的游戏类型以及刚体动力学系统如何适应每种游戏类型。

13.1.2.1 模拟（Sims）

模拟的主要目标是准确重现真实体验。例如《飞行模拟器》、《GT赛车》和《纳斯卡赛车》系列游戏。显然，刚体动力学系统提供的真实感非常适合这类游戏。

1 NaturalMotion 还提供 Endorphin 的运行时版本，称为 Euphoria 。

- Dangling props (canteens, necklaces, swords), semi-realistic hair, clothing movements.
- Cloth simulations.
- Water surface simulations and buoyancy.
- Audio propagation.

And the list goes on.

We should note here that in addition to running a physics simulation at runtime in our game, we can also run a simulation as part of an offline preprocessing step in order to generate an animation clip. A number of physics plug-ins are available for animation tools like Maya. This is also the approach taken by the *Endorphin*[1] package by NaturalMotion, Inc. (http://www.naturalmotion.com/endorphin.htm). In this chapter, we'll restrict our discussion to runtime rigid body dynamics simulations, but offline tools are a powerful option, of which we should always remain aware as we plan our game projects.

13.1.2 Is Physics Fun?

The presence of a rigid body dynamics system in a game does not necessarily make the game fun. More often than not, the inherently chaotic behavior of a physics sim can actually detract from the gameplay experience rather than enhancing it. The fun derived from physics depends on many factors, including the quality of the simulation itself, the care with which it has been integrated with other engine systems, the selection of physics-driven gameplay elements versus elements that are controlled in a more direct manner, how the physical elements interact with the goals of the player and the abilities of the player character, and the genre of game being made.

Let's take a look at a few broad game genres and how a rigid body dynamics system might fit into each one.

13.1.2.1 Simulations (Sims)

The primary goal of a sim is to accurately reproduce a real-life experience. Examples include the *Flight Simulator*, *Gran Turismo* and *NASCAR Racing* series of games. Clearly, the realism provided by a rigid body dynamics system fits extremely well into these kinds of games.

[1] NaturalMotion also offers a runtime version of Endorphin called *Euphoria*.

13.1.2.2 物理益智游戏

物理谜题的本质是让玩家玩转动态模拟的玩具。因此，这类游戏的核心机制几乎完全依赖于物理原理。这类游戏的例子包括《桥梁建造者》（Bridge Builder）、《不可思议的机器》（The Incredible Machine）、在线游戏《神奇的装置》（Fantastic Contraption）以及iPhone版《蜡笔物理》（Crayon Physics）。

13.1.2.3 沙盒游戏

沙盒游戏中，可能完全没有目标，也可能有大量可选目标。玩家的主要目标通常是“捣鼓”，探索游戏世界中的物体可以做什么。沙盒游戏的例子包括《围攻》、《孢子》、《小小大星球》系列，当然还有《我的世界》。

沙盒游戏可以很好地利用逼真的动态模拟，尤其是在乐趣主要源于游戏世界中物体之间逼真（或半逼真）的互动的情况下。因此，在这种情况下，物理效果本身就很有趣。然而，许多游戏为了增加趣味性而牺牲了真实感（例如，夸张的爆炸、比正常更强或更弱的重力等等）。因此，动态模拟可能需要进行各种调整才能达到合适的“感觉”。

13.1.2.4 基于目标和故事驱动的游戏

目标导向型游戏有规则和特定目标，玩家必须完成这些目标才能前进；而在故事驱动型游戏中，讲述故事至关重要。将物理系统融入这类游戏中可能颇具挑战性。我们通常会放弃控制权以换取逼真的模拟，而这种控制权的丧失可能会抑制玩家完成目标的能力或游戏讲述故事的能力。

例如，在基于角色的平台游戏中，我们希望玩家角色的移动方式既有趣又易于控制，但物理效果不一定逼真。在战争游戏中，我们可能希望桥梁以逼真的方式爆炸，但也可能希望确保碎片不会挡住玩家前进的唯一道路。在这类游戏中，物理效果通常并不一定有趣，事实上，当玩家的目标与游戏世界中物体的物理模拟行为相冲突时，它往往会妨碍游戏的趣味性。因此，开发者必须谨慎地运用物理效果，并采取措施以各种方式控制模拟行为，以确保其不会妨碍游戏玩法。通常，为玩家提供摆脱困境的方法也是一个好主意。

13.1.2.2 Physics Puzzle Games

The whole idea of a physics puzzle is to let the user play around with dynamically simulated toys. So obviously this kind of game relies almost entirely on physics for its core mechanic. Examples of this genre include *Bridge Builder*, *The Incredible Machine*, the online game *Fantastic Contraption*, and *Crayon Physics* for the iPhone.

13.1.2.3 Sandbox Games

In a sandbox game, there may be no objectives at all, or there may be a large number of optional objectives. The player's primary objective is usually to "mess around" and explore what the objects in the game world can be made to do. Examples of sandbox games include *Besiege*, *Spore*, the *LittleBigPlanet* series, and of course *Minecraft*.

Sandbox games can put a realistic dynamics simulation to good use, especially if much of the fun is derived from playing with realistic (or semi-realistic) interactions between objects in the game world. So in these contexts, physics can be fun in and of itself. However, many games trade realism for an increased fun factor (e.g., larger-than-life explosions, gravity that is stronger or weaker than normal, etc.). So the dynamics simulation may need to be tweaked in various ways to achieve the right "feel."

13.1.2.4 Goal-Based and Story-Driven Games

A goal-based game has rules and specific objectives that the player must accomplish in order to progress; in a story-driven game, telling a story is of paramount importance. Integrating a physics system into these kinds of games can be tricky. We generally give away *control* in exchange for a *realistic simulation*, and this loss of control can inhibit the player's ability to accomplish goals or the game's ability to tell the story.

For example, in a character-based platformer game, we want the player character to move in ways that are fun and easy to control but not necessarily physically realistic. In a war game, we might want a bridge to explode in a realistic way, but we also may want to ensure that the debris doesn't end up blocking the player's only path forward. In these kinds of games, physics is often not necessarily fun, and in fact it can often get in the way of fun when the player's goals are at odds with the physically simulated behaviors of the objects in the game world. Therefore, developers must be careful to apply physics judiciously and take steps to control the behavior of the simulation in various ways to ensure it doesn't get in the way of gameplay. It's usually a good idea to provide the player with a way out of difficult situations, too.

《光环》系列游戏就是一个很好的例子，玩家可以按 X 键翻转倒置的车辆。

13.1.3 物理对游戏的影响

在游戏中添加物理模拟会对项目和游戏玩法产生各种影响。以下是一些涵盖不同游戏开发领域的示例。

13.1.3.1 设计影响

• 可预测性。物理模拟行为与动画行为之间固有的混乱性和多变性也是不可预测性的根源。如果某件事每次都必须以某种方式发生，通常最好将其制作成动画，而不是试图强制动态模拟可靠地生成该动作。

• 调整和控制。物理定律（在精确建模的情况下）是固定的。在游戏中，我们可以调整重力值或刚体的恢复系数，从而获得一定程度的控制力。然而，调整物理参数的结果往往是间接的，难以直观呈现。调整力以使角色朝所需方向移动比调整角色行走的动画要困难得多。

• 突发行为。有时，物理学会给游戏带来意想不到的功能——例如，《军团要塞经典版》中的火箭发射器跳跃技巧、《光晕》中高空爆炸的疣猪号以及《心理战》中飞行的“冲浪板”。

一般来说，游戏设计通常应该推动游戏引擎的物理要求，而不是相反。

13.1.3.2 工程影响

• 工具管道。良好的碰撞/物理管道需要时间来构建和维护。

• 用户界面。玩家如何控制游戏中的物理物体？是射击它们？撞上它们？还是捡起它们？是像《入侵者》那样用虚拟手臂握住它们？还是像《半条命2》那样用“重力枪”？

• 碰撞检测。用于动态模拟的碰撞模型可能需要比非物理驱动的碰撞模型更详细、更仔细地构建。

A good example of this can be found in the *Halo* series of games, where the player can press X to flip over a vehicle that has landed upside-down.

13.1.3 Impact of Physics on a Game

Adding a physics simulation to a game can have all sorts of impacts on the project and the gameplay. Here are a few examples across various game development disciplines.

13.1.3.1 Design Impacts

- *Predictability*. The inherent chaos and variability that sets a physically simulated behavior apart from an animated one is also a source of unpredictability. If something absolutely must happen a certain way every time, it's usually better to animate it than to try to coerce your dynamics simulation into producing the motion reliably.
- *Tuning and control*. The laws of physics (when modeled accurately) are fixed. In a game, we can tweak the value of gravity or the coefficient of restitution of a rigid body, which gives back some degree of control. However, the results of tweaking physics parameters are often indirect and difficult to visualize. It's much harder to tweak a force in order to get a character to move in the desired direction than it is to tweak an animation of a character walking.
- *Emergent behaviors*. Sometimes physics introduces unexpected features into a game—for example, the rocket-launcher jump trick in *Team Fortress Classic*, the high-flying exploding Warthog in *Halo* and the flying "surfboards" in *PsyOps*.

In general, the game design should usually drive the physics requirements of a game engine—not the other way around.

13.1.3.2 Engineering Impacts

- *Tools pipeline*. A good collision/physics pipeline takes time to build and maintain.
- *User interface*. How does the player control the physics objects in the world? Does he or she shoot them? Walk into them? Pick them up? Does he or she hold them using virtual arms, as in *Trespasser*? Or using a "gravity gun," as in *Half-Life 2*?
- *Collision detection*. Collision models intended for use within a dynamics simulation may need to be more detailed and more carefully constructed than their non-physics-driven counterparts.

• AI。在存在物理模拟物体的情况下，路径可能无法预测。引擎可能需要处理可能移动或爆炸的动态掩体点。AI 能否利用物理原理发挥其优势？

• 行为异常的物体。动画驱动的物体之间可能会轻微地相互穿透，且几乎没有不良影响。但当由动态模拟驱动时，物体可能会以意想不到的方式相互弹开或严重抖动。可能需要应用碰撞过滤来允许物体轻微地相互穿透。可能需要设置一些机制来确保物体能够正常稳定并进入休眠状态。

• 布娃娃物理。布娃娃需要大量微调，并且在模拟中经常会出现不稳定的情况。动画可能会导致角色身体的某些部分与其他碰撞体发生穿透——当角色变成布娃娃时，这些穿透可能会导致严重的不稳定性。必须采取措施避免这种情况。

• 图形。物理驱动的运动可能会影响可渲染对象的边界体积（否则它们会保持静态或更可预测）。可破坏的建筑物和物体的存在可能会使某些预计算的照明和阴影方法失效。

• 网络和多人游戏。不影响游戏玩法的物理效果可以在每台客户端机器上单独（且独立）模拟。但是，对游戏玩法有影响的物理效果（例如手榴弹的轨迹）必须在服务器上模拟，并在所有客户端上准确复制。

• 录制和回放。录制游戏过程并在稍后回放的功能非常有助于调试/测试，同时也是一个有趣的游戏功能。此功能的实现难度很大，因为它要求每个引擎系统都以确定性的方式运行，以确保回放过程中的所有内容与录制时完全相同。如果您的物理模拟缺乏确定性，这可能会成为一个很大的缺陷。

13.1.3.3 艺术影响

• 额外的工具和工作流程复杂性。需要装配具有质量、摩擦、约束和其他属性的物体以供动态模拟使用，这也使得艺术部门的工作更加困难。

• 更复杂的内容。我们可能需要一个物体的多个视觉上相同的版本，但具有不同的碰撞和动力学配置，以实现不同的效果。

- *AI*. Pathing may not be predictable in the presence of physically simulated objects. The engine may need to handle dynamic cover points that can move or blow up. Can the AI use the physics to its advantage?
- *Misbehaved objects*. Animation-driven objects can clip slightly through one another with few or no ill effects. But when driven by a dynamics simulation, objects may bounce off one another in unexpected ways or jitter badly. Collision filtering may need to be applied to permit objects to interpenetrate slightly. Mechanisms may need to be put in place to ensure that objects settle and go to sleep properly.
- *Rag doll physics*. Rag dolls require a lot of fine-tuning and often suffer from instability in the simulation. An animation may drive parts of a character's body into penetration with other collision volumes—when the character turns into a rag doll, these interpenetrations can cause enormous instability. Steps must be taken to avoid this.
- *Graphics*. Physics-driven motion can have an effect on renderable objects' bounding volumes (where they would otherwise be static or more predictable). The presence of destructible buildings and objects can invalidate some kinds of precomputed lighting and shadow methods.
- *Networking and multiplayer*. Physics effects that do not affect gameplay may be simulated exclusively (and independently) on each client machine. However, physics that has an effect on gameplay (such as the trajectory that a grenade follows) must be simulated on the server and accurately replicated on all clients.
- *Record and playback*. The ability to record gameplay and play it back at a later time is very useful as a debugging/testing aid, and it can also serve as a fun game feature. This feature is difficult to implement because it requires every engine system to behave in a deterministic manner, so that everything will play out exactly in the same way during playback as it did when the recording was made. If your physics simulation isn't deterministic, this can become a major fly in the ointment.

13.1.3.3 Art Impacts

- *Additional tool and workflow complexity*. The need to rig up objects with mass, friction, constraints and other attributes for consumption by the dynamics simulation makes the art department's job more difficult as well.
- *More complex content*. We may need multiple visually identical versions of an object with different collision and dynamics configurations for dif-

不同的目的——例如，原始版本和可破坏版本。

- 失去控制。物理驱动对象的不可预测性使得控制场景的艺术构图变得困难。

13.1.3.4 其他影响

- 跨学科影响。在游戏中引入动态模拟需要工程、艺术、音频和设计之间的密切合作。

- 生产影响。物理学会增加项目的开发成本、技术和组织的复杂性以及风险。

在探索了这些影响之后，如今大多数团队都会选择将刚体动力学系统集成到游戏中。只要经过周密的规划和明智的选择，将物理效果添加到游戏中就能带来丰厚的回报。正如我们将在下文中看到的，第三方中间件正在让物理效果比以往任何时候都更容易实现。

13.2 碰撞/物理中间件

编写碰撞系统和刚体动力学模拟是一项极具挑战性且耗时的工作。游戏引擎的碰撞/物理系统通常占据了源代码的很大一部分。这需要编写和维护大量的代码！

值得庆幸的是，现在已经有许多强大且高质量的碰撞/物理引擎可用，它们既有商业产品，也有开源版本。以下列出了其中一些。有关各种物理 SDK 的优缺点的讨论，请访问在线游戏开发论坛（例如，http://www.gamedev.net/community/forums/topic.asp?topic_id=463024）。

13.2.1 常微分方程

ODE 的全称是“Open Dynamics Engine”（开放动力学引擎）（http://www.ode.org）。顾名思义，ODE 是一个开源的碰撞和刚体动力学 SDK。其功能集与 Havok 等商业产品类似。它的优势包括免费（对于小型游戏工作室和学校项目来说，这是一个很大的优势！）以及提供完整的源代码（这使得调试更加容易，并提供了修改物理引擎以满足特定游戏需求的可能性）。

ferent purposes—for example, a pristine version and a destructible version.

- *Loss of control*. The unpredictability of physics-driven objects can make it difficult to control the artistic composition of a scene.

13.1.3.4 Other Impacts

- *Interdisciplinary impacts*. The introduction of a dynamics simulation into your game requires close cooperation between engineering, art, audio and design.
- *Production impacts*. Physics can add to a project's development costs, technical and organizational complexity and risk.

Having explored the impacts, most teams today do choose to integrate a rigid body dynamics system into their games. With some careful planning and wise choices along the way, adding physics to your game can be rewarding and fruitful. And as we'll see below, third-party middleware is making physics more accessible than ever.

13.2 Collision/Physics Middleware

Writing a collision system and rigid body dynamics simulation is challenging and time-consuming work. The collision/physics system of a game engine can account for a significant percentage of the source code in a typical game engine. That's a lot of code to write and maintain!

Thankfully, a number of robust, high-quality collision/physics engines are now available, either as commercial products or in open source form. Some of these are listed below. For a discussion of the pros and cons of various physics SDKs, check out the online game development forums (e.g., http://www.gamedev.net/community/forums/topic.asp?topic_id=463024).

13.2.1 ODE

ODE stands for "Open Dynamics Engine" (http://www.ode.org). As its name implies, ODE is an open source collision and rigid body dynamics SDK. Its feature set is similar to a commercial product like Havok. Its benefits include being free (a big plus for small game studios and school projects!) and the availability of full source code (which makes debugging much easier and opens up the possibility of modifying the physics engine to meet the specific needs of a particular game).

13.2.2 项目符号

Bullet 是一个开源碰撞检测和物理库，广泛应用于游戏和电影行业。它的碰撞引擎与其动态模拟集成，但提供了钩子函数，以便碰撞系统可以独立使用或与其他物理引擎集成。它支持连续碰撞检测 (CCD)，也称为撞击时间 (TOI) 碰撞检测。正如我们将在下文中看到的，当模拟中包含小型快速移动的物体时，此功能非常有用。Bullet SDK 可在 http://code.google.com/p/bullet/ 下载，Bullet 维基百科位于 http://www.bulletphysics.com/mediawiki-1.5.8/index.php?title=Main_Page。

13.2.3 真轴

TrueAxis 是另一个碰撞/物理 SDK。非商业用途免费。
您可以在 http://trueaxis.com 了解有关 TrueAxis 的更多信息。

13.2.4 PhysX

PhysX 最初是一个名为 Novodex 的库，由 Ageia 公司制作和发行，是其推广专用物理协处理器战略的一部分。后来，它被 NVIDIA 收购并进行了重新设计，使其能够使用 NVIDIA 的 GPU 作为协处理器运行。（它也可以完全在 CPU 上运行，无需 GPU 支持。）PhysX 的下载地址为 http://www.nvidia.com/object/nvidia_physx.html。Ageia 和 NVIDIA 的营销策略之一是完全免费提供 CPU 版本的 SDK，以推动物理协处理器市场的发展。开发者也可以付费获取完整的源代码，并根据需要自定义库。PhysX 现已与 NVIDIA 的可扩展多平台动力学框架 APEX 集成。PhysX/APEX 适用于 Windows、Linux、Mac、Android、Xbox 360、PlayStation 3、Xbox One、PlayStation 4 和 Wii。

13.2.5 Havok

Havok 是商业物理 SDK 的黄金标准，提供最丰富的功能集之一，并在所有支持平台上拥有卓越的性能。（它也是最昂贵的解决方案。）Havok 由核心碰撞/物理引擎以及一系列可选的附加产品组成，包括车辆物理系统、可破坏环境建模系统以及可直接集成到 Havok 布娃娃物理系统中的全功能动画 SDK。它适用于 Xbox 360、

13.2.2 Bullet

Bullet is an open source collision detection and physics library used by both the game and film industries. Its collision engine is integrated with its dynamics simulation, but hooks are provided so that the collision system can be used stand-alone or integrated with other physics engines. It supports *continuous collision detection* (CCD)—also known as *time of impact* (TOI) collision detection—which, as we'll see below, can be extremely helpful when a simulation includes small, fast-moving objects. The Bullet SDK is available for download at http://code.google.com/p/bullet/, and the Bullet wiki is located at http://www.bulletphysics.com/mediawiki-1.5.8/index.php?title=Main_Page.

13.2.3 TrueAxis

TrueAxis is another collision/physics SDK. It is free for non-commercial use. You can learn more about TrueAxis at http://trueaxis.com.

13.2.4 PhysX

PhysX started out as a library called Novodex, produced and distributed by Ageia as part of their strategy to market their dedicated physics coprocessor. It was bought by NVIDIA and retooled so that it can run using NVIDIA's GPUs as a coprocessor. (It can also run entirely on a CPU, without GPU support.) It is available at http://www.nvidia.com/object/nvidia_physx.html. Part of Ageia's and NVIDIA's marketing strategy has been to provide the CPU version of the SDK entirely for free, in order to drive the physics coprocessor market forward. Developers can also pay a fee to obtain full source code and the ability to customize the library as needed. PhysX is now combined with APEX, NVIDIA's scalable multiplatform dynamics framework. PhysX/APEX is available for Windows, Linux, Mac, Android, Xbox 360, PlayStation 3, Xbox One, PlayStation 4 and Wii.

13.2.5 Havok

Havok is the gold standard in commercial physics SDKs, providing one of the richest feature sets available and boasting excellent performance characteristics on all supported platforms. (It's also the most expensive solution.) Havok is comprised of a core collision/physics engine, plus a number of optional add-on products including a vehicle physics system, a system for modeling destructible environments and a fully featured animation SDK with direct integration into Havok's rag doll physics system. It is available on Xbox 360,

支持 PlayStation 3、Xbox One、PlayStation 4、PlayStation Vita、Wii、Wii U、Windows 8、Android、Apple Mac 和 iOS。更多 Havok 资讯，请访问 http://www.havok.com。

13.2.6 物理抽象层（PAL）

物理抽象层 (PAL) 是一个开源库，允许开发者在一个项目中使用多个物理 SDK。它为 PhysX (Novodex)、Newton、ODE、OpenTissue、Tokamak、TrueAxis 和其他一些 SDK 提供了接口。您可以在 http://www.adrianboeing.com/pal/index.html 上了解更多关于 PAL 的信息。

13.2.7 数字分子物质（DMM）

位于瑞士日内瓦的 Pixelux Entertainment SA 公司开发了一款独特的物理引擎，名为“数字分子物质”（DMM），它使用有限元法模拟可变形物体和易碎物体的动力学。该引擎包含离线和运行时组件。它于 2008 年发布，并在 LucasArts 的《星球大战：原力释放》中得到实际应用。关于可变形物体力学的讨论超出了本文的范围，但您可以在 http://www. 上了解更多关于 DMM 的信息。

pixeluxentertainment.com。

13.3 碰撞检测系统

游戏引擎碰撞检测系统的主要目的是确定游戏世界中的任意物体是否发生接触。为了解决这个问题，每个逻辑对象都由一个或多个几何形状表示。这些形状通常非常简单，例如球体、盒子和胶囊体。但是，也可以使用更复杂的形状。碰撞系统会判断在任意给定时刻，任意形状是否相交（即重叠）。因此，碰撞检测系统本质上是一个高级的几何相交测试器。

当然，碰撞系统的作用远不止回答形状相交是否成立的问题。它还提供了每次接触的相关信息。接触信息可用于防止屏幕上出现不切实际的视觉异常，例如物体相互穿透。这通常是通过在渲染下一帧之前将所有穿透的物体移开来实现的。碰撞可以为物体提供支撑——一个或多个接触点共同作用，使物体静止下来，与重力和/或作用于其上的任何其他力保持平衡。碰撞还可以用于

PlayStation 3, Xbox One, PlayStation 4, PlayStation Vita, Wii, Wii U, Windows 8, Android, Apple Mac and iOS. You can learn more about Havok at http://www.havok.com.

13.2.6 Physics Abstraction Layer (PAL)

The Physics Abstraction Layer (PAL) is an open source library that allows developers to work with more than one physics SDK on a single project. It provides hooks for PhysX (Novodex), Newton, ODE, OpenTissue, Tokamak, TrueAxis and a few other SDKs. You can read more about PAL at http://www.adrianboeing.com/pal/index.html.

13.2.7 Digital Molecular Matter (DMM)

Pixelux Entertainment S.A., located in Geneva, Switzerland, has produced a unique physics engine that uses finite element methods to simulate the dynamics of deformable bodies and breakable objects, called Digital Molecular Matter (DMM). The engine has both an offline and a runtime component. It was released in 2008 and can be seen in action in LucasArts' *Star Wars: The Force Unleashed*. A discussion of deformable body mechanics is beyond our scope here, but you can read more about DMM at http://www.pixeluxentertainment.com.

13.3 The Collision Detection System

The primary purpose of a game engine's collision detection system is to determine whether any of the objects in the game world have come into *contact*. To answer this question, each logical object is represented by one or more geometric *shapes*. These shapes are usually quite simple, such as spheres, boxes and capsules. However, more complex shapes can also be used. The collision system determines whether or not any of the shapes are *intersecting* (i.e., overlapping) at any given moment in time. So a collision detection system is essentially a glorified geometric intersection tester.

Of course, the collision system does more than answer yes/no questions about shape intersection. It also provides relevant information about the nature of each contact. Contact information can be used to prevent unrealistic visual anomalies on-screen, such as objects *interpenetrating* one another. This is generally accomplished by moving all interpenetrating objects apart prior to rendering the next frame. Collisions can provide *support* for an object—one or more contacts that together allow the object to come to rest, in equilibrium with gravity and/or any other forces acting on it. Collisions can also be used

其他用途包括使导弹击中目标时爆炸，或使玩家角色穿过漂浮的医疗包时获得生命值提升。刚体动力学模拟通常是碰撞系统最苛刻的客户端，它用它来模拟弹跳、滚动、滑动和静止等物理上逼真的行为。当然，即使是没有物理系统的游戏，仍然可以大量使用碰撞检测引擎。

本章我们将简要介绍碰撞检测引擎的工作原理。如果想深入探讨这个主题，可以参考一些关于实时碰撞检测的优秀书籍，包括[14]、[48]和[11]。

13.3.1 可碰□□撞实体

如果我们希望游戏中某个特定的逻辑对象能够与其他对象发生碰撞，我们需要为其提供一个碰撞表示，描述该对象的形状及其在游戏世界中的位置和方向。这是一个独立的数据结构，独立于对象的游戏玩法表示（定义其在游戏中的角色和行为的代码和数据），也独立于其视觉表示（可能是三角形网格、细分曲面、粒子效果或其他视觉表示的实例）。

从检测相交的角度来看，我们通常倾向于选择几何和数学上简单的形状。例如，为了进行碰撞检测，一块岩石可以被建模为一个球体；汽车的引擎盖可以用一个矩形框来表示；人体可以用一系列相互连接的胶囊（药丸状的体积）来近似表示。理想情况下，只有当简单的表示不足以实现游戏中所需的行为时，我们才应该采用更复杂的形状。图 13.1 展示了一些使用简单形状来近似物体体积以进行碰撞检测的示例。

Havok 使用术语“可碰撞”来描述可以参与碰撞检测的独特刚性物体。它用 C++ 类 hkpCollidable 的实例表示每个可碰撞物体。PhysX 将其刚性对象称为 actors，并将它们表示为 NxActor 类的实例。在这两个库中，可碰撞实体包含两条基本信息——形状和变换。形状描述可碰撞物体的几何形状，变换描述形状在游戏世界中的位置和方向。可碰撞物体需要变换，原因有三：

1. 从技术上讲，形状仅描述物体的形式（即它是球体、盒子、胶囊还是其他类型的体积）。

for other purposes, such as to cause a missile to explode when it strikes its target or to give the player character a health boost when he passes through a floating health pack. A rigid body dynamics simulation is often the most demanding client of the collision system, using it to mimic physically realistic behaviors like bouncing, rolling, sliding and coming to rest. But, of course, even games that have no physics system can still make heavy use of a collision detection engine.

In this chapter, we'll go on a brief high-level tour of how collision detection engines work. For an in-depth treatment of this topic, a number of excellent books on real-time collision detection are available, including [14], [48] and [11].

13.3.1 Collidable Entities

If we want a particular logical object in our game to be capable of colliding with other objects, we need to provide it with a *collision representation*, describing the object's shape and its position and orientation in the game world. This is a distinct data structure, separate from the object's *gameplay representation* (the code and data that define its role and behavior in the game) and separate from its *visual representation* (which might be an instance of a triangle mesh, a subdivision surface, a particle effect or some other visual representation).

From the point of view of detecting intersections, we generally favor shapes that are geometrically and mathematically simple. For example, a rock might be modeled as a sphere for collision purposes; the hood of a car might be represented by a rectangular box; a human body might be approximated by a collection of interconnected *capsules* (pill-shaped volumes). Ideally, we should resort to a more complex shape only when a simpler representation proves inadequate to achieve the desired behavior in the game. Figure 13.1 shows a few examples of using simple shapes to approximate object volumes for collision detection purposes.

Havok uses the term *collidable* to describe a distinct, rigid object that can take part in collision detection. It represents each collidable with an instance of the C++ class `hkpCollidable`. PhysX calls its rigid objects *actors* and represents them as instances of the class `NxActor`. In both of these libraries, a collidable entity contains two basic pieces of information—a *shape* and a *transform*. The shape describes the collidable's geometric form, and the transform describes the shape's position and orientation in the game world. Collidables need transforms for three reasons:

1. Technically speaking, a shape only describes the form of an object (i.e., whether it is a sphere, a box, a capsule or some other kind of volume).

图 13.1。简单的几何形状通常用于近似游戏中物体的碰撞体积。

它也可以描述物体的大小（例如，球体的半径或盒子的尺寸）。但形状通常以原点为中心，并相对于坐标轴具有某种规范的方向。因此，为了使形状可用，必须对其进行变换，以便在世界空间中对其进行适当的定位和方向调整。

2. 游戏中的许多对象都是动态的。如果我们必须单独移动形状的各个特征（顶点、平面等），那么在空间中移动任意复杂的形状可能会非常耗时。但是，通过变换，任何形状都可以在空间中以低成本移动，无论形状的特征多么简单或复杂。

3. 描述某些更复杂形状的信息可能会占用大量内存。因此，允许多个可碰撞体共享一个形状描述可能会很有帮助。例如，在赛车游戏中，许多赛车的形状信息可能相同。在这种情况下，游戏中的所有可碰撞体都可以共享一个赛车形状。

游戏中的任何特定对象可能根本没有可碰撞对象（如果它不需要碰撞检测服务），只有一个可碰撞对象（如果该对象是一个简单的刚体）或多个可碰撞对象（例如，每个对象代表一个铰接式机器人手臂的一个刚性部件）。

Figure 13.1. Simple geometric shapes are often used to approximate the collision volumes of the objects in a game.

It may also describe the object's size (e.g., the radius of a sphere or the dimensions of a box). But a shape is usually defined with its center at the origin and in some sort of canonical orientation relative to the coordinate axes. To be useful, a shape must therefore be transformed in order to position and orient it appropriately in world space.

2. Many of the objects in a game are dynamic. Moving an arbitrarily complex shape through space could be expensive if we had to move the *features* of the shape (vertices, planes, etc.) individually. But with a transform, any shape can be moved in space inexpensively, no matter how simple or complex the shape's features may be.

3. The information describing some of the more complex kinds of shapes can take up a nontrivial amount of memory. So, it can be beneficial to permit more than one collidable to share a single shape description. For example, in a racing game, the shape information for many of the cars might be identical. In that case, all of the car collidables in the game can share a single car shape.

Any particular object in the game may have no collidable at all (if it doesn't require collision detection services), a single collidable (if the object is a simple rigid body) or multiple collidables (each representing one rigid component of an articulated robot arm, for example).

13.3.2 碰撞/物理世界

碰撞系统通常通过一个称为碰撞世界 (collision world) 的单例数据结构来跟踪所有可碰撞实体。碰撞世界是游戏世界的完整表示，专为碰撞检测系统使用而设计。Havok 的碰撞世界是 hkpWorld 类的一个实例。同样，PhysX 的世界是 NxScene 的一个实例。ODE 使用 dSpace 类的一个实例来表示碰撞世界；它实际上是代表游戏中所有可碰撞形状的几何体积层次结构的根。

将所有碰撞信息保存在私有数据结构中，相比尝试将碰撞信息存储在游戏对象本身中，具有诸多优势。首先，碰撞世界只需包含那些可能相互碰撞的游戏对象的可碰撞数据。这消除了碰撞系统遍历任何不相关数据结构的需要。这种设计还允许以最高效的方式组织碰撞数据。例如，碰撞系统可以利用缓存一致性来最大化性能。碰撞世界也是一种有效的封装机制，从可理解性、可维护性、可测试性和软件复用潜力的角度来看，这通常是一个优势。

13.3.2.1 物理世界

如果游戏具有刚体动力学系统，它通常与碰撞系统紧密集成。它通常与碰撞系统共享其“世界”数据结构，并且模拟中的每个刚体通常与碰撞系统中的单个可碰撞对象相关联。这种设计在物理引擎中很常见，因为物理系统需要频繁且详细的碰撞查询。通常情况下，物理系统会实际驱动碰撞系统的运行，指示其在每个模拟时间步至少运行一次碰撞测试，有时甚至多次。因此，碰撞世界通常被称为碰撞/物理世界，有时简称为物理世界。

物理模拟中的每个动态刚体通常与碰撞系统中的单个可碰撞对象相关联（尽管并非所有可碰撞对象都必须是动态刚体）。例如，在 Havok 中，一个刚体由 hkpRigidBody 类的一个实例表示，并且每个刚体都有一个指向一个 hkpCollidable 的指针。在 PhysX 中，可碰撞和刚体的概念是混合的——NxActor 类同时满足这两个目的（尽管刚体的物理属性是单独存储的，但在

13.3.2 The Collision/Physics World

A collision system typically keeps track of all of its collidable entities via a singleton data structure known as the *collision world*. The collision world is a complete representation of the game world designed explicitly for use by the collision detection system. Havok's collision world is an instance of the class `hkpWorld`. Likewise, the PhysX world is an instance of `NxScene`. ODE uses an instance of class `dSpace` to represent the collision world; it is actually the root of a hierarchy of geometric volumes representing all the collidable shapes in the game.

Maintaining all collision information in a private data structure has a number of advantages over attempting to store collision information with the game objects themselves. For one thing, the collision world need only contain collidables for those game objects that can potentially collide with one another. This eliminates the need for the collision system to iterate over any irrelevant data structures. This design also permits collision data to be organized in the most efficient manner possible. The collision system can take advantage of cache coherency to maximize performance, for example. The collision world is also an effective encapsulation mechanism, which is generally a plus from the perspectives of understandability, maintainability, testability and the potential for software reuse.

13.3.2.1 The Physics World

If a game has a rigid body dynamics system, it is usually tightly integrated with the collision system. It typically shares its "world" data structure with the collision system, and each rigid body in the simulation is usually associated with a single collidable in the collision system. This design is commonplace among physics engines because of the frequent and detailed collision queries required by the physics system. It's typical for the physics system to actually *drive* the operation of the collision system, instructing it to run collision tests at least once, and sometimes multiple times, per simulation time step. For this reason, the collision world is often called the *collision/physics world* or sometimes just the *physics world*.

Each dynamic rigid body in the physics simulation is usually associated with a single collidable object in the collision system (although not all collidables need be dynamic rigid bodies). For example, in Havok, a rigid body is represented by an instance of the class `hkpRigidBody`, and each rigid body has a pointer to exactly one `hkpCollidable`. In PhysX, the concepts of collidable and rigid body are comingled—the `NxActor` class serves both purposes (although the physical properties of the rigid body are stored separately, in

NxBodyDesc 的一个实例）。在这两个 SDK 中，都可以告诉刚体其位置和方向在空间中是固定的，这意味着它将从动力学模拟中被省略，并且仅用作可碰撞体。

尽管这种紧密集成，大多数物理 SDK 至少会尝试将碰撞库与刚体动力学模拟分离。这使得碰撞系统可以作为独立的库使用（这对于不需要物理但需要检测碰撞的游戏来说非常重要）。这也意味着游戏工作室理论上可以完全替换物理 SDK 的碰撞系统，而无需重写动力学模拟。（实际上，这可能比听起来要难一些！）

13.3.3 形状概念

丰富的数学理论构成了我们日常形状概念的基础（参见 http://en.wikipedia.org/wiki/Shape）。为了便于理解，我们可以将形状简单地理解为由边界描述的空间区域，该区域具有明确的内部和外部。在二维空间中，形状具有面积，其边界由一条曲线或三条或更多条直边定义（在这种情况下，形状称为多边形）。在三维空间中，形状具有体积，其边界由曲面或多边形组成（在这种情况下，形状称为多面体）。

值得注意的是，某些类型的游戏对象，例如地形、河流或薄墙，可能最适合用表面来表示。在三维空间中，表面是一个二维几何实体，有正面和背面，但没有内部或外部。示例包括平面、三角形、细分曲面以及由一组相连的三角形或其他多边形构成的曲面。大多数碰撞 SDK 都支持表面基元，并扩展了“形状”一词，使其涵盖封闭体积和开放表面。

碰撞库通常允许通过可选的挤压参数来赋予表面体积。该参数指定了表面的“厚度”。这样做有助于减少小型快速移动物体与极薄表面之间发生碰撞失败的情况（即所谓的“子弹穿纸”问题——参见第 13.3.5.7 节）。

13.3.3.1 交叉口

我们对交集的概念都比较直观。从技术角度来说，这个术语源自集合论 (http://en.wikipedia.org/wiki/Intersection_ (set_theory))。两个集合的交集由两个集合共有元素的子集组成。用几何术语来说，两个形状的交集就是位于两个形状内部的所有点的集合（无穷大！）。

an instance of `NxBodyDesc`). In both SDKs, it is possible to tell a rigid body that its location and orientation are to be fixed in space, meaning that it will be omitted from the dynamics simulation and will serve as a collidable only.

Despite this tight integration, most physics SDKs do make at least some attempt to separate the collision library from the rigid body dynamics simulation. This permits the collision system to be used as a stand-alone library (which is important for games that don't need physics but do need to detect collisions). It also means that a game studio could *theoretically* replace a physics SDK's collision system entirely, without having to rewrite the dynamics simulation. (Practically speaking, this may be a bit harder than it sounds!)

13.3.3 Shape Concepts

A rich body of mathematical theory underlies the everyday concept of *shape* (see http://en.wikipedia.org/wiki/Shape). For our purposes, we can think of a shape simply as a region of space described by a boundary, with a definite *inside* and *outside*. In two dimensions, a shape has area, and its boundary is defined either by a curved line or by three or more straight edges (in which case it's a *polygon*). In three dimensions, a shape has volume, and its boundary is either a curved surface or is composed of polygons (in which case is it called a *polyhedron*).

It's important to note that some kinds of game objects, like terrain, rivers or thin walls, might be best represented by *surfaces*. In three-space, a surface is a two-dimensional geometric entity with a *front* and a *back* but no inside or outside. Examples include planes, triangles, subdivision surfaces and surfaces constructed from a group of connected triangles or other polygons. Most collision SDKs provide support for surface primitives and extend the term *shape* to encompass both closed volumes and open surfaces.

It's commonplace for collision libraries to allow surfaces to be given volume via an optional extrusion parameter. Such a parameter specifies how "thick" a surface should be. Doing this helps reduce the occurrence of missed collisions between small, fast-moving objects and infinitesimally thin surfaces (the so-called "bullet through paper" problem—see Section 13.3.5.7).

13.3.3.1 Intersection

We all have an intuitive notion of what an *intersection* is. Technically speaking, the term comes from set theory (http://en.wikipedia.org/wiki/Intersection_ (set_theory)). The intersection of two sets is comprised of the subset of members that are common to both sets. In geometrical terms, the intersection between two shapes is just the (infinitely large!) set of all points that lie inside both shapes.

13.3.3.2 联系方式

在游戏中，我们通常不关心严格意义上的交点，即一组点的交点。相反，我们只想知道两个物体是否相交。如果发生碰撞，碰撞系统通常会提供有关接触性质的额外信息。例如，这些信息使我们能够以物理上合理且高效的方式分离物体。

碰撞系统通常将接触信息打包成一个方便的数据结构，该结构可针对检测到的每个接触进行实例化。例如，Havok 将接触点作为 hkContactPoint 类的实例返回。接触信息通常包含一个分离向量——我们可以沿着该向量滑动物体，从而有效地将它们移出碰撞。它通常还包含关于哪些可碰撞体处于接触状态的信息，包括哪些形状相交，甚至可能包括这些形状的哪些特征处于接触状态。系统还可能返回其他信息，例如投射到分离法线上的物体的速度。

13.3.3.3 凸性

碰撞检测领域中最重要的概念之一是凸形和非凸形（即凹形）之间的区别。从技术角度来看，凸形的定义是：任何从形状内部发出的射线都不会重复穿过其表面。判断一个形状是否凸起的一个简单方法是想象用塑料薄膜将其收缩包装——如果它是凸起的，薄膜下方就不会留下任何气穴。因此，在二维空间中，圆形、矩形和三角形都是凸起的，但吃豆人不是。这个概念同样适用于三维空间。

凸性非常重要，因为正如我们将看到的，检测凸形之间的交点通常比检测凹形之间的交点更简单，计算量也更小。有关凸形的更多信息，请参阅 http://en.wikipedia.org/wiki/Convex。

13.3.4 碰撞基元

碰撞检测系统通常只能处理相对有限的形状类型。一些碰撞系统将这些形状称为碰撞基元，因为它们是构建更复杂形状的基本构件。在本节中，我们将简要介绍一些最常见的碰撞基元类型。

13.3.3.2 Contact

In games, we're not usually interested in finding the intersection in the strictest sense, as a set of points. Instead, we want to know simply whether or not two objects are intersecting. In the event of a collision, the collision system will usually provide additional information about the nature of the contact. This information allows us to separate the objects in a physically plausible and efficient way, for example.

Collision systems usually package contact information into a convenient data structure that can be instanced for each contact detected. For example, Havok returns contacts as instances of the class `hkContactPoint`. Contact information often includes a *separating vector*—a vector along which we can slide the objects in order to efficiently move them out of collision. It also typically contains information about which two collidables were in contact, including which individual shapes were intersecting and possibly even which individual features of those shapes were in contact. The system may also return additional information, such as the velocity of the bodies projected onto the separating normal.

13.3.3.3 Convexity

One of the most important concepts in the field of collision detection is the distinction between *convex* and *non-convex* (i.e., *concave*) shapes. Technically, a convex shape is defined as one for which no ray originating inside the shape will pass through its surface more than once. A simple way to determine if a shape is convex is to imagine shrink-wrapping it with plastic film—if it's convex, no air pockets will be left under the film. So in two dimensions, circles, rectangles and triangles are all convex, but Pac Man is not. The concept extends equally well to three dimensions.

The property of *convexity* is important because, as we'll see, it's generally simpler and less computationally intensive to detect intersections between convex shapes than concave ones. See http://en.wikipedia.org/wiki/Convex for more information about convex shapes.

13.3.4 Collision Primitives

Collision detection systems can usually work with a relatively limited set of shape types. Some collision systems refer to these shapes as *collision primitives* because they are the fundamental building blocks out of which more complex shapes can be constructed. In this section, we'll take a brief look at some of the most common types of collision primitives.

13.3.4.1 球体

最简单的三维体是球体。正如您所料，球体是最高效的碰撞基元。球体由一个中心点和一个半径表示。这些信息可以方便地打包成一个四元素浮点向量——这种格式与 SIMD 数学库配合得非常好。

13.3.4.2 胶囊

胶囊体是一个药丸状的体积，由一个圆柱体和两个半球形端盖组成。它可以被认为是一个扫描球体——球体从 A 点移动到 B 点时描绘出的形状。（然而，静态胶囊体和随时间扫描出胶囊形体积的球体之间存在一些重要差异，因此两者并不完全相同。）胶囊体通常用两个点和一个半径表示（图 13.2）。胶囊体比圆柱体或盒子更容易相交，因此它们常用于模拟大致呈圆柱形的物体，例如人体的四肢。

图 13.2。一个胶囊可以用两个点和一个半径来表示。

13.3.4.3 轴对齐边界框

轴对齐边界框 (AABB) 是一个矩形体（技术上称为长方体），其面与坐标系的轴平行。当然，在一个坐标系中轴对齐的边界框在另一个坐标系中不一定也是轴对齐的。因此，我们只能在 AABB 所对齐的特定坐标系的背景下讨论它。

AABB 可以方便地通过两个点来定义：一个点包含盒子在三个主轴上的最小坐标，另一个点包含盒子在三个主轴上的最大坐标。如图 13.3 所示。

轴对齐盒子的主要优点在于，它们可以高效地测试与其他轴对齐盒子的相互穿透。使用 AABB 的一大限制在于，如果要保持其计算优势，它们必须始终保持轴对齐。这意味着，如果使用 AABB 来近似游戏中物体的形状，

13.3.4.1 Spheres

The simplest three-dimensional volume is a sphere. And as you might expect, spheres are the most efficient kind of collision primitive. A sphere is represented by a center point and a radius. This information can be conveniently packed into a four-element floating-point vector—a format that works particularly well with SIMD math libraries.

13.3.4.2 Capsules

A capsule is a pill-shaped volume, composed of a cylinder and two hemispherical end caps. It can be thought of as a *swept sphere*—the shape that is traced out as a sphere moves from point A to point B. (There are, however, some important differences between a static capsule and a sphere that sweeps out a capsule-shaped volume over time, so the two are not identical.) Capsules are often represented by two points and a radius (Figure 13.2). Capsules are more efficient to intersect than cylinders or boxes, so they are often used to model objects that are roughly cylindrical, such as the limbs of a human body.

Figure 13.2. A capsule can be represented by two points and a radius.

13.3.4.3 Axis-Aligned Bounding Boxes

An axis-aligned bounding box (AABB) is a rectangular volume (technically known as a *cuboid*) whose faces are parallel to the axes of the coordinate system. Of course, a box that is axis-aligned in one coordinate system will not necessarily be axis-aligned in another. So we can only speak about an AABB in the context of the particular coordinate frame(s) with which it aligns.

An AABB can be conveniently defined by two points: one containing the minimum coordinates of the box along each of the three principal axes and the other containing its maximum coordinates. This is depicted in Figure 13.3.

The primary benefit of axis-aligned boxes is that they can be tested for interpenetration with other axis-aligned boxes in a highly efficient manner. The big limitation of using AABBs is that they must remain axis-aligned at all times if their computational advantages are to be maintained. This means that if an AABB is used to approximate the shape of an object in the game,

图 13.3. 轴对齐的盒子。

每当物体旋转时，AABB 都必须重新计算。即使一个物体大致呈盒子形状，当它偏离轴旋转时，其 AABB 也可能会退化成与其形状非常接近的近似值。如图 13.4 所示。

图 13.4. 只有当物体的主轴与坐标系的轴大致对齐时，AABB 才能很好地近似盒子状物体。

13.3.4.4 方向边界框

如果我们允许一个轴对齐的盒子相对于其坐标系旋转，我们就得到了所谓的定向边界框（OBB）。它通常由三个半维度（半宽、半深和半高）和一个变换来表示，该变换确定了盒子的中心位置并定义了其相对于坐标轴的方向。定向框是一种常用的碰撞原语，因为它们在拟合任意方向的物体方面表现更好，但它们的表示仍然非常简单。

13.3.4.5 离散定向多面体（DOP）

离散定向多面体 (DOP) 是 AABB 和 OBB 的更一般情况。它是一个凸多面体，可以近似表示物体的形状。DOP 的构造方法是：取无穷远处的多个平面，并沿其法向量滑动，直到它们与物体接触。

Figure 13.3. An axis-aligned box.

the AABB will have to be recalculated whenever that object rotates. Even if an object is roughly box-shaped, its AABB may degenerate into a very poor approximation to its shape when the object rotates off-axis. This is shown in Figure 13.4.

Figure 13.4. An AABB is only a good approximation to a box-shaped object when the object's principal axes are roughly aligned with the coordinated system's axes.

13.3.4.4 Oriented Bounding Boxes

If we permit an axis-aligned box to rotate relative to its coordinate system, we have what is known as an oriented bounding box (OBB). It is often represented by three half-dimensions (half-width, half-depth and half-height) and a transformation, which positions the center of the box and defines its orientation relative to the coordinate axes. Oriented boxes are a commonly used collision primitive because they do a better job at fitting arbitrarily oriented objects, yet their representation is still quite simple.

13.3.4.5 Discrete Oriented Polytopes (DOP)

A discrete oriented polytope (DOP) is a more-general case of the AABB and OBB. It is a convex polytope that approximates the shape of an object. A DOP can be constructed by taking a number of planes at infinity and sliding them along their normal vectors until they come into contact with the object whose

形状将被近似。AABB 是一个 6-DOP，其中平面法线平行于坐标轴。OBB 也是一个 6-DOP，其中平面法线平行于物体的自然主轴。k -DOP 由任意数量的平面 k 构成。构建 DOP 的一种常用方法是从相关物体的 OBB 开始，然后用其他平面将边缘和/或角以 45 度角斜切，以尝试产生更紧密的贴合。图 13.5 显示了 αk -DOP 的一个示例。

图 13.5。八个角均已斜切的 OBB 称为 14-DOP。

13.3.4.6 任意凸体

大多数碰撞引擎允许 3D 艺术家使用 Maya 等软件构建任意凸面体。艺术家用多边形（三角形或四边形）构建形状。离线工具会分析这些三角形，以确保它们确实构成凸多面体。如果形状通过了凸性测试，其三角形将被转换为平面集合（本质上是 αk -DOP），用 k 个平面方程或 k 个点和 k 个法向量表示。（如果发现形状是非凸的，仍然可以用多边形集合表示——下一节将对此进行描述。）此方法如图 13.6 所示。

凸体积的相交测试比我们目前讨论的简单几何图元更昂贵。然而，正如我们将在章节

图 13.6 任意凸体都可以用相交平面的集合来表示。

shape is to be approximated. An AABB is a 6-DOP in which the plane normals are taken parallel to the coordinate axes. An OBB is also a 6-DOP in which the plane normals are parallel to the object's natural principal axes. A *k*-DOP is constructed from an arbitrary number of planes *k*. A common method of constructing a DOP is to start with an OBB for the object in question and then bevel the edges and/or corners at 45 degrees with additional planes in an attempt to yield a tighter fit. An example of a *k*-DOP is shown in Figure 13.5.

Figure 13.5. An OBB that has been beveled on all eight corners is known as a 14-DOP.

13.3.4.6 Arbitrary Convex Volumes

Most collision engines permit arbitrary convex volume to be constructed by a 3D artist in a package like Maya. The artist builds the shape out of polygons (triangles or quads). An offline tool analyzes the triangles to ensure that they actually do form a convex polyhedron. If the shape passes the convexity test, its triangles are converted into a collection of planes (essentially a *k*-DOP), represented by *k* plane equations, or *k* points and *k* normal vectors. (If it is found to be non-convex, it can still be represented by a polygon soup—described in the next section.) This approach is depicted in Figure 13.6.

Convex volumes are more expensive to intersection-test than the simpler geometric primitives we've discussed thus far. However, as we'll see in Section

Figure 13.6. An arbitrary convex volume can be represented by a collection of intersecting planes.

13.3.5.5，某些高效的交点查找算法（例如 GJK）适用于这些形状，因为它们是凸的。

13.3.4.7 聚汤

一些碰撞系统也支持完全任意的非凸形状。这些形状通常由三角形或其他简单的多边形构成。因此，这类形状通常被称为多边形汤（Polygon Soup），或简称为多边形汤（Poly ...

您可能已经想到，使用多边形汤 (Poly Soup) 检测碰撞是成本最高的碰撞测试类型。实际上，碰撞引擎必须测试每个三角形，并且还必须妥善处理相邻三角形之间共享的三角形边的虚假相交。因此，大多数游戏会尝试将多边形汤形状的使用限制在不会参与动力学模拟的物体上。

聚汤有内部吗?

与凸形和简单形状不同，多边形汤不一定代表体积——它也可以代表开放表面。多边形汤形状通常不包含足够的信息，无法让碰撞系统区分封闭体积和开放表面。这使得我们难以确定应该将穿透多边形汤的物体推向哪个方向，以使两个物体脱离碰撞。

值得庆幸的是，这绝不是一个棘手的问题。

图 13.7. 多边形汤通常用于模拟复杂的静态表面，例如地形或建筑物。

13.3.5.5, certain highly efficient intersection-finding algorithms such as GJK are applicable to these shapes because they are convex.

13.3.4.7 Poly Soup

Some collision systems also support totally arbitrary, non-convex shapes. These are usually constructed out of triangles or other simple polygons. For this reason, this type of shape is often called a *polygon soup*, or *poly soup* for short. Poly soups are often used to model complex static geometry, such as terrain and buildings (Figure 13.7).

As you might imagine, detecting collisions with a poly soup is the most expensive kind of collision test. In effect, the collision engine must test every individual triangle, and it must also properly handle spurious intersections with triangle edges that are shared between adjacent triangles. As a result, most games try to limit the use of poly soup shapes to objects that will not take part in the dynamics simulation.

Does a Poly Soup Have an Inside?

Unlike convex and simple shapes, a poly soup does not necessarily represent a volume—it can represent an open surface as well. Poly soup shapes often don't include enough information to allow the collision system to differentiate between a closed volume and an open surface. This can make it difficult to know in which direction to push an object that is interpenetrating a poly soup in order to bring the two objects out of collision.

Thankfully, this is by no means an intractable problem. Each triangle in

Figure 13.7. A poly soup is often used to model complex static surfaces such as terrain or buildings.

多边形汤 (Poly Soup) 有正面和背面，由其顶点的缠绕顺序定义。因此，可以精心构建一个多边形汤形状，使所有多边形的顶点缠绕顺序一致（即相邻三角形始终“朝向”同一方向）。这赋予了整个多边形汤“正面”和“背面”的概念。如果我们还存储了给定多边形汤形状是开放还是封闭的信息（假设可以通过离线工具确定这一点），那么对于封闭形状，我们可以将“正面”和“背面”解释为“外部”和“内部”（反之亦然，具体取决于构建多边形汤时使用的约定）。

我们还可以针对某些类型的开放多边形汤（Open Poly Soup）形状（即表面）“伪造”内部和外部。例如，如果我们游戏中的地形由开放多边形汤表示，那么我们可以任意决定表面的正面始终指向远离地球的方向。这意味着“正面”应该始终对应于“外部”。实际上，为了实现这一点，我们可能需要以某种方式定制碰撞引擎，使其能够感知我们特定的约定选择。

13.3.4.8 复合形状

有些物体无法用单一形状充分近似，但可以用一组形状很好地近似。例如，一把椅子可以用两个盒子来建模——一个盒子包裹椅背，另一个盒子包裹椅座和四条腿。如图 13.8 所示。

对于非凸面物体的建模，复合形状通常比多边形汤（Poly Soup）更高效；两个或多个凸面体通常比单个多边形汤形状表现更好。此外，某些碰撞系统在测试碰撞时可以利用复合形状整体的凸面边界体。在 Havok 中，这被称为中间阶段碰撞检测。如图 13.9 中的示例所示，碰撞系统首先测试两个复合形状的凸面边界体。如果它们不相交，则系统根本不需要测试子形状是否发生碰撞。

图 13.8. 可以使用一对相互连接的盒子形状来建模椅子。

a poly soup has a front and a back, as defined by the winding order of its vertices. Therefore, it is possible to carefully construct a poly soup shape so that all of the polygons' vertex winding orders are consistent (i.e., adjacent triangles always "face" in the same direction). This gives the entire poly soup a notion of "front" and "back." If we also store information about whether a given poly soup shape is open or closed (presuming that this fact can be ascertained by offline tools), then for closed shapes, we can interpret "front" and "back" to mean "outside" and "inside" (or vice versa, depending on the conventions used when constructing the poly soup).

We can also "fake" an inside and outside for certain kinds of *open* poly soup shapes (i.e., surfaces). For example, if the terrain in our game is represented by an open poly soup, then we can decide arbitrarily that the front of the surface always points away from the Earth. This implies that "front" should always correspond to "outside." Practically speaking, to make this work, we would probably need to customize the collision engine in some way in order to make it aware of our particular choice of conventions.

13.3.4.8 Compound Shapes

Some objects that cannot be adequately approximated by a single shape can be approximated well by a *collection* of shapes. For example, a chair might be modeled out of two boxes—one for the back of the chair and one enclosing the seat and all four legs. This is shown in Figure 13.8.

A compound shape can often be a more-efficient alternative to a poly soup for modeling non-convex objects; two or more convex volumes can often outperform a single poly soup shape. What's more, some collision systems can take advantage of the convex bounding volume of the compound shape as a whole when testing for collisions. In Havok, this is called *midphase* collision detection. As the example in Figure 13.9 shows, the collision system first tests the convex bounding volumes of the two compound shapes. If they do not intersect, the system needn't test the subshapes for collisions at all.

Figure 13.8. A chair can be modeled using a pair of interconnected box shapes.

图 13.9。当发现一对复合形状的凸边界体（在本例中为球体 A 和球体 B）相交时，碰撞系统仅需要测试它们的子形状。

13.3.5 碰撞测试和解析几何

碰撞系统利用解析几何（三维体和表面的数学描述）来计算检测形状之间的相交。有关这一深奥而广泛的研究领域的更多详细信息，请参阅 http://en.wikipedia.org/wiki/Analytic_geometry。在本节中，我们将简要介绍解析几何背后的概念，展示一些常见示例，然后讨论针对任意凸多面体的广义 GJK 相交检测算法。

13.3.5.1 点与球

我们可以判断点 p 是否位于球体内，只需构造该点与球心 c 之间的分离向量 s ，然后检查其长度即可。如果 s 大于球体的半径 r ，则该点位于球体外；否则，则位于球体内：

$$\mathbf{s} = \mathbf{c} - \mathbf{p};$$
$$\text{if } |\mathbf{s}| \leq r, \text{then } \mathbf{p} \text{ is inside.}$$

13.3.5.2 球体与球体

判断两个球体是否相交几乎和用一个点和一个球体进行测试一样简单。同样，我们构造一个连接两个球体中心点的向量 s。我们取其长度，并将其与两个球体的半径之和进行比较。如果分离向量的长度小于或等于半径之和，则两个球体相交；否则，则两个球体不相交：

$$\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2; \tag{13.1}$$
$$\text{if } |\mathbf{s}| \leq (r_1 + r_2), \text{then spheres intersect.}$$

Figure 13.9. A collision system need only test the subshapes of a pair of compound shapes when their convex bounding volumes (in this case, Sphere A and Sphere B) are found to be intersecting.

13.3.5 Collision Testing and Analytical Geometry

A collision system makes use of *analytical geometry*—mathematical descriptions of three-dimensional volumes and surfaces—in order to detect intersections between shapes computationally. See http://en.wikipedia.org/wiki/Analytic_geometry for more details on this profound and broad area of research. In this section, we'll briefly introduce the concepts behind analytical geometry, show a few common examples and then discuss the generalized GJK intersection testing algorithm for arbitrary convex polyhedra.

13.3.5.1 Point versus Sphere

We can determine whether a point **p** lies within a sphere by simply forming the separation vector **s** between the point and the sphere's center **c** and then checking its length. If it is greater than the radius of the sphere r, then the point lies outside the sphere; otherwise, it lies inside:

$$\mathbf{s} = \mathbf{c} - \mathbf{p};$$
$$\text{if } |\mathbf{s}| \leq r, \text{then } \mathbf{p} \text{ is inside.}$$

13.3.5.2 Sphere versus Sphere

Determining if two spheres intersect is almost as simple as testing a point against a sphere. Again, we form a vector **s** connecting the center points of the two spheres. We take its length and compare it with the sum of the radii of the two spheres. If the length of the separating vector is less than or equal to the sum of the radii, the spheres intersect; otherwise, they do not:

$$\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2; \tag{13.1}$$
$$\text{if } |\mathbf{s}| \leq (r_1 + r_2), \text{then spheres intersect.}$$

为了避免计算向量 s 长度时固有的平方根运算，我们可以简单地对整个方程求平方。因此，方程 (13.1) 变为

$$\begin{aligned} \mathbf{s} &= \mathbf{c}_1 - \mathbf{c}_2; \\ |\mathbf{s}|^2 &= \mathbf{s} \cdot \mathbf{s}; \\ \text{if } |\mathbf{s}|^2 &\leq (r_1 + r_2)^2, \text{then spheres intersect.} \end{aligned}$$

13.3.5.3 分离轴定理

大多数碰撞检测系统都大量使用了分离轴定理 (http://en.wikipedia.org/wiki/Separating_axis _theorem)。该定理指出，如果可以找到一条轴，使两个凸形状的投影不重叠，那么我们可以肯定这两个形状根本不相交。如果不存在这样的轴，并且形状是凸的，那么我们肯定知道它们确实相交。（如果形状是凹的，那么尽管没有分离轴，它们也可能不会相互穿透。这就是我们在碰撞检测中倾向于使用凸形状的原因之一。）该定理在二维中最容易形象化。直观地说，如果可以找到一条线，使得物体 A 完全位于该线的一侧，而物体 B 完全位于另一侧，则物体 A 和 B 不重叠。这样的线称为分离线，它始终垂直于分离轴。因此，一旦我们找到了分界线，通过观察形状在垂直于分界线的轴上的投影，我们就可以更容易地相信该理论实际上是正确的。

二维凸形在轴上的投影就像物体在细线上留下的阴影。它始终是一条位于轴上的线段，表示物体在轴方向上的最大延伸。我们也可以将投影视为沿轴的最小和最大坐标，可以将其写成完全闭区间 [c min , c max] 。如图 13.10 所示，当两个形状之间存在分隔线时，它们的投影不会沿分隔轴重叠。但是，投影可能会沿其他非分隔轴重叠。

在三维空间中，分割线变成了分割平面，但分割轴仍然是一个轴（即一条无限长的线）。同样，三维凸形在轴上的投影是一条线段，我们可以用完全闭区间 [c min , c max] 来表示。

某些类型的形状具有使潜在分离轴显而易见的属性。为了检测两个这样的形状 A 和 B 之间的交点，我们可以依次将形状投影到每个潜在分离轴上，然后检查两个投影区间 [c A min , c A max] 和 [c B min , c B max] 是否满足：

To avoid the square root operation inherent in calculating the length of vector **s**, we can simply square the entire equation. So Equation (13.1) becomes

$$\begin{aligned} \mathbf{s} &= \mathbf{c}_1 - \mathbf{c}_2; \\ |\mathbf{s}|^2 &= \mathbf{s} \cdot \mathbf{s}; \\ \text{if } |\mathbf{s}|^2 &\leq (r_1 + r_2)^2, \text{then spheres intersect.} \end{aligned}$$

13.3.5.3 The Separating Axis Theorem

Most collision detection systems make heavy use of a theorem known as the *separating axis theorem* (http://en.wikipedia.org/wiki/Separating_axis _theorem). It states that if an axis can be found along which the *projections* of two convex shapes do not overlap, then we can be certain that the two shapes do not intersect at all. If such an axis does not exist *and* the shapes are convex, then we know for certain that they do intersect. (If the shapes are concave, then they may not be interpenetrating despite the lack of a separating axis. This is one reason why we tend to favor convex shapes in collision detection.)

This theorem is easiest to visualize in two dimensions. Intuitively, it says that if a line can be found, such that object A is entirely on one side of the line and object B is entirely on the other side, then objects A and B do not overlap. Such a line is called a *separating line*, and it is always *perpendicular* to the separating axis. So once we've found a separating line, it's a lot easier to convince ourselves that the theory is in fact correct by looking at the projections of our shapes onto the axis that is perpendicular to the separating line.

The projection of a two-dimensional *convex* shape onto an axis acts like the shadow that the object would leave on a thin wire. It is always a line segment, lying on the axis, that represents the maximum extents of the object in the direction of the axis. We can also think of a projection as a minimum and maximum coordinate along the axis, which we can write as the fully closed interval $[c_{\text{min}}, c_{\text{max}}]$. As you can see in Figure 13.10, when a separating line exists between two shapes, their projections do not overlap along the separating axis. However, the projections may overlap along other, non-separating axes.

In three dimensions, the separating line becomes a separating plane, but the separating axis is still an axis (i.e., an infinite line). Again, the projection of a three-dimensional *convex* shape onto an axis is a line segment, which we can represent by the fully closed interval $[c_{\text{min}}, c_{\text{max}}]$.

Some types of shapes have properties that make the potential separating axes obvious. To detect intersections between two such shapes A and B, we can project the shapes onto each potential separating axis in turn and then check whether or not the two projection intervals, $[c^{\text{A}}_{\text{min}}, c^{\text{A}}_{\text{max}}]$ and $[c^{\text{B}}_{\text{min}}, c^{\text{B}}_{\text{max}}]$,

图 13.10。两个形状在分离轴上的投影始终是两条不相交的线段。这些相同形状在非分离轴上的投影不一定不相交。如果不存在分离轴，则形状相交。

是不相交的（即不重叠）。用数学术语来说，如果 c A max < c B min 或 c B max < c A min ，则区间是不相交的。如果沿着其中一个潜在分离轴的投影区间是不相交的，那么我们就找到了一个分离轴，并且我们知道这两个形状不相交。

这一原理的一个实际应用示例是球体对球体测试。如果两个球体不相交，则与连接球体中心点的线段平行的轴始终是有效的分离轴（尽管可能存在其他分离轴，具体取决于两个球体的距离）。为了形象地理解这一点，请考虑两个球体即将接触但尚未接触时的极限。在这种情况下，唯一的分离轴是与球体中心点到中心点的线段平行的轴。随着球体逐渐分离，我们可以将分离轴向任意方向旋转得越来越远。如图 13.11 所示。

13.3.5.4 AABB 与 AABB

要判断两个 AABB 是否相交，我们可以再次应用分离轴定理。由于两个 AABB 的面都保证与一组公共坐标轴平行，因此如果存在分离轴，它将是这三个坐标轴之一。

因此，为了测试两个 AABB（我们称之为 A 和 B）之间的交集，我们只需分别检查两个盒子沿每个轴的最小和最大坐标。沿 x 轴，我们有两个区间

Figure 13.10. The projections of two shapes onto a separating axis are always two disjoint line segments. The projections of these same shapes onto a non-separating axis are not necessarily disjoint. If no separating axis exists, the shapes intersect.

are disjoint (i.e., do not overlap). In math terms, the intervals are disjoint if $c^{A}_{max} < c^{B}_{min}$ or if $c^{B}_{max} < c^{A}_{min}$. If the projection intervals along one of the potential separating axes are disjoint, then we've found a separating axis, and we know the two shapes do not intersect.

One example of this principle in action is the sphere-versus-sphere test. If two spheres do not intersect, then the axis parallel to the line segment joining the spheres' center points will always be a valid separating axis (although other separating axes may exist, depending on how far apart the two spheres are). To visualize this, consider the limit when the two spheres are just about to touch but have not yet come into contact. In that case, the *only* separating axis is the one parallel to the center-to-center line segment. As the spheres move apart, we can rotate the separating axis more and more in either direction. This is shown in Figure 13.11.

13.3.5.4 AABB versus AABB

To determine whether two AABBs are intersecting, we can again apply the separating axis theorem. The fact that the faces of both AABBs are guaranteed to lie parallel to a common set of coordinate axes tells us that if a separating axis exists, it will be one of these three coordinate axes.

So, to test for intersections between two AABBs, which we'll call A and B, we merely inspect the minimum and maximum coordinates of the two boxes along each axis independently. Along the x-axis, we have the two intervals

图 13.11 当两个球体之间的距离无穷小时，唯一的分离轴与两个球体中心点形成的线段平行。

[x A min , x A max] 和 [x B min , x B max]，并且我们在 y 轴和 z 轴上分别有对应的区间。如果区间沿所有三个轴都重叠，则两个 AABB 相交——在其他所有情况下，它们不相交。图 13.12 展示了相交和不相交 AABB 的示例（为了便于说明，简化为二维）。有关 AABB 碰撞的深入讨论，请参阅 http://www.gamasutra.com/features/20000203/lander_01.htm。

13.3.5.5 检测凸碰撞：GJK 算法

存在一种非常有效的算法，用于检测任意凸多面体（即二维凸多边形或三维凸多面体）之间的交点。该算法被称为 GJK 算法，以其发明者密歇根大学的 E. G. Gilbert、D.W. Johnson 和 S.S. Keerthi 的名字命名。目前已有许多关于该算法及其变体的论文。

图 13.12 相交和非相交 AABB 的二维示例。请注意，尽管第二对 AABB 沿 x 轴相交，但它们沿 y 轴不相交。

Figure 13.11. When two spheres are an infinitesimal distance apart, the only separating axis lies parallel to the line segment formed by the two spheres' center points.

$[x^{A}_{min}, x^{A}_{max}]$ and $[x^{B}_{min}, x^{B}_{max}]$, and we have corresponding intervals for the y- and z-axes. If the intervals overlap along *all three axes*, then the two AABBs are intersecting—in all other cases, they are not. Examples of intersecting and non-intersecting AABBs are shown in Figure 13.12 (simplified to two dimensions for the purposes of illustration). For an in-depth discussion of AABB collision, see http://www.gamasutra.com/features/20000203/lander_01.htm.

13.3.5.5 Detecting Convex Collisions: The GJK Algorithm

A very efficient algorithm exists for detecting intersections between arbitrary convex *polytopes* (i.e., convex polygons in two dimensions, or convex polyhedra in three dimensions). It is known as the GJK algorithm, named after its inventors, E. G. Gilbert, D. W. Johnson and S. S. Keerthi of the University of Michigan. Many papers have been written on the algorithm and its vari-

Figure 13.12. A two-dimensional example of intersecting and non-intersecting AABBs. Notice that even though the second pair of AABBs are intersecting along the x-axis, they are not intersecting along the y-axis.

蚂蚁算法的资料非常丰富，包括原始论文（http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?&arnumber=2083）、Christer Ericson 的精彩 SIGGRAPH PowerPoint 演示文稿（http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt）以及 Gino van den Bergen 的另一个精彩 PowerPoint 演示文稿（www.laas.fr/~nic/MOVIE/Workshop/Slides/Gino.vander.Bergen.ppt）。然而，对该算法最容易理解（也是最有趣）的描述可能是 Casey Muratori 的教学视频“实现 GJK”，可在线观看：http://mollyrocket.com/849。由于这些描述非常精彩，我在这里只让您大致了解该算法的精髓，然后引导您访问 Molly Rocket 网站和上面引用的其他参考资料以获取更多详细信息。

GJK 算法依赖于一种称为闵可夫斯基差分的几何运算。这个运算听起来很奇特，实际上非常简单：我们取形状 B 中的每个点，然后将其与形状 A 中的每个点成对地减去。得到的点集 $\{(A_i - B_j)\}$ 就是闵可夫斯基差分。

闵可夫斯基差分的实用之处在于，当应用于两个凸形时，当且仅当这两个形状相交时，它才包含原点。该命题的证明略微超出了我们的范围，但我们可以通过以下方式直观地理解它成立的原因：当我们说两个形状 A 和 B 相交时，我们实际上指的是 A 内部的点也位于 B 内部。在用 A 中的每个点减去 B 中的每个点的过程中，我们期望最终会碰到位于两个形状内的公共点之一。一个点减去它本身的值全为零，因此，当（且仅当）球体 A 和球体 B 存在公共点时，闵可夫斯基差分才包含原点。如图 13.13 所示。

两个凸形状的闵可夫斯基差本身也是一个凸形状。我们只关心闵可夫斯基差分的凸包，而不是所有内部点。GJK 的基本过程是尝试找到一个四面体（即由三角形组成的四边形），它位于闵可夫斯基差分的凸包上，并且包含原点。如果能找到一个四面体，则这两个形状相交；如果找不到，则它们不相交。

四面体只是被称为单纯形的几何物体的一种情况。但别被这个名字吓到——单纯形只是点的集合。单点单纯形是一个点，两点单纯形是一条线段，三点单纯形是一个三角形，四点单纯形是一个四面体（见图 13.14）。

GJK 是一种迭代算法，它从位于 Minkowski 差分壳内任意位置的单点单纯形开始。然后，它尝试构建可能包含原点的高阶单纯形。在循环的每次迭代中，我们都会检查当前拥有的单纯形，并确定

ants, including the original paper (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?&arnumber=2083), an excellent SIGGRAPH PowerPoint presentation by Christer Ericson (http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt) and another great PowerPoint presentation by Gino van den Bergen (www.laas.fr/~nic/MOVIE/Workshop/Slides/Gino.vander.Bergen.ppt). However, the easiest-to-understand (and most entertaining) description of the algorithm is probably Casey Muratori's instructional video entitled "Implementing GJK," available online at http://mollyrocket.com/849. Because these descriptions are so good, I'll just give you a feel for the essence of the algorithm here and then direct you to the Molly Rocket website and the other references cited above for additional details.

The GJK algorithm relies on a geometric operation known as the *Minkowski difference*. This fancy-sounding operation is really quite simple: We take every point that lies within shape B and subtract it pairwise from every point inside shape A. The resulting set of points $\{(\mathbf{A}_i - \mathbf{B}_j)\}$ is the Minkowski difference.

The useful thing about the Minkowski difference is that, when applied to two convex shapes, it will *contain the origin* if and only if those two shapes intersect. Proof of this statement is a bit beyond our scope, but we can intuit why it is true by remembering that when we say two shapes A and B intersect, we really mean that there are points within A that are *also* within B. During the process of subtracting every point in B from every point in A, we would expect to eventually hit one of those shared points that lies within both shapes. A point minus itself is all zeros, so the Minkowski difference will contain the origin if (and only if) sphere A and sphere B have points in common. This is illustrated in Figure 13.13.

The Minkowski difference of two convex shapes is itself a convex shape. All we care about is the *convex hull* of the Minkowski difference, not all of the interior points. The basic procedure of GJK is to try to find a *tetrahedron* (i.e., a four-sided shape made out of triangles) that lies on the convex hull of the Minkwoski difference and that encloses the origin. If one can be found, then the shapes intersect; if one cannot be found, then they don't.

A tetrahedron is just one case of a geometrical object known as a *simplex*. But don't let that name scare you—a simplex is just a collection of points. A single-point simplex is a point, a two-point simplex is a line segment, a three-point simplex is a triangle and a four-point simplex is a tetrahedron (see Figure 13.14).

GJK is an iterative algorithm that starts with a one-point simplex lying anywhere within the Minkowski difference hull. It then attempts to build higher-order simplexes that might potentially contain the origin. During each iteration of the loop, we take a look at the simplex we currently have and determine

图 13.13 两个相交凸形状的闵可夫斯基差包含原点，但两个不相交形状的闵可夫斯基差不包含原点。

原点相对于它的方向。然后，我们在该方向上找到 Minkowski 差分的支撑顶点 - 即，凸包中在我们当前前进方向上最接近原点的顶点。我们将这个新点添加到单纯形中，创建一个高阶单纯形（即，一个点变成线段，一个线段变成三角形，一个三角形变成四面体）。如果添加这个新点导致单纯形围绕原点，那么我们就完成了 - 我们就知道这两个形状相交。另一方面，如果我们无法找到比当前单纯形更靠近原点的支撑顶点，那么我们就知道我们永远无法到达那里，这意味着这两个形状不相交。这个想法如图 13.15 所示。

要真正理解 GJK 算法，你需要查看我之前提到的论文和视频。但希望这些描述能激发你深入研究的兴趣。或者，至少，你能留下深刻的印象。

图 13.14. 包含一个、两个、三个和四个点的单纯形。

Figure 13.13. The Minkowski difference of two intersecting convex shapes contains the origin, but the Minkowski difference of two non-intersecting shapes does not.

in which direction the origin lies relative to it. We then find a *supporting vertex* of the Minkowski difference in that direction—i.e., the vertex of the convex hull that is closest to the origin in the direction we're currently going. We add that new point to the simplex, creating a higher-order simplex (i.e., a point becomes a line segment, a line segment becomes a triangle and a triangle becomes a tetrahedron). If the addition of this new point causes the simplex to surround the origin, then we're done—we know the two shapes intersect. On the other hand, if we are unable to find a supporting vertex that is closer to the origin than the current simplex, then we know that we can never get there, which implies that the two shapes do *not* intersect. This idea is illustrated in Figure 13.15.

To truly understand the GJK algorithm, you'll need to check out the papers and video I referenced previously. But hopefully this description will whet your appetite for deeper investigation. Or, at the very least, you can impress

Figure 13.14. Simplexes containing one, two, three and four points.

图 13.15 在 GJK 算法中，如果向当前单纯形添加一个点会创建一个包含原点的形状，则我们就知道这两个形状相交；如果没有支撑顶点可以使单纯形更靠近原点，则这两个形状不相交。

在聚会上提到"GJK"这个名字，就能让你的朋友们记住你。（除非你真的懂算法，否则千万别在面试时这么做！）

13.3.5.6 其他形状-形状组合

我们不会在这里介绍任何其他形状-形状相交组合，因为它们在其他文本（例如 [14]、[48] 和 [11]）中已有详细介绍。然而，这里需要认识到的关键点是形状-形状组合的数量非常大。事实上，对于 N 种形状类型，所需的成对测试次数为 O(N2)。碰撞引擎的大部分复杂性源于它必须处理的相交案例数量。这就是碰撞引擎作者通常试图限制原始类型数量的原因之一——这样做可以大大减少碰撞检测器必须处理的案例数量。（这也是 GJK 受欢迎的原因——它可以一次性处理所有凸形状类型之间的碰撞检测。形状类型之间唯一的不同之处在于算法中使用的支持函数。）还有一个实际问题，即如何实现代码，以便在给定两个要测试的任意形状的情况下选择合适的碰撞测试函数。许多碰撞引擎使用双重调度方法 (http://en.wikipedia.org/wiki/Double_dispatch)。在单调度（即虚函数）中，单个对象的类型用于确定在运行时应调用特定抽象函数的哪个具体实现。双重调度将虚函数概念扩展到两种对象类型。它可以通过一个二维函数查找表来实现，该查找表以被测试的两个对象的类型为键。它也可以这样实现：基于对象 A 类型的虚函数调用基于对象 B 类型的第二个虚函数。

让我们来看一个真实的例子。Havok 使用称为碰撞代理（派生自 hkpCollisionAgent 的类）的对象来处理特定的

Figure 13.15. In the GJK algorithm, if adding a point to the current simplex creates a shape that contains the origin, we know the shapes intersect; if there is no supporting vertex that will bring the simplex any closer to the origin, then the shapes do not intersect.

your friends by dropping the name "GJK" at parties. (Just don't try this at job interviews unless you really do understand the algorithm!)

13.3.5.6 Other Shape-Shape Combinations

We won't cover any of the other shape-shape intersection combinations here, as they are covered well in other texts such as [14], [48] and [11]. The key point to recognize here, however, is that the *number* of shape-shape combinations is very large. In fact, for N shape types, the number of pairwise tests required is $O(N^2)$. Much of the complexity of a collision engine arises because of the sheer number of intersection cases it must handle. This is one reason why the authors of collision engines usually try to limit the number of primitive types—doing so drastically reduces the number of cases the collision detector must handle. (This is also why GJK is popular—it handles collision detection between *all* convex shape types in one fell swoop. The only thing that differs from shape type to shape type is the *support function* used in the algorithm.)

There's also the practical matter of how to implement the code that selects the appropriate collision-testing function given two arbitrary shapes that are to be tested. Many collision engines use a *double dispatch* method (http://en.wikipedia.org/wiki/Double_dispatch). In single dispatch (i.e., virtual functions), the type of a single object is used to determine which concrete implementation of a particular abstract function should be called at runtime. Double dispatch extends the virtual function concept to two object types. It can be implemented via a two-dimensional function look-up table keyed by the types of the two objects being tested. It can also be implemented by arranging for a virtual function based on the type of object A to call a second virtual function based on the type of object B.

Let's take a look at a real-world example. Havok uses objects known as *collision agents* (classes derived from `hkpCollisionAgent`) to handle specific

图 13.16。小型、快速移动的物体可能会在碰撞世界的连续快照之间在其运动路径中留下间隙，这意味着可能会完全错过碰撞。

交叉测试用例。具体的代理类包括 hkpSphereSphere Agent 、 hkpSphereCapsule Agent 、 hkpGskConvexConvexAgent 等等。这些代理类型由一个二维调度表引用，该调度表由 hkpCollisionDispatcher 类管理。正如您所料，调度器的工作是根据一对待进行碰撞测试的可碰撞对象，高效地查找合适的代理，然后调用它，并将这两个可碰撞对象作为参数传递。

13.3.5.7 检测运动物体之间的碰撞

到目前为止，我们仅考虑了静止物体之间的静态相交测试。当物体移动时，这会带来一些额外的复杂性。游戏中的运动通常以离散的时间步长进行模拟。因此，一种简单的方法是将每个刚体在每个时间步长上的位置和方向视为静止的，并对碰撞世界的每个“快照”进行静态相交测试。只要物体的移动速度相对于其尺寸而言不是太快，这种方法就有效。事实上，这种方法效果非常好，以至于许多碰撞/物理引擎（包括 Havok）都默认使用这种方法。

然而，这种技术对于快速移动的小物体不起作用。想象一下，一个物体移动得如此之快，以至于它在时间步长之间移动的距离大于其自身尺寸（沿行进方向测量）。如果我们叠加碰撞世界的两个连续快照，我们会注意到快速移动物体在两个快照中的图像之间存在间隙。如果另一个物体恰好位于这个间隙内，我们将完全错过与它的碰撞。如图 13.16 所示，这个问题被称为“子弹穿纸”问题，也称为“隧道效应”。以下部分介绍了一些解决此问题的常用方法。

扫描形状

避免隧道效应的一种方法是利用扫掠形状。扫掠形状

Figure 13.16. A small, fast-moving object can leave gaps in its motion path between consecutive snapshots of the collision world, meaning that collisions might be missed entirely.

intersection test cases. Concrete agent classes include `hkpSphereSphere-Agent`, `hkpSphereCapsuleAgent`, `hkpGskConvexConvexAgent` and so on. The agent types are referenced by what amounts to a two-dimensional dispatch table, managed by the class `hkpCollisionDispatcher`. As you'd expect, the dispatcher's job is to efficiently look up the appropriate agent given a pair of collidables that are to be collision-tested and then call it, passing the two collidables as arguments.

13.3.5.7 Detecting Collisions between Moving Bodies

Thus far, we've considered only static intersection tests between stationary objects. When objects move, this introduces some additional complexity. Motion in games is usually simulated in discrete time steps. So one simple approach is to treat the positions and orientations of each rigid body as stationary at each time step and use static intersection tests on each "snapshot" of the collision world. This technique works as long as objects aren't moving too fast relative to their sizes. In fact, it works so well that many collision/physics engines, including Havok, use this approach by default.

However, this technique breaks down for small, fast-moving objects. Imagine an object that is moving so fast that it covers a distance *larger than its own size* (measured in the direction of travel) between time steps. If we were to overlay two consecutive snapshots of the collision world, we'd notice that there is now a gap between the fast-moving object's images in the two snapshots. If another object happens to lie within this gap, we'll miss the collision with it entirely. This problem, illustrated in Figure 13.16, is known as the "bullet through paper" problem, also known as "tunneling." The following sections describe a number of common ways to overcome this problem.

Swept Shapes

One way to avoid tunneling is to make use of *swept shapes*. A swept shape

是指形状随时间从一个点移动到另一个点而形成的新形状。例如，扫描球体是一个胶囊，扫描三角形是一个三棱柱（参见图 13.17）。

我们不必测试碰撞世界的静态快照是否相交，而是可以测试扫描形状，该扫描形状是通过将形状从上一个快照中的位置和方向移动到当前快照中的位置和方向而形成的。这种方法相当于在快照之间对可碰撞体的运动进行线性插值，因为我们通常沿着线段从一个快照扫描到另一个快照。

当然，线性插值可能不是快速移动的可碰撞物体运动的良好近似。如果可碰撞物体沿着曲线路径运动，那么理论上我们应该沿着该曲线路径扫描其形状。不幸的是，沿着曲线扫描的凸形状本身并不是凸的，因此这会使我们的碰撞测试更加复杂且计算量巨大。

此外，如果我们扫描的凸形正在旋转，那么即使沿线段扫描，最终的扫描形状也不一定凸。如图 13.18 所示，我们总是可以通过线性外推来自前一个和当前快照的形状的极端特征来形成凸形——但最终的凸形不一定能准确表示形状在时间步长内的实际变化。换句话说，线性插值通常不适用于旋转形状。因此，除非我们的形状不允许旋转，否则扫描形状的相交测试将比基于静态快照的测试更加复杂且计算量巨大。

扫描形状是一种有用的技术，可以确保碰撞世界状态的静态快照之间不会错过碰撞。然而，当线性插值曲线路径或

图 13.17 扫描球体是一个胶囊；扫描三角形是一个三棱柱体。

is a new shape formed by the motion of a shape from one point to another over time. For example, a swept sphere is a capsule, and a swept triangle is a triangular prism (see Figure 13.17).

Rather than testing static snapshots of the collision world for intersections, we can test the swept shapes formed by moving the shapes from their positions and orientations in the previous snapshot to their positions and orientations in the current snapshot. This approach amounts to *linearly interpolating* the motion of the collidables between snapshots, because we generally sweep the shapes along line segments from snapshot to snapshot.

Of course, linear interpolation may not be a good approximation of the motion of a fast-moving collidable. If the collidable is following a curved path, then theoretically we should sweep its shape along that curved path. Unfortunately, a convex shape that has been swept along a curve is not itself convex, so this can make our collision tests much more complex and computationally intensive.

In addition, if the convex shape we are sweeping is rotating, the resulting swept shape is not necessarily convex, even when it is swept along a line segment. As Figure 13.18 shows, we *can* always form a convex shape by linearly extrapolating the extreme features of the shapes from the previous and current snapshots—but the resulting convex shape is not necessarily an accurate representation of what the shape really would have done over the time step. Put another way, a linear interpolation is not appropriate in general for rotating shapes. So unless our shapes are not permitted to rotate, intersection testing of swept shapes becomes much more complex and computationally intensive than its static snapshot-based counterpart.

Swept shapes can be a useful technique for ensuring that collisions are not missed between static snapshots of the collision world state. However, the results are generally inaccurate when linearly interpolating curved paths or

Figure 13.17. A swept sphere is a capsule; a swept triangle is a triangular prism.

2. 大多数游戏世界包含大量物体，并且随着物体数量的增加，所需的相交测试次数也会迅速增长。

为了检测 n 个对象之间的相交，最常用的暴力算法是测试所有可能的对象对，从而产生一个 $O(n^2)$ 的算法。然而，实际应用中会使用更高效的算法。碰撞引擎通常采用某种形式的空间哈希 (http://bit.ly/1fLtX1D)、空间细分或分层边界体积，以减少必须执行的相交测试次数。

13.3.6.1 时间一致性

一种常见的优化技术是利用时间一致性（也称为帧间一致性）。当可碰撞物体以合理的速度移动时，它们的位置和方向通常在各个时间步长上非常相似。我们通常可以通过在多个时间步长上缓存结果来避免每帧重新计算某些信息。例如，在 Havok 中，碰撞代理（hkpCollisionAgent）通常在帧之间保持持久性，只要相关可碰撞物体的运动没有使这些计算失效，它们就可以重用之前时间步长的计算结果。

13.3.6.2 空间分区

空间划分的基本思想是通过将空间划分成多个较小的区域，大大减少需要进行相交检查的可碰撞体数量。如果我们能够（以一种低成本的方式）确定一对可碰撞体不位于同一区域，那么我们就不需要对它们进行更详细的相交测试。

各种层次化划分方案，例如八叉树、二叉空间划分树 (BSP)、k-d 树或球体树，均可用于细分空间，以优化碰撞检测。这些树以不同的方式细分空间，但它们都以层次化的方式进行，从树根处的粗略细分开始，进一步细分每个区域，直到获得足够细粒度的区域。然后可以遍历树以查找和测试可能发生碰撞的物体组是否发生实际相交。由于树对空间进行了划分，我们知道，当我们遍历树的一个分支时，该分支中的物体不会与其他兄弟分支中的物体发生碰撞。

2. Most game worlds contain a large number of objects, and the number of intersection tests required grows rapidly as the number of objects increases.

To detect intersections between n objects, the brute-force technique would be to test every possible pair of objects, yielding an $O(n^2)$ algorithm. However, much more efficient algorithms are used in practice. Collision engines typically employ some form of spatial hashing (http://bit.ly/1fLtX1D), spatial subdivision or hierarchical bounding volumes in order to reduce the number of intersection tests that must be performed.

13.3.6.1 Temporal Coherency

One common optimization technique is to take advantage of *temporal coherency*, also known as *frame-to-frame coherency*. When collidables are moving at reasonable speeds, their positions and orientations are usually quite similar from time step to time step. We can often avoid recalculating certain kinds of information every frame by caching the results across multiple time steps. For example, in Havok, collision agents (`hkpCollisionAgent`) are usually persistent between frames, allowing them to reuse calculations from previous time steps as long as the motion of the collidables in question hasn't invalidated those calculations.

13.3.6.2 Spatial Partitioning

The basic idea of spatial partitioning is to greatly reduce the number of collidables that need to be checked for intersection by dividing space into a number of smaller regions. If we can determine (in an inexpensive manner) that a pair of collidables do not occupy the same region, then we needn't perform more-detailed intersection tests on them.

Various hierarchical partitioning schemes, such as octrees, binary space partitioning trees (BSPs), kd-trees or sphere trees, can be used to subdivide space for the purposes of collision detection optimization. These trees subdivide space in different ways, but they all do so in a hierarchical fashion, starting with a gross subdivision at the root of the tree and further subdividing each region until sufficiently fine-grained regions have been obtained. The tree can then be walked in order to find and test groups of potentially colliding objects for actual intersections. Because the tree partitions space, we know that when we traverse down one branch of the tree, the objects in that branch cannot be colliding with objects in other sibling branches.

13.3.6.3 宽相、中相和窄相

Havok 使用三层方法来修剪每个时间步骤中需要进行碰撞测试的可碰撞物体集。

- 首先，使用粗略的 AABB 测试来确定哪些可碰撞物体可能相交。这被称为广义相碰撞检测。
- 其次，测试复合形状的粗边界体积。
  这被称为中间阶段碰撞检测。例如，在一个由三个球体组成的复合形状中，边界体可能是第四个更大的球体，它将其他球体包裹起来。复合形状可能包含其他复合形状，因此通常情况下，复合可碰撞体具有边界体层级结构。中间阶段会遍历此层级结构，以寻找可能相交的子形状。
- 最后，对可碰撞体的各个图元进行相交测试。
  这被称为窄相碰撞检测。

清除和修剪算法

在所有主流碰撞/物理引擎（例如 Havok、ODE、PhysX）中，广义碰撞检测都采用一种称为“扫描和剪枝”的算法（http://en.wikipedia.org/wiki/Sweep_and_prune）。其基本思想是沿三个主轴对可碰撞物体的 AABB 的最小和最大维度进行排序，然后遍历排序后的列表来检查重叠的 AABB。扫描和剪枝算法可以利用帧间一致性（参见第 13.3.6.1 节）将 O(nlogn) 的排序操作缩短为预期的 O(n) 运行时间。帧一致性还可以帮助在物体旋转时更新 AABB。

13.3.7 碰撞查询

碰撞检测系统的另一个职责是回答关于游戏世界中碰撞体积的假设性问题。例如：

- 如果一颗子弹从玩家的武器中沿给定方向射出，它会击中的第一个目标是什么（如果有的话）？
- 车辆能否从 A 点移动到 B 点而不撞到途中的任何东西？
- 找到角色给定半径范围内的所有敌方物体。

一般来说，这样的操作被称为碰撞查询。

13.3.6.3 Broad Phase, Midphase and Narrow Phase

Havok uses a three-tiered approach to prune the set of collidables that need to be tested for collisions during each time step.

- First, gross AABB tests are used to determine which collidables are potentially intersecting. This is known as *broad phase* collision detection.
- Second, the coarse bounding volumes of compound shapes are tested. This is known as *midphase* collision detection. For example, in a compound shape composed of three spheres, the bounding volume might be a fourth, larger sphere that encloses the other spheres. A compound shape may contain other compound shapes, so in general a compound collidable has a bounding volume hierarchy. The midphase traverses this hierarchy in search of subshapes that are potentially intersecting.
- Finally, the collidables' individual primitives are tested for intersection. This is known as *narrow phase* collision detection.

The Sweep and Prune Algorithm

In all of the major collision/physics engines (e.g., Havok, ODE, PhysX), broad phase collision detection employs an algorithm known as *sweep and prune* (http://en.wikipedia.org/wiki/Sweep_and_prune). The basic idea is to sort the minimum and maximum dimensions of the collidables' AABBs along the three principal axes, and then check for overlapping AABBs by traversing the sorted lists. Sweep and prune algorithms can make use of frame-to-frame coherency (see Section 13.3.6.1) to reduce an $O(n \log n)$ sort operation to an expected $O(n)$ running time. Frame coherency can also aid in the updating of AABBs when objects rotate.

13.3.7 Collision Queries

Another responsibility of the collision detection system is to answer hypothetical questions about the collision volumes in the game world. Examples include the following:

- If a bullet travels from the player's weapon in a given direction, what is the first target it will hit, if any?
- Can a vehicle move from point A to point B without striking anything along the way?
- Find all enemy objects within a given radius of a character.

In general, such operations are known as *collision queries*.

最常见的查询类型是碰撞投射，有时简称为投射。（“投射”的另外两个常见同义词是“追踪”和“探测”。）投射用于确定如果将一个假想的物体放入碰撞世界并沿着射线或线段移动，它会撞击什么（如果有的话）。投射不同于常规的碰撞检测操作，因为被投射的实体实际上并不在碰撞世界中——它不能以任何方式影响世界中的其他物体。这就是为什么我们说碰撞投射回答了关于世界中可碰撞物体的假想问题。

13.3.7.1 射线投射

最简单的碰撞投射类型是射线投射，尽管这个名字实际上有点用词不当。我们实际上投射的是一条有向线段——换句话说，我们的投射总是有一个起点（p0）和一个终点（p1）。投射的线段会与碰撞世界中的可碰撞物体进行测试。如果它与其中任何一个相交，则返回接触点。

射线投射系统通常通过起点 p 0 和一个增量向量 d 来描述线段，增量向量 d 与 p 0 相加，得到终点 p 1 。该线段上的任意点都可以通过以下参数方程找到，其中参数 t 可以在 0 和 1 之间变化：

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}, \qquad t \in [0, 1].$$

显然，p 0 = p(0)，p 1 = p(1)。此外，线段上的任何接触点都可以通过指定与该接触点对应的参数 t 的值来唯一描述。大多数射线投射 API 将其接触点返回为“t 值”，或者允许通过额外的函数调用将接触点转换为其对应的 t 值。

大多数碰撞检测系统能够返回最早的接触点，即距离 p0 最近且对应于 t 最小值的接触点。一些系统还能够返回与射线或线段相交的所有可碰撞实体的完整列表。每次接触返回的信息通常包括 t 值、被碰撞的可碰撞实体的某种唯一标识符，以及可能的其他信息，例如接触点的表面法线或被碰撞的形状或表面的其他相关属性。下图展示了一种可能的接触点数据结构。

```
struct RayCastContact
{
    F32    m_t;            // the t value for this
                           // contact
```


The most common kind of query is a *collision cast*, sometimes just called a *cast*. (The terms *trace* and *probe* are other common synonyms for “cast.”) A cast determines what, if anything, a hypothetical object would hit if it were to be placed into the collision world and moved along a ray or line segment. Casts are different from regular collision detection operations because the entity being cast is not really in the collision world—it cannot affect the other objects in the world in any way. This is why we say that a collision cast answers *hypothetical* questions about the collidables in the world.

13.3.7.1 Ray Casting

The simplest type of collision cast is a *ray cast*, although this name is actually a bit of a misnomer. What we're really casting is a *directed line segment*—in other words, our casts always have a start point ($\mathbf{p}_0$) and an end point ($\mathbf{p}_1$). The cast line segment is tested against the collidable objects in the collision world. If it intersects any of them, the contact point or points are returned.

Ray casting systems typically describe the line segment via its start point $\mathbf{p}_0$ and a delta vector $\mathbf{d}$ that, when added to $\mathbf{p}_0$, yields the end point $\mathbf{p}_1$. Any point on this line segment can be found via the following *parametric equation*, where the parameter t is permitted to vary between zero and one:

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}, \qquad t \in [0, 1].$$

Clearly, $\mathbf{p}_0 = \mathbf{p}(0)$ and $\mathbf{p}_1 = \mathbf{p}(1)$. In addition, any contact point along the segment can be uniquely described by specifying the value of the parameter t corresponding to the contact. Most ray casting APIs return their contact points as “t values,” or they permit a contact point to be converted into its corresponding t by making an additional function call.

Most collision detection systems are capable of returning the *earliest contact*—i.e., the contact point that lies closest to $\mathbf{p}_0$ and corresponds to the smallest value of t. Some systems are also capable of returning a complete list of all collidables that were intersected by the ray or line segment. The information returned for each contact typically includes the t value, some kind of unique identifier for the collidable entity that was hit, and possibly other information such as the surface normal at the point of contact or other relevant properties of the shape or surface that was struck. One possible contact point data structure is shown below.

```
struct RayCastContact
{
    F32    m_t;            // the t value for this
                           // contact
```

```
    U32    m_collidableId; // which collidable did we
                           // hit?

    Vector m_normal;       // surface normal at
                           // contact pt.

    // other information...
};
```

射线投射的应用

射线投射在游戏中被广泛使用。例如，我们可能想询问碰撞系统角色 A 是否与角色 B 有直接的视线。为了确定这一点，我们只需从角色 A 的眼睛投射一条有向线段到角色 B 的胸部。如果射线击中角色 B，我们就知道 A 可以“看到” B。但如果射线在到达角色 B 之前击中了其他物体，我们就知道视线被该物体遮挡了。射线投射可用于武器系统（例如，确定子弹命中）、玩家机制（例如，确定角色脚下是否有坚实的地面）、AI 系统（例如，视线检查、瞄准、移动查询等）、车辆系统（例如，定位并将车辆轮胎与地形对齐）等等。

13.3.7.2 成型铸造

另一个常见的问题是询问碰撞系统，一个假想的凸形在碰到某个实体之前能够沿着有向线段行进多远。当投射的体积是球体时，这被称为球体投射，或者一般的形状投射。（Havok 称之为线性投射。）与射线投射一样，形状投射通常通过指定起点 p 0 、行进距离 d 以及我们希望投射的形状的类型、尺寸和方向来描述。

铸造凸形时需要考虑两种情况。

1. 铸造形状已经相互穿透或接触至少一个其他可碰撞物体，从而阻止其移离其起始位置。

2. 铸造形状在其起始位置不与任何其他物体相交，因此它可以自由地沿着其路径移动非零距离。

在第一种情况下，碰撞系统通常会报告铸件形状与其最初穿透的所有可碰撞物体之间的接触。这些接触可能位于铸件形状内部，也可能位于其表面，如图 1 3.19 所示。


```
    U32    m_collidableId; // which collidable did we
                           // hit?

    Vector m_normal;       // surface normal at
                           // contact pt.

    // other information...
};
```

Applications of Ray Casts

Ray casts are used heavily in games. For example, we might want to ask the collision system whether character A has a direct line of sight to character B. To determine this, we simply cast a directed line segment from the eyes of character A to the chest of character B. If the ray hits character B, we know that A can “see” B. But if the ray strikes some other object *before* reaching character B, we know that the line of sight is being blocked by that object. Ray casts are used by weapon systems (e.g., to determine bullet hits), player mechanics (e.g., to determine whether or not there is solid ground beneath the character’s feet), AI systems (e.g., line of sight checks, targeting, movement queries, etc.), vehicle systems (e.g., to locate and snap the vehicle’s tires to the terrain) and so on.

13.3.7.2 Shape Casting

Another common query involves asking the collision system how far an imaginary convex shape would be able to travel along a directed line segment before it hits something solid. This is known as a *sphere cast* when the volume being cast is a sphere, or a *shape cast* in general. (Havok calls them *linear casts*.) As with ray casts, a shape cast is usually described by specifying the start point $\mathbf{p}_0$, the distance to travel $\mathbf{d}$ and of course the type, dimensions and orientation of the shape we wish to cast.

There are two cases to consider when casting a convex shape.

1. The cast shape is already interpenetrating or contacting at least one other collidable, preventing it from moving away from its starting location.

2. The cast shape is not intersecting with anything else at its starting location, so it is free to move a nonzero distance along its path.

In the first scenario, the collision system typically reports the contact(s) between the cast shape and all of the collidables with which it is initially interpenetrating. These contacts might be *inside* the cast shape or on its *surface*, as shown in Figure 13.19.

图 13.19. 开始穿透的铸造球体将无法移动，并且可能许多接触点通常位于铸造形状内部。

在第二种情况下，形状可以沿着线段移动一段非零距离后再撞击到某物。假设它撞击到某物，通常只会撞击一个可碰撞体。但是，如果轨迹恰到好处，铸造形状有可能同时撞击多个可碰撞体。当然，如果撞击的可碰撞体是非凸多边形汤，铸造形状最终可能会同时接触到多边形汤的多个部分。我们可以肯定地说，无论铸造的是哪种凸多边形，铸造都有可能产生多个接触点。在这种情况下，接触点始终位于铸造形状的表面，而不是内部（因为我们知道铸造形状在开始行程时没有穿透任何东西）。这种情况如图 13.20 所示。

与射线投射一样，一些形状投射 API 仅报告投射形状所经历的最早接触，而其他形状投射 API 则允许形状继续沿着其假设路径行进，并返回其在旅途中遇到的所有接触。

如图 13.21 所示。

形状投射返回的接触信息必然比射线投射复杂一些。我们不能简单地返回一个或多个 t 值，因为 at 值仅描述了形状沿其路径的中心点位置。它并未告诉我们它在形状的表面或内部的哪个位置与受撞击的可碰撞体发生了接触。因此，大多数形状投射 API 都会返回 at 值和实际接触点，以及其他相关信息（例如，哪个可碰撞体被撞击，接触点处的表面法线等）。

与射线投射 API 不同，形状投射系统必须始终能够报告多个接触点。这是因为即使我们只报告 t 值最早的接触点，该形状也可能接触过游戏世界中多个不同的可碰撞体，或者它可能在多个点上接触同一个非凸可碰撞体。因此，碰撞系统通常会返回一个数组

图 13.20 如果铸造形状的起始位置没有贯穿任何东西，那么该形状将沿着其线段移动非零距离，并且其接触点（如果有）将始终在其表面上。

Figure 13.19. A cast sphere that starts in penetration will be unable to move, and possibly many contact points will lie inside the cast shape in general.

In the second case, the shape can move a nonzero distance along the line segment before striking something. Presuming that it hits something, it will usually hit only a single collidable. However, it is possible for a cast shape to strike more than one collidable simultaneously if its trajectory is just right. And of course, if the impacted collidable is a non-convex poly soup, the cast shape may end up touching more than one part of the poly soup simultaneously. We can safely say that no matter what kind of convex shape is cast, it is *possible* for the cast to generate multiple contact points. The contacts will always be on the *surface* of the cast shape in this case, never inside it (because we know that the cast shape was not interpenetrating anything when it started its journey). This case is illustrated in Figure 13.20.

As with ray casts, some shape casting APIs report only the *earliest* contact experienced by the cast shape, while others allow the shape to continue along its hypothetical path, returning all the contacts it experiences on its journey. This is illustrated in Figure 13.21.

The contact information returned by a shape cast is necessarily a bit more complex than it is for a ray cast. We cannot simply return one or more t values, because a t value only describes the location of the center point of the shape along its path. It tells us nothing of where, on the surface or interior of the shape, it came into contact with the impacted collidable. As a result, most shape casting APIs return both a t value and the actual contact point, along with other relevant information (such as which collidable was struck, the surface normal at the contact point, etc.).

Unlike ray casting APIs, a shape casting system must always be capable of reporting multiple contacts. This is because even if we only report the contact with the earliest t value, the shape may have touched multiple distinct collidables in the game world, or it may be touching a single non-convex collidable at more than one point. As a result, collision systems usually return an array

Figure 13.20. If the starting location of a cast shape is not interpenetrating anything, then the shape will move a nonzero distance along its line segment, and its contacts (if any) will always be on its surface.

图 13.21。形状转换 API 可能会返回所有联系人，而不仅仅是最早的联系人。

或接触点数据结构列表，每个接触点数据结构可能看起来像这样：

```
struct ShapeCastContact
{
    F32    m_t;            // the t value for this
                           // contact

    U32    m_collidableId; // which collidable did we
                           // hit?

    Point  m_contactPoint; // location of actual
                           // contact

    Vector m_normal;       // surface normal at
                           // contact pt.

    // other information...
};
```

给定一个接触点列表，我们通常希望区分每个不同 t 值的接触点组。例如，最早的接触点实际上是由列表中所有共享最小 t 值的接触点组描述的。重要的是要意识到，碰撞系统可能会或可能不会返回按 t 排序的接触点。如果没有，手动按 t 值对结果进行排序几乎总是一个好主意。这确保了当查看列表中的第一个接触点时，它将保证位于形状路径上最早的接触点之一。

Figure 13.21. A shape casting API might return all contacts instead of only the earliest contact.

or list of contact point data structures, each of which might look something like this:

```
struct ShapeCastContact
{
    F32    m_t;            // the t value for this
                           // contact

    U32    m_collidableId; // which collidable did we
                           // hit?

    Point  m_contactPoint; // location of actual
                           // contact

    Vector m_normal;       // surface normal at
                           // contact pt.

    // other information...
};
```

Given a list of contact points, we often want to distinguish between the groups of contact points for each distinct t value. For example, the earliest contact is actually described by the group of contact points that all share the minimum t in the list. It's important to realize that collision systems may or may not return their contact points sorted by t. If it does not, it's almost always a good idea to sort the results by t manually. This ensures that if one looks at the first contact point in the list, it will be guaranteed to be among the earliest contact points along the shape's path.

成型铸件的应用

形状投射在游戏中极其有用。球形投射可用于判断虚拟摄像机是否与游戏世界中的物体发生碰撞。球形或胶囊体投射也常用于实现角色移动。例如，为了让角色在不平坦的地形上向前滑动，我们可以在角色的双脚之间投射一个球体或胶囊体，使其沿着运动方向移动。我们可以通过第二次投射来上下调整它，以确保它始终与地面保持接触。如果球体撞到非常短的垂直障碍物（例如路缘石），它会“弹起”越过路缘石。如果垂直障碍物过高（例如墙壁），投射的球体可以沿着墙壁水平滑动。投射球体的最终静止位置将成为角色在下一帧的新位置。

13.3.7.3 幻影

有时，游戏需要确定哪些可碰撞物体位于游戏世界中的某个特定区域内。例如，我们可能需要列出玩家角色周围一定半径范围内的所有敌人。为此，Havok 支持一种称为幻影 (Phantom) 的特殊可碰撞物体。

幻影的作用很像距离矢量 d 为零的形状模型。
我们随时都可以向 Phantom 请求其与世界中其他可碰撞物体的接触列表。它返回的数据格式与零距离形状模型返回的格式基本相同。

然而，与形状铸件不同，幻影在碰撞世界中是持久的。这意味着它可以充分利用碰撞引擎在检测“真实”可碰撞物体之间的碰撞时所使用的时间一致性优化。事实上，幻影与常规可碰撞物体之间的唯一区别在于，它对碰撞世界中的所有其他可碰撞物体都是“不可见的”（并且它不参与动力学模拟）。这使得它能够回答关于如果它是“真实”可碰撞物体会与哪些物体发生碰撞的假设性问题，但保证不会对碰撞世界中的其他可碰撞物体（包括其他幻影）产生任何影响。

13.3.7.4 其他类型的查询

除了强制类型转换之外，一些碰撞引擎还支持其他类型的查询。例如，Havok 支持最近点查询，该查询用于查找碰撞世界中与给定碰撞对象最近的其他碰撞对象上的点集。

13.3.8 碰撞过滤

游戏开发者经常会想要启用或禁用某些特定物体之间的碰撞。例如，大多数物体允许

Applications of Shape Casts

Shape casts are extremely useful in games. Sphere casts can be used to determine whether the virtual camera is in collision with objects in the game world. Sphere or capsule casts are also commonly used to implement character movement. For example, in order to slide the character forward on uneven terrain, we can cast a sphere or capsule that lies between the character's feet in the direction of motion. We can adjust it up or down via a second cast, to ensure that it remains in contact with the ground. If the sphere hits a very short vertical obstruction, such as a street curb, it can "pop up" over the curb. If the vertical obstruction is too tall, like a wall, the cast sphere can be slid horizontally along the wall. The final resting place of the cast sphere becomes the character's new location next frame.

13.3.7.3 Phantoms

Sometimes, games need to determine which collidable objects lie within some specific volume in the game world. For example, we might want the list of all enemies that are within a certain radius of the player character. Havok supports a special kind of collidable object known as a *phantom* for this purpose.

A phantom acts much like a shape cast whose distance vector **d** is zero. At any moment, we can ask the phantom for a list of its contacts with other collidables in the world. It returns this data in essentially the same format that would be returned by a zero-distance shape cast.

However, unlike a shape cast, a phantom is persistent in the collision world. This means that it can take full advantage of the temporal coherency optimizations used by the collision engine when detecting collisions between "real" collidables. In fact, the only difference between a phantom and a regular collidable is that it is "invisible" to all other collidables in the collision world (and it does not take part in the dynamics simulation). This allows it to answer hypothetical questions about what objects it would collide with were it a "real" collidable, but it is guaranteed not to have any effect on the other collidables—including other phantoms—in the collision world.

13.3.7.4 Other Types of Queries

Some collision engines support other kinds of queries in addition to casts. For example, Havok supports *closest point* queries, which are used to find the set of points on other collidables that are closest to a given collidable in the collision world.

13.3.8 Collision Filtering

It is quite common for game developers to want to enable or disable collisions between certain kinds of objects. For example, most objects are permitted to

物体穿过水面时——我们可能会使用浮力模拟使它们漂浮，或者它们可能直接沉入水底，但无论哪种情况，我们都不希望水面看起来像固体。大多数碰撞引擎允许根据游戏特定的标准接受或拒绝可碰撞物体之间的接触。这被称为碰撞过滤。

13.3.8.1 碰撞遮罩和图层

一种常见的过滤方法是将世界中的物体分类，然后使用查找表来确定某些类别是否允许相互碰撞。例如，在 Havok 中，可碰撞物体可以属于一个（且只能属于一个）碰撞层。Havok 中的默认碰撞过滤器由 hkpGroupFilter 类的实例表示，它为每个层维护一个 32 位掩码，每个位指示系统该特定层是否可以与其他层之一发生碰撞。

13.3.8.2 碰撞回调

另一种过滤技术是安排碰撞库在检测到碰撞时调用回调函数。该回调函数可以检查碰撞的具体细节，并根据适当的标准决定允许或拒绝碰撞。Havok 也支持这种过滤。当接触点首次添加到世界时，会调用 contactPointAdded() 回调。如果之后确定接触点有效（如果之前发现了 TOI 接触，则可能无效），则会调用 contactPointConfirmed() 回调。如果需要，应用程序可以在这些回调中拒绝接触点。

13.3.8.3 游戏特定的碰撞材质

游戏开发者经常需要对游戏世界中的可碰撞物体进行分类，一方面是为了控制它们的碰撞方式（例如使用碰撞过滤），另一方面也是为了控制其他次要效果，例如一种物体撞击另一种物体时发出的声音或产生的粒子效果。例如，我们可能需要区分木头、石头、金属、泥土、水和人体。

为了实现这一点，许多游戏都实现了碰撞形状分类机制，该机制在很多方面与渲染引擎中使用的材质系统类似。事实上，一些游戏团队使用“碰撞材质”一词来描述这种分类。其基本思想是将一组属性与每个可碰撞表面关联起来，这些属性定义了该特定表面在物理和碰撞方面的行为方式。碰撞属性可以包括声音和粒子效果，以及诸如恢复系数之类的物理属性，或者

pass through the surface of a body of water—we might employ a buoyancy simulation to make them float, or they might just sink to the bottom, but in either case we do not want the water's surface to appear solid. Most collision engines allow contacts between collidables to be accepted or rejected based on game-specific critiera. This is known as *collision filtering*.

13.3.8.1 Collision Masking and Layers

One common filtering approach is to categorize the objects in the world and then use a look-up table to determine whether certain categories are permitted to collide with one another or not. For example, in Havok, a collidable can be a member of one (and only one) collision layer. The default collision filter in Havok, represented by an instance of the class `hkpGroupFilter`, maintains a 32-bit mask for each layer, each bit of which tells the system whether or not that particular layer can collide with one of the other layers.

13.3.8.2 Collision Callbacks

Another filtering technique is to arrange for the collision library to invoke a *callback function* whenever a collision is detected. The callback can inspect the specifics of the collision and make the decision to either allow or reject the collision based on suitable criteria. Havok also supports this kind of filtering. When contact points are first added to the world, the `contactPointAdded()` callback is invoked. If the contact point is later determined to be valid (it may not be if an earlier TOI contact was found), the `contactPointConfirmed()` callback is invoked. The application may reject contact points in these callbacks if desired.

13.3.8.3 Game-Specific Collision Materials

Game developers often need to categorize the collidable objects in the game world, in part to control how they collide (as with collision filtering) and in part to control other secondary effects, such as the sound that is made or the particle effect that is generated when one type of object hits another. For example, we might want to differentiate between wood, stone, metal, mud, water and human flesh.

To accomplish this, many games implement a collision shape categorization mechanism similar in many respects to the *material system* used in the rendering engine. In fact, some game teams use the term *collision material* to describe this categorization. The basic idea is to associate with each collidable surface a set of properties that defines how that particular surface should behave from a physical and collision standpoint. Collision properties can include sound and particle effects, physical properties like coefficient of restitution or

摩擦系数、碰撞过滤信息以及游戏可能需要的任何其他信息。

对于简单的凸面图元，碰撞属性通常与整个形状相关联。对于多边形汤形状，属性可能以每个三角形为单位指定。由于后者的用法，我们通常尝试保持碰撞图元与其碰撞材质之间的绑定尽可能紧凑。一种典型的方法是通过 8 位、16 位或 32 位整数或指向材质数据的指针将碰撞图元绑定到碰撞材质。该整数索引到包含详细碰撞属性本身的数据结构全局数组。

13.4 刚体动力学

在游戏引擎中，我们特别关注物体的运动学——它们如何随时间移动。许多游戏引擎都包含一个物理系统，目的是以某种物理上逼真的方式模拟虚拟游戏世界中物体的运动。从技术上讲，游戏物理引擎通常关注一个称为动力学的特定物理领域。这是关于力如何影响物体运动的研究。直到最近，游戏物理系统几乎只关注一个称为经典刚体动力学的特定子学科。这个名称意味着在游戏的物理模拟中，做出了两个重要的简化假设：

- 经典（牛顿）力学。模拟中的物体被假定遵循牛顿运动定律。物体足够大，以至于不会产生量子效应；其速度足够低，以至于不会产生相对论效应。

- 刚体。模拟中的所有物体都是完全固体，无法变形。换句话说，它们的形状是恒定的。这个想法与碰撞检测系统的假设非常吻合。此外，刚度假设大大简化了模拟固体物体动力学所需的数学计算。

游戏物理引擎还能确保游戏世界中刚体的运动符合各种约束。最常见的约束是不可穿透——换句话说，物体之间不能相互穿透。因此，每当刚体被发现相互穿透时，物理系统都会尝试提供逼真的碰撞响应。

friction coefficients, collision filtering information and whatever other information the game might require.

For simple convex primitives, the collision properties are usually associated with the shape as a whole. For polygon soup shapes, the properties might be specified on a per-triangle basis. Because of this latter usage, we usually try to keep the binding between the collision primitive and its collision material as compact as possible. A typical approach is to bind collision primitives to collision materials via an 8-, 16- or 32-bit integer, or a pointer to the material data. This integer indexes into a global array of data structures containing the detailed collision properties themselves.

13.4 Rigid Body Dynamics

In a game engine, we are particularly concerned with the *kinematics* of objects—how they move over time. Many game engines include a *physics system* for the purposes of simulating the motion of the objects in the virtual game world in a somewhat physically realistic way. Technically speaking, game physics engines are typically concerned with a particular field of physics known as *dynamics*. This is the study of how *forces* affect the movement of objects. Until very recently, game physics systems have been focused almost exclusively on a specific subdiscipline known as *classical rigid body dynamics*. This name implies that in a game's physics simulation, two important simplifying assumptions are made:

- *Classical (Newtonian) mechanics*. The objects in the simulation are assumed to obey Newton's laws of motion. The objects are large enough that there are no quantum effects, and their speeds are low enough that there are no relativistic effects.
- *Rigid bodies*. All objects in the simulation are perfectly solid and cannot be deformed. In other words, their shape is constant. This idea meshes well with the assumptions made by the collision detection system. Furthermore, the assumption of rigidity greatly simplifies the mathematics required to simulate the dynamics of solid objects.

Game physics engines are also capable of ensuring that the motions of the rigid bodies in the game world conform to various *constraints*. The most common constraint is that of non-penetration—in other words, objects aren't allowed to pass through one another. Hence the physics system attempts to provide realistic *collision responses* whenever bodies are found to be interpen-

2 这是物理引擎和碰撞检测系统紧密互联的主要原因之一。

大多数物理系统还允许游戏开发者设置其他类型的约束，以定义物理模拟刚体之间的真实交互。这些约束可能包括铰链、棱柱关节（滑块）、球窝关节、轮子以及用于模拟昏迷或死亡角色的“布娃娃”等等。

物理系统通常共享碰撞世界数据结构，实际上，它通常将碰撞检测算法的执行作为其时间步长更新例程的一部分来驱动。动力学模拟中的刚体和碰撞引擎管理的可碰撞体之间通常存在一对一的映射。例如，在 Havok 中， hkpRigidBody 对象维护对一个且仅一个 hkpCollidable 的引用（尽管可以创建没有刚体的可碰撞体）。在 PhysX 中，这两个概念结合得更紧密一些 - NxActor 既可用作可碰撞对象，又可用作动力学模拟的刚体。这些刚体及其相应的可碰撞体通常保存在称为碰撞/物理世界（有时简称为物理世界）的单例数据结构中。

从游戏玩法的角度来看，物理引擎中的刚体通常与构成虚拟世界的逻辑对象截然不同。游戏对象的位置和方向可以由物理模拟驱动。为此，我们每帧都会查询物理引擎以获取每个刚体的变换，并以某种方式将其应用于相应游戏对象的变换。游戏对象的运动也可能由其他引擎系统（例如动画系统或角色控制系统）决定，从而驱动物理世界中刚体的位置和旋转。如第 13.3.1 节所述，单个逻辑游戏对象可能由物理世界中的一个或多个刚体表示。像岩石、武器或枪管这样的简单物体可能对应一个刚体。但一个关节角色或一台复杂的机器可能由许多相互连接的刚体部件组成。

本章的剩余部分将致力于探究游戏物理引擎的工作原理。我们将简要介绍刚体动力学模拟的基础理论。然后，我们将探讨游戏物理系统的一些最常见功能，并了解如何将物理引擎集成到游戏中。

2 或者在连续碰撞检测的情况下，碰撞响应实际上阻止了穿透的发生。

etrating.[2] This is one of the primary reasons for the tight interconnection between the physics engine and the collision detection system.

Most physics systems also allow game developers to set up other kinds of constraints in order to define realistic interactions between physically simulated rigid bodies. These may include hinges, prismatic joints (sliders), ball joints, wheels, “rag dolls” to emulate unconscious or dead characters and so on.

The physics system usually shares the collision world data structure, and in fact it usually drives the execution of the collision detection algorithm as part of its time step update routine. There is typically a one-to-one mapping between the rigid bodies in the dynamics simulation and the collidables managed by the collision engine. For example, in Havok, an `hkpRigidBody` object maintains a reference to one and only one `hkpCollidable` (although it is possible to create a collidable that has no rigid body). In PhysX, the two concepts are a bit more tightly integrated—an `NxActor` serves both as a collidable object and as a rigid body for the purposes of the dynamics simulation. These rigid bodies and their corresponding collidables are usually maintained in a singleton data structure known as the *collision/physics world*, or sometimes just the *physics world*.

The rigid bodies in the physics engine are typically distinct from the logical objects that make up the virtual world from a gameplay perspective. The positions and orientations of game objects can be driven by the physics simulation. To accomplish this, we query the physics engine every frame for the transform of each rigid body, and apply it in some way to the transform of the corresponding game object. It’s also possible for a game object’s motion, as determined by some other engine system (such as the animation system or the character control system) to drive the position and rotation of a rigid body in the physics world. As mentioned in Section 13.3.1, a single logical game object may be represented by one rigid body in the physics world, or by many. A simple object like a rock, weapon or barrel, might correspond to one rigid body. But an articulated character or a complex machine might be composed of many interconnected rigid pieces.

The remainder of this chapter will be devoted to investigating how game physics engines work. We’ll briefly introduce the theory that underlies rigid body dynamics simulations. Then we’ll investigate some of the most common features of a game physics system and have a look at how a physics engine might be integrated into a game.

[2]Or in the case of continuous collision detection, the collision response actually *prevents* the penetration from occurring.

13.4.1 一些基础

关于经典刚体动力学，已经有大量优秀的书籍、文章和幻灯片。[17] 提供了坚实的分析力学理论基础。与我们的讨论更相关的是 [39]、[13] 和 [29] 等文本，它们专门针对游戏所做的物理模拟而写。其他文本，如 [2]、[11] 和 [32]，也包含有关游戏刚体动力学的章节。Chris Hecker 为《游戏开发者杂志》撰写了一系列关于游戏物理主题的实用文章；Chris 已将这些文章以及其他各种有用的资源发布在 http://chrishecker.com/Rigid_Body_Dynamics 上。ODE 的主要作者 Russell Smith 制作了一个内容丰富的游戏动力学模拟幻灯片；可在 http://www.ode.org/slides/parc/dynamics.pdf 获取。

在本节中，我将总结大多数游戏物理引擎背后的基本理论概念。这将是一次快速的概述，因此我不得不省略一些细节。读完本章后，我强烈建议您至少阅读一些前面提到的其他资源。

13.4.1.1 单位

大多数刚体动力学模拟采用 MKS 单位制。在该单位制中，距离以米（缩写为“m”）为单位，质量以千克（缩写为“kg”）为单位，时间以秒（缩写为“s”）为单位。MKS 由此得名。

您可以根据需要配置物理系统使用其他单位，但这样做需要确保模拟中的所有内容保持一致。例如，像重力加速度 g 这样的常数，在 MKS 制中以 m/s^2 为单位，在您选择的任何单位制中都必须重新表示。大多数游戏团队为了简化操作，仍然坚持使用 MKS 制。

13.4.1.2 线性和角动力学的可分离性

不受约束的刚体是指能够沿所有三个笛卡尔轴自由平移，并且能够绕这三个轴自由旋转的刚体。我们称这样的刚体具有六个自由度 (DOF)。

不受约束的刚体的运动可以分为两个独立的部分，这也许有点令人惊讶：

- 线性动力学。这是在忽略所有旋转效应的情况下对物体运动的描述。（我们可以单独使用线性动力学来

13.4.1 Some Foundations

A great many excellent books, articles and slide presentations have been written on the topic of classical rigid body dynamics. A solid foundation in analytical mechanics theory can be obtained from [17]. Even more relevant to our discussion are texts like [39], [13] and [29], which have been written specifically about the kind of physics simulations done by games. Other texts, like [2], [11] and [32], include chapters on rigid body dynamics for games. Chris Hecker wrote a series of helpful articles on the topic of game physics for *Game Developer Magazine*; Chris has posted these and a variety of other useful resources at http://chrishecker.com/Rigid_Body_Dynamics. An informative slide presentation on dynamics simulation for games was produced by Russell Smith, the primary author of ODE; it is available at http://www.ode.org/slides/parc/dynamics.pdf.

In this section, I'll summarize the fundamental theoretical concepts that underlie the majority of game physics engines. This will be a whirlwind tour only, and by necessity I'll have to omit some details. Once you've read this chapter, I strongly encourage you to read at least a few of the additional resources cited previously.

13.4.1.1 Units

Most rigid body dynamics simulations operate in the MKS system of units. In this system, distance is measured in meters (abbreviated "m"), mass is measured in kilograms (abbreviated "kg") and time is measured in seconds (abbreviated "s"). Hence the name MKS.

You could configure your physics system to use other units if you wanted to, but if you do this, you need to make sure everything in the simulation is consistent. For example, constants like the acceleration due to gravity $\mathbf{g}$, which is measured in m/s^2 in the MKS system, would have to be re-expressed in whatever unit system you select. Most game teams just stick with MKS to keep life simple.

13.4.1.2 Separability of Linear and Angular Dynamics

An *unconstrained* rigid body is one that can translate freely along all three Cartesian axes and that can rotate freely about these three axes as well. We say that such a body has six *degrees of freedom* (DOF).

It is perhaps somewhat surprising that the motion of an unconstrained rigid body can be separated into two independent components:

- *Linear dynamics*. This is a description of the motion of the body when we ignore all rotational effects. (We can use linear dynamics alone to

描述理想化点质量的运动——即无限小且不能旋转的质量。）

- 角动力学。这是对物体旋转运动的描述。

可以想象，这种分离刚体运动线性分量和角度分量的能力在分析或模拟其行为时极其有用。这意味着我们可以计算物体的线性运动，而无需考虑旋转——就好像它是一个理想化的点质量——然后再叠加其角度运动，从而得到物体运动的完整描述。

13.4.1.3 质心

就线性动力学而言，不受约束的刚体表现为其所有质量都集中在一个点上，该点称为质心（缩写为 CM，有时也写作 COM）。质心本质上是刚体在所有可能方向上的平衡点。换句话说，刚体的质量在其质心周围均匀分布于各个方向。

对于密度均匀的物体，质心位于物体的质心。也就是说，如果我们将物体分成 N 个非常小的块，将所有这些块的位置加起来作为矢量和，然后除以块数，我们就能得到一个相当近似的质心位置。如果物体的密度不均匀，则每个小块的位置都需要根据该小块的质量进行加权，这意味着通常质心实际上是这些小块位置的加权平均值。因此，我们有

$$\mathbf{r}_{\mathrm{CM}} = \frac{\sum\limits_{\forall i} m_i \mathbf{r}_i}{\sum\limits_{\forall i} m_i} = \frac{\sum\limits_{\forall i} m_i \mathbf{r}_i}{m},$$

其中符号 m 表示物体的总质量，符号 r 表示半径矢量或位置矢量——即从世界空间原点延伸到所讨论点的矢量。（当小块的大小和质量趋近于零时，这些和在极限下变为积分。）凸体的质心始终位于凸体内部，但如果是凹体，它实际上可能位于凸体外部。（例如，字母“C”的质心在哪里？）

describe the motion of an idealized *point mass*—i.e., a mass that is infinitesimally small and cannot rotate.)

- *Angular dynamics*. This is a description of the rotational motion of the body.

As you can well imagine, this ability to separate the linear and angular components of a rigid body's motion is extremely helpful when analyzing or simulating its behavior. It means that we can calculate a body's linear motion without regard to rotation—as if it were an idealized point mass—and then layer its angular motion on top in order to arrive at a complete description of the body's motion.

13.4.1.3 Center of Mass

For the purposes of *linear* dynamics, an unconstrained rigid body acts as though all of its mass were concentrated at a single point known as the *center of mass* (abbreviated CM, or sometimes COM). The center of mass is essentially the balancing point of the body for all possible orientations. In other words, the mass of a rigid body is distributed evenly around its center of mass in all directions.

For a body with uniform density, the center of mass lies at the *centroid* of the body. That is, if we were to divide the body up into N very small pieces, add up the positions of all these pieces as a vector sum and then divide by the number of pieces, we'd end up with a pretty good approximation to the location of the center of mass. If the body's density is not uniform, the position of each little piece would need to be *weighted* by that piece's mass, meaning that in general the center of mass is really a *weighted average* of the pieces' positions. So we have

$$\mathbf{r}_{\mathrm{CM}} = \frac{\sum\limits_{\forall i} m_i \mathbf{r}_i}{\sum\limits_{\forall i} m_i} = \frac{\sum\limits_{\forall i} m_i \mathbf{r}_i}{m},$$

where the symbol m represents the total mass of the body, and the symbol $\mathbf{r}$ represents a *radius vector* or *position vector*—i.e., a vector extending from the world-space origin to the point in question. (These sums become integrals in the limit as the sizes and masses of the little pieces approach zero.)

The center of mass always lies inside a convex body, although it may actually lie outside the body if it is concave. (For example, where would the center of mass of the letter "C" lie?)

13.4.2 线性动力学

就线性动力学而言，刚体的位置可以用位置向量 rCM 来完整描述，该向量从世界空间原点延伸到刚体的质心，如图 13.22 所示。由于我们使用的是 MKS 系统，因此位置以米 (m) 为单位。在本文的剩余部分，我们将省略 CM 下标，因为我们描述的是刚体质心的运动。

图 13.22. 就线性动力学而言，刚体的位置可以通过其质心的位置完全描述。

13.4.2.1 线速度和加速度

刚体的线速度定义了刚体中心运动的速度和方向。它是一个矢量，通常以米/秒 (m/s) 为单位。速度是位置的一阶时间导数，因此我们可以写成

$$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t),$$

其中向量 r 上的点表示对时间求导。向量的微分与对每个分量分别求导相同，因此

$$v_x(t) = \frac{dr_x(t)}{dt} = \dot{r}_x(t),$$

对于 y 和 z 分量也是如此。

线性加速度是线速度对时间的一阶导数，或者说是物体中心位置对时间的二阶导数。加速度是一个矢量，通常用符号 a 表示。因此，我们可以

13.4.2 Linear Dynamics

For the purposes of linear dynamics, the position of a rigid body can be fully described by a position vector $\mathbf{r}_{CM}$ that extends from the world-space origin to the center of mass of the body, as shown in Figure 13.22. Since we're using the MKS system, position is measured in meters (m). For the remainder of this discussion, we'll drop the CM subscripts, as it is understood that we are describing the motion of the body's center of mass.

Figure 13.22. For the purposes of linear dynamics, the position of a rigid body can be fully described by the position of its center of mass.

13.4.2.1 Linear Velocity and Acceleration

The *linear velocity* of a rigid body defines the speed and direction in which the body's CM is moving. It is a vector quantity, typically measured in meters per second (m/s). Velocity is the first time derivative of position, so we can write

$$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t),$$

where the dot over the vector **r** denotes taking the derivative with respect to time. Differentiating a vector is the same as differentiating each component independently, so

$$v_x(t) = \frac{dr_x(t)}{dt} = \dot{r}_x(t),$$

and so on for the y- and z-components.

Linear acceleration is the first derivative of linear velocity with respect to time, or the second derivative of the position of a body's CM versus time. Acceleration is a vector quantity, usually denoted by the symbol **a**. So we can

写

$$\begin{aligned}\mathbf{a}(t) &= \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t)\\ &= \frac{d^2\mathbf{r}(t)}{dt^2} = \ddot{\mathbf{r}}(t).\end{aligned}$$

13.4.2.2 力和动量

力被定义为导致有质量物体加速或减速的任何因素。力在空间中既有大小也有方向，因此所有力都用矢量表示。力通常用符号 F 表示。当 N 个力施加于刚体时，它们对刚体线性运动的净效应可以通过简单地将力矢量相加来得出：

$$\mathbf{F}_{\text{net}} = \sum_{i=1}^{N} \mathbf{F}_i.$$

Newton's famous Second Law states that force is proportional to acceleration and mass:

$$\mathbf{F}(t) = m\mathbf{a}(t) = m\ddot{\mathbf{r}}(t). \tag{13.2}$$

根据牛顿定律，力的测量单位是千克米每平方秒（kg-m/s²）。这个单位也称为牛顿。

当我们将物体的线速度乘以其质量时，结果是一个称为线动量的量。通常用符号 p 表示线动量：

$$\mathbf{p}(t) = m\mathbf{v}(t).$$

当质量恒定时，公式 (13.2) 成立。但如果质量不是恒定的，例如火箭的燃料会逐渐消耗并转化为能量，公式 (13.2) 就不完全正确。正确的公式实际上如下：

$$\mathbf{F}(t) = \frac{d\mathbf{p}(t)}{dt} = \frac{d(m(t)\mathbf{v}(t))}{dt}.$$

当质量恒定且可以在导数之外取值时，这当然可以简化为更熟悉的 F = mα。线性动量对我们来说不太重要。然而，动量的概念在我们讨论角动力学时会变得重要。

write

$$\begin{aligned}\mathbf{a}(t) &= \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t)\\ &= \frac{d^2\mathbf{r}(t)}{dt^2} = \ddot{\mathbf{r}}(t).\end{aligned}$$

13.4.2.2 Force and Momentum

A *force* is defined as anything that causes an object with mass to accelerate or decelerate. A force has both a magnitude and a direction in space, so all forces are represented by vectors. A force is often denoted by the symbol **F**. When N forces are applied to a rigid body, their net effect on the body's linear motion is found by simply adding up the force vectors:

$$\mathbf{F}_{\text{net}} = \sum_{i=1}^{N} \mathbf{F}_i.$$

Newton's famous Second Law states that force is proportional to acceleration and mass:

$$\mathbf{F}(t) = m\mathbf{a}(t) = m\ddot{\mathbf{r}}(t). \tag{13.2}$$

As Newton's law implies, force is measured in units of kilogram-meters per second squared (kg-m/s^2). This unit is also called the Newton.

When we multiply a body's linear velocity by its mass, the result is a quantity known as *linear momentum*. It is customary to denote linear momentum with the symbol **p**:

$$\mathbf{p}(t) = m\mathbf{v}(t).$$

When mass is constant, Equation (13.2) holds true. But if mass is not constant, as would be the case for a rocket whose fuel is being gradually used up and converted into energy, Equation (13.2) is not exactly correct. The proper formulation is actually as follows:

$$\mathbf{F}(t) = \frac{d\mathbf{p}(t)}{dt} = \frac{d(m(t)\mathbf{v}(t))}{dt}.$$

which of course reduces to the more familiar $\mathbf{F} = m\mathbf{a}$ when the mass is constant and can be brought outside the derivative. Linear momentum is not of much concern to us. However, the concept of momentum will become relevant when we discuss angular dynamics.

13.4.3 求解运动方程

刚体动力学的核心问题是，在给定一组已知作用于物体的力的情况下，求解物体的运动。对于线性动力学，这意味着在已知净力 F net（t）以及可能的其他信息（例如先前某个时间点的位置和速度）的情况下，求解 v（t）和 r（t）。正如我们将在下文中看到的，这相当于求解一对常微分方程——一个方程给定 a（t）求解 v（t），另一个方程给定 v（t）求解 r（t）。

13.4.3.1 力作为函数

力可以是常数，也可以是时间的函数，如上所示。力也可以是物体位置、速度或任何其他量的函数。因此，一般来说，力的表达式应该写成如下形式：

$$\mathbf{F}(t, \mathbf{r}(t), \mathbf{v}(t), \ldots) = m\mathbf{a}(t). \tag{13.3}$$

这可以根据位置向量及其一阶和二阶导数重写如下：

$$\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t), \ldots) = m\ddot{\mathbf{r}}(t).$$

例如，弹簧施加的力与其被拉伸的距离成正比。在一维空间中，弹簧的静止位置为 x = 0 ，我们可以写成

$$F(t, x(t)) = -kx(t),$$

其中 k 是弹簧常数，是弹簧刚度的度量。
再举一个例子，机械粘性阻尼器（也称为阻尼器）产生的阻尼力与阻尼器活塞的速度成正比。因此，在一维空间中，我们可以写成

$$F(t, v(t)) = -bv(t),$$

其中 b 是粘性阻尼系数。

13.4.3.2 常微分方程

一般来说，常微分方程 (ODE) 是一个包含一个自变量函数及其各种导数的方程。假设自变量是时间，函数是 x(t)，那么 ODE 就是如下形式的关系

$$\frac{d^n x}{dt^n} = f\left(t, x(t), \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2}, \ldots, \frac{d^{n-1}x(t)}{dt^{n-1}}\right).$$

13.4.3 Solving the Equations of Motion

The central problem in rigid body dynamics is to solve for the motion of the body, given a set of known forces acting on it. For linear dynamics, this means finding $\mathbf{v}(t)$ and $\mathbf{r}(t)$ given knowledge of the net force $\mathbf{F}_{\text{net}}(t)$ and possibly other information, such as the position and velocity at some previous time. As we'll see below, this amounts to solving a pair of ordinary differential equations—one to find $\mathbf{v}(t)$ given $\mathbf{a}(t)$ and the other to find $\mathbf{r}(t)$ given $\mathbf{v}(t)$.

13.4.3.1 Force as a Function

A force can be constant, or it can be a function of time as shown above. A force can also be a function of the position of the body, its velocity, or any number of other quantities. So in general, the expression for force should really be written as follows:

$$\mathbf{F}(t, \mathbf{r}(t), \mathbf{v}(t), \ldots) = m\mathbf{a}(t). \tag{13.3}$$

This can be rewritten in terms of the position vector and its first and second derivatives as follows:

$$\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t), \ldots) = m\ddot{\mathbf{r}}(t).$$

For example, the force exerted by a spring is proportional to how far it has been stretched away from its natural resting position. In one dimension, with the spring's resting position at $x = 0$, we can write

$$F(t, x(t)) = -kx(t),$$

where k is the *spring constant*, a measure of the spring's stiffness.

As another example, the damping force exerted by a mechanical viscous damper (a so-called dashpot) is proportional to the velocity of the damper's piston. So in one dimension, we can write

$$F(t, v(t)) = -bv(t),$$

where b is a *viscous damping coefficient*.

13.4.3.2 Ordinary Differential Equations

In general, an *ordinary differential equation* (ODE) is an equation involving a function of one independent variable and various derivatives of that function. If our independent variable is time and our function is $x(t)$, then an ODE is a relation of the form

$$\frac{d^n x}{dt^n} = f\left(t, x(t), \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2}, \ldots, \frac{d^{n-1}x(t)}{dt^{n-1}}\right).$$

换句话说，x(t) 的 n 次导数表示为一个函数 f，其参数可以是时间 (t)、位置 (x(t)) 以及任意数量的 x(t) 导数，只要这些导数的阶数低于 n。

正如我们在公式 (13.3) 中看到的，力通常是时间、位置和速度的函数：

$$\ddot{\mathbf{r}}(t) = \frac{1}{m}\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t)).$$

这显然是一个常微分方程。我们希望求解这个常微分方程，以便找到 $\mathbf{v}(t)$ and $\mathbf{r}(t)$.

13.4.3.3 分析解决方案

在一些罕见的情况下，运动微分方程可以通过解析解得到，这意味着可以找到一个简单的闭式函数来描述物体在所有可能的时间 t 值下的位置。一个常见的例子是抛射体在重力加速度 a(t) = [0, g, 0] 的影响下垂直运动，其中 g = −9.8 m/s²。在这种情况下，运动微分方程可以归结为

$$\ddot{y}(t) = g.$$

积分一次可得

$$\dot{y}(t) = gt + v_0,$$

其中 v 0 是时刻 t = 0 时的垂直速度。再次积分可得到熟悉的解

$$y(t) = \tfrac{1}{2}gt^2 + v_0 t + y_0,$$

其中 y 0 是物体的初始垂直位置。

然而，在游戏物理学中，解析解几乎是不可能的。部分原因在于，某些微分方程的闭式解根本未知。此外，游戏是一种交互式模拟，因此我们通常无法预测游戏中的力量随时间的变化。这使得我们无法找到游戏中物体位置和速度随时间变化的简单闭式表达式。

当然，这条经验法则也有例外。例如，为了确定抛射物必须以多大速度发射才能击中预定目标，我们常常需要求解一个闭式表达式。

13.4.4 数值积分

出于上述原因，游戏物理引擎采用了一种称为数值积分的技术。利用这种技术，我们以时间步长的方式求解微分方程——使用前一个时间步长的解

Put another way, the nth derivative of $x(t)$ is expressed as a function f whose arguments can be time (t), position $(x(t))$, and any number of derivatives of $x(t)$ as long as those derivatives are of *lower order* than n.

As we saw in Equation (13.3), force is a function of time, position and velocity in general:

$$\ddot{\mathbf{r}}(t) = \frac{1}{m}\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t)).$$

This clearly qualifies as an ODE. We wish to solve this ODE in order to find $\mathbf{v}(t)$ and $\mathbf{r}(t)$.

13.4.3.3 Analytical Solutions

In some rare situations, the differential equations of motion can be solved *analytically*, meaning that a simple, closed-form function can be found that describes the body's position for *all possible values* of time t. A common example is the vertical motion of a projectile under the influence of a constant acceleration due to gravity, $\mathbf{a}(t) = [0, g, 0]$, where $g = -9.8\text{m/s}^2$. In this case, the ODE of motion boils down to

$$\ddot{y}(t) = g.$$

Integrating once yields

$$\dot{y}(t) = gt + v_0,$$

where v_0 is the vertical velocity at time $t = 0$. Integrating a second time yields the familiar solution

$$y(t) = \tfrac{1}{2}gt^2 + v_0 t + y_0,$$

where y_0 is the initial vertical position of the object.

However, analytical solutions are almost never possible in game physics. This is due in part to the fact that closed-form solutions to some differential equations are simply not known. Moreover, a game is an interactive simulation, so we usually cannot predict how the forces in a game will behave over time. This makes it impossible to find simple, closed-form expressions for the positions and velocities of the objects in the game as functions of time.

There are of course exceptions to this rule of thumb. For example, it's pretty common to solve for a closed-form expression in order to determine with what velocity a projectile must be launched in order to hit a predefined target.

13.4.4 Numerical Integration

For the reasons cited above, game physics engines turn to a technique known as *numerical integration*. With this technique, we solve our differential equations in a *time-stepped* manner—using the solution from a previous time step

得出下一个时间步长的解。时间步长的持续时间通常（大致）取为常数，用符号 Δ t 表示。假设我们知道物体在当前时间 t 1 的位置和速度，并且力是时间、位置和/或速度的函数，我们希望找到下一个时间步长 t 2 = t 1 + Δ t 的位置和速度。换句话说，给定 r(t 1)、v(t 1) 和 F(t, r, v)，问题就是找到 r(t 2) 和 v(t 2)。

13.4.4.1 显式欧拉

最简单的常微分方程数值解法之一被称为显式欧拉方法。这是游戏新手程序员经常采用的直观方法。假设我们已经知道当前速度，并希望求解以下常微分方程来找到下一帧中物体的位置：

$$\mathbf{v}(t) = \dot{\mathbf{r}}(t). \tag{13.4}$$

使用显式欧拉方法，我们只需将速度从米/秒乘以时间增量转换为米/帧，然后将“一帧”的速度加到当前位置上，即可找到下一帧的新位置。这可以得到方程 (13.4) 给出的常微分方程的近似解：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t. \tag{13.5}$$

我们可以采用类似的方法，给定作用于下一帧的净力，求出下一帧物体的速度。因此，该常微分方程的近似显式欧拉解

$$\mathbf{a}(t) = \frac{\mathbf{F}_{\text{net}}(t)}{m} = \dot{\mathbf{v}}(t)$$

如下：

$$\mathbf{v}(t_2) = \mathbf{v}(t_1) + \frac{\mathbf{F}_{\text{net}}(t)}{m}\Delta t. \tag{13.6}$$

显式欧拉的解释

在方程 (13.5) 中，我们实际上假设物体的速度在时间步长内保持不变。因此，我们可以使用当前速度来预测物体在下一帧的位置。因此，时间 t 1 和 t 2 之间的位置变化 Δ r 为 Δ r = v(t1)Δt。从图形上讲，如果我们想象一个物体位置与时间的关系图，我们就是取函数在时间 t 1 处的斜率（也就是 v(t1)），然后将其线性外推到下一个时间步 t 2 。正如我们在图 13.23 中看到的，线性外推并不一定能让我们很好地估计物体在

to arrive at the solution for the next time step. The duration of the time step is usually taken to be (roughly) constant and is denoted by the symbol Δt. Given that we know the body's position and velocity at the current time t_1 and that the force is known as a function of time, position and/or velocity, we wish to find the position and velocity at the next time step $t_2 = t_1 + \Delta t$. In other words, given $\mathbf{r}(t_1)$, $\mathbf{v}(t_1)$ and $\mathbf{F}(t, \mathbf{r}, \mathbf{v})$, the problem is to find $\mathbf{r}(t_2)$ and $\mathbf{v}(t_2)$.

13.4.4.1 Explicit Euler

One of the simplest numerical solutions to an ODE is known as the *explicit Euler method*. This is the intuitive approach often taken by new game programmers. Let's assume for the moment that we already know the current velocity and that we wish to solve the following ODE to find the body's position on the next frame:

$$\mathbf{v}(t) = \dot{\mathbf{r}}(t). \tag{13.4}$$

Using the explicit Euler method, we simply convert the velocity from meters per second into meters per frame by multiplying by the time delta, and then we add "one frame's worth" of velocity onto the current position in order to find the new position on the next frame. This yields the following approximate solution to the ODE given by Equation (13.4):

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t. \tag{13.5}$$

We can take an analogous approach to find the body's velocity next frame given the net force acting this frame. Hence, the approximate explicit Euler solution to the ODE

$$\mathbf{a}(t) = \frac{\mathbf{F}_{\text{net}}(t)}{m} = \dot{\mathbf{v}}(t)$$

is as follows:

$$\mathbf{v}(t_2) = \mathbf{v}(t_1) + \frac{\mathbf{F}_{\text{net}}(t)}{m}\Delta t. \tag{13.6}$$

Interpretations of Explicit Euler

What we're really doing in Equation (13.5) is assuming that the velocity of the body is constant during the time step. Therefore, we can use the *current* velocity to predict the body's position on the *next* frame. The change in position $\Delta\mathbf{r}$ between times t_1 and t_2 is hence $\Delta\mathbf{r} = \mathbf{v}(t_1)\Delta t$. Graphically, if we imagine a plot of the position of the body versus time, we are taking the *slope* of the function at time t_1 (which is just $\mathbf{v}(t_1)$) and extrapolating it linearly to the next time step t_2. As we can see in Figure 13.23, linear extrapolation does not necessarily provide us with a particularly good estimate of the true position at the

下一个时间步长为 r(t2)，但只要速度大致恒定，它确实可以很好地工作。

图 13.23 提出了另一种解释显式欧拉方法的方法——将其视为导数的近似值。根据定义，任何导数都是两个无穷小差（在本例中为 dr / dt ）的商。显式欧拉方法使用两个有限差的商来近似导数。换句话说，dr 变为 Δ r ，dt 变为 Δ t 。由此得出

$$\frac{d\mathbf{r}}{dt} \approx \frac{\Delta\mathbf{r}}{\Delta t};$$
$$\mathbf{v}(t_1) \approx \frac{\mathbf{r}(t_2) - \mathbf{r}(t_1)}{t_2 - t_1},$$

再次简化为公式 (13.5)。该近似值实际上仅在速度在时间步长上恒定时才有效。当 Δ t 趋于零（此时它变得恰好正确）时，它在极限情况下也成立。显然，同样的分析也可以应用于公式 (13.6)。

13.4.4.2 数值方法的性质

我们已经暗示过，显式欧拉方法并不是特别精确。让我们更具体地阐述一下这个想法。常微分方程的数值解实际上有三个重要且相互关联的性质：

- 收敛。随着时间步长Δ t趋于零，近似解是否越来越接近真实解？
- 阶数 。给定一个常微分方程解的特定数值近似，其误差有多“严重”？数值常微分方程解中的误差通常与时间步长 Δ t 的某个幂成正比，因此它们通常使用“大 O”符号表示（例如，O(Δt²)）。我们说

图 13.23 在显式欧拉方法中，使用时间 t 1 时 r(t) 的斜率，从 r(t1) 线性外推到 r(t2) 真实值的估计值。

next time step $\mathbf{r}(t_2)$, but it does work reasonably well as long as the velocity is roughly constant.

Figure 13.23 suggests another way to interpret the explicit Euler method—as an approximation of a derivative. By definition, any derivative is the quotient of two infinitesimally small differences (in our case, $d\mathbf{r}/dt$). The explicit Euler method approximates this using the quotient of two *finite differences*. In other words, $d\mathbf{r}$ becomes $\Delta\mathbf{r}$ and dt becomes Δt. This yields

$$\frac{d\mathbf{r}}{dt} \approx \frac{\Delta\mathbf{r}}{\Delta t};$$
$$\mathbf{v}(t_1) \approx \frac{\mathbf{r}(t_2) - \mathbf{r}(t_1)}{t_2 - t_1},$$

which again simplifies to Equation (13.5). This approximation is really only valid when the velocity is constant over the time step. It is also valid in the limit as Δt tends toward zero (at which point it becomes *exactly* right). Obviously, this same analysis can be applied to Equation (13.6) as well.

13.4.4.2 Properties of Numerical Methods

We've implied that the explicit Euler method is not particularly accurate. Let's pin this idea down more concretely. A numerical solution to an ordinary differential equation actually has three important and interrelated properties:

- *Convergence*. As the time step Δt tends toward zero, does the approximate solution get closer and closer to the real solution?
- *Order*. Given a particular numerical approximation to the solution of an ODE, how "bad" is the error? Errors in numerical ODE solutions are typically proportional to some power of the time step duration Δt, so they are often written using "big O" notation (e.g., $O(\Delta t^2)$). We say

Figure 13.23. In the explicit Euler method, the slope of $\mathbf{r}(t)$ at time t_1 is used to linearly extrapolate from $\mathbf{r}(t_1)$ to an estimate of the true value of $\mathbf{r}(t_2)$.

当误差项为
$O(\Delta t^{(n+1)})$.

- 稳定性。数值解是否会随着时间的推移而趋于“稳定”?
  如果数值方法向系统中添加能量，物体速度最终会“爆炸”，系统就会变得不稳定。另一方面，如果数值方法倾向于从系统中移除能量，则会对整体产生阻尼效应，系统就会变得稳定。

阶的概念需要进一步解释。我们通常通过比较数值方法的近似方程与常微分方程精确解的无限泰勒级数展开式来测量其误差。然后，我们通过减去这两个方程来消除项。剩余的泰勒项表示该方法固有的误差。例如，显式欧拉方程为

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t.$$

精确解的无限泰勒级数展开为

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots,$$

其中 r（3）表示关于时间的三阶导数。因此，误差由 v Δ t 项之后的所有项表示，该项的阶数为 O（Δ t 2）（因为该项的阶数大于其他高阶项的阶数）：

$$\begin{aligned}\mathbf{E} &= \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots \\ &= O\left(\Delta t^2\right).\end{aligned}$$

为了明确方法的误差，我们通常会在其方程式末尾用“大O”符号加上误差项。例如，显式欧拉方法的方程式最准确的写法如下：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + O\left(\Delta t^2\right).$$

我们称显式欧拉方法为“一阶”方法，因为它的精度高达（包括）包含Δ t 的一次方项的泰勒级数项。一般而言，如果一个方法的误差项为 O（Δ t（n + 1）），则称其为“n 阶”方法。

13.4.4.3 显式欧拉的替代方法

显式欧拉方法在游戏中的简单积分任务中应用广泛，在速度接近恒定时效果最佳。然而，由于其

that a particular numerical method is of “order n” when its error term is $O(\Delta t^{(n+1)})$.

- *Stability*. Does the numerical solution tend to “settle down” over time? If a numerical method adds energy into the system, object velocities will eventually “explode,” and the system will become *unstable*. On the other hand, if a numerical method tends to remove energy from the system, it will have an overall damping effect, and the system will be *stable*.

The concept of order warrants a little more explanation. We usually measure the error of a numerical method by comparing its approximate equation with the infinite Taylor series expansion of the exact solution to the ODE. We then cancel terms by subtracting the two equations. The remaining Taylor terms represent the error inherent in the method. For example, the explicit Euler equation is

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t.$$

The infinite Taylor series expansion of the exact solution is

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots,$$

where $\mathbf{r}^{(3)}$ represents the third derivative with respect to time. Therefore, the error is represented by all of the terms after the $\mathbf{v}\Delta t$ term, which is of order $O(\Delta t^2)$ (because this term dwarfs the other higher-order terms):

$$\begin{aligned}\mathbf{E} &= \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + \ldots \\ &= O\left(\Delta t^2\right).\end{aligned}$$

To make the error of a method explicit, we'll often write its equation with the error term added in “big O” notation at the end. For example, the explicit Euler method's equation is most accurately written as follows:

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + O\left(\Delta t^2\right).$$

We say that the explicit Euler method is a “first-order” method because it is accurate up to *and including* the Taylor series term involving Δt to the first power. In general, if a method's error term is $O(\Delta t^{(n+1)})$, then it is said to be an “order n” method.

13.4.4.3 Alternatives to Explicit Euler

The explicit Euler method sees quite a lot of use for simple integration tasks in games, producing the best results when the velocity is nearly constant. However, it is not used in general-purpose dynamics simulations because of its

误差大，稳定性差。求解常微分方程 (ODE) 的数值方法有很多，包括后向欧拉法（另一种一阶方法）、中点欧拉法（一种二阶方法）以及龙格-库塔法。（四阶龙格-库塔法，通常缩写为“RK4”，尤其流行。）我们不会在这里详细介绍这些方法，因为您可以在网上和文献中找到大量关于它们的信息。维基百科页面 http://en.wikipedia.org/wiki/Numerical_ordinary _differential_equations 是学习这些方法的绝佳起点。

13.4.4.4 Verlet积分

目前在交互式游戏中最常用的数值常微分方程 (ODE) 方法可能是 Verlet 方法，所以我将花点时间详细地描述一下它。该方法实际上有两种变体：常规 Verlet 和所谓的速度 Verlet。我将在这里介绍这两种方法，但我会将理论和深入解释留给大量相关的论文和网页。（首先，请查看 http://en.wikipedia.org/wiki/Verlet_integration。）常规 Verlet 方法之所以具有吸引力，是因为它能够实现高阶（低误差），计算起来相对简单且成本低，并且可以一步直接以加速度的形式得到位置解（而不是通常需要两步才能从加速度转化为速度，再从速度转化为位置）。该公式通过将两个泰勒级数展开式相加得出，一个向前沿时间方向展开，另一个向后沿时间方向展开：

$$\mathbf{r}(t_1+\Delta t) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + O(\Delta t^4);$$

$$\mathbf{r}(t_1-\Delta t) = \mathbf{r}(t_1) - \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 - \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + O(\Delta t^4).$$

将这些表达式相加，会使负项与相应的正项相消。结果给出了下一个时间步的加速度位置，以及当前和前一个时间步的两个（已知）位置。这是常规的 Verlet 方法：

$$\mathbf{r}(t_1+\Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1-\Delta t) + \mathbf{a}(t_1)\Delta t^2 + O(\Delta t^4).$$

就净力而言，Verlet 方法变为

$$\mathbf{r}(t_1+\Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1-\Delta t) + \frac{\mathbf{F}_{\text{net}}(t_1)}{m}\Delta t^2 + O(\Delta t^4).$$

这个表达式中明显没有速度。但是，可以使用以下不太准确的近似值（以及其他一些替代方法）来求出速度：

$$\mathbf{v}(t_1+\Delta t) = \frac{\mathbf{r}(t_1+\Delta t) - \mathbf{r}(t_1)}{\Delta t} + O(\Delta t).$$

high error and poor stability. There are all sorts of other numerical methods for solving ODEs, including backward Euler (another first-order method), midpoint Euler (a second-order method) and the family of Runge-Kutta methods. (The fourth-order Runge-Kutta, often abbreviated “RK4,” is particularly popular.) We won’t describe these in any detail here, as you can find voluminous amounts of information about them online and in the literature. The Wikipedia page http://en.wikipedia.org/wiki/Numerical_ordinary _differential_equations serves as an excellent jumping-off point for learning these methods.

13.4.4.4 Verlet Integration

The numerical ODE method most often used in interactive games these days is probably the Verlet method, so I’ll take a moment to describe it in some detail. There are actually two variants of this method: regular Verlet and the so-called *velocity Verlet*. I’ll present both methods here, but I’ll leave the theory and deep explanations to the myriad papers and Web pages available on the topic. (For a start, check out http://en.wikipedia.org/wiki/Verlet_integration.)

The regular Verlet method is attractive because it achieves a high order (low error), is relatively simple and inexpensive to evaluate, and produces a solution for position directly in terms of acceleration in one step (as opposed to the two steps normally required to go from acceleration to velocity and then from velocity to position). The formula is derived by adding two Taylor series expansions, one going forward in time and one going backward in time:

$$\mathbf{r}(t_1+\Delta t) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + O(\Delta t^4);$$

$$\mathbf{r}(t_1-\Delta t) = \mathbf{r}(t_1) - \dot{\mathbf{r}}(t_1)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 - \tfrac{1}{6}\mathbf{r}^{(3)}(t_1)\Delta t^3 + O(\Delta t^4).$$

Adding these expressions causes the negative terms to cancel with the corresponding positive ones. The result gives us the position at the next time step in terms of the acceleration and the two (known) positions at the current and previous time steps. This is the regular Verlet method:

$$\mathbf{r}(t_1+\Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1-\Delta t) + \mathbf{a}(t_1)\Delta t^2 + O(\Delta t^4).$$

In terms of net force, the Verlet method becomes

$$\mathbf{r}(t_1+\Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1-\Delta t) + \frac{\mathbf{F}_{\text{net}}(t_1)}{m}\Delta t^2 + O(\Delta t^4).$$

The velocity is conspicuously absent from this expression. However, it can be found using the following somewhat inaccurate approximation (among other alternatives):

$$\mathbf{v}(t_1+\Delta t) = \frac{\mathbf{r}(t_1+\Delta t) - \mathbf{r}(t_1)}{\Delta t} + O(\Delta t).$$

13.4.4.5 速度 Verlet

更常用的速度Verlet方法是一个四步过程，其中时间步长分为两部分以方便求解。已知a(t1)=1mF(t1,r(t1),v(t1))，我们执行以下操作：

1. 计算 r(t1+Δt)=r(t1)+v(t1)Δt+12a(t1)Δt2。
2. 计算 v(t1+12Δt)=v(t1)+12a(t1)Δt。
3. 确定 a(t1+Δt)=a(t2)=1mF(t2,r(t2),v(t2))。
4. 计算 v(t1+Δt)=v(t1+12Δt)+12a(t1+Δt)Δt。

注意，在第三步中，力函数取决于下一时间步的位置和速度，即 r(t2) 和 v(t2)。我们已经在步骤 1 中计算了 r(t2)，因此只要力不依赖于速度，我们就掌握了所需的所有信息。如果它依赖于速度，那么我们必须近似计算下一帧的速度，或许可以使用显式欧拉方法。

13.4.5 二维角动力学

到目前为止，我们一直专注于分析物体质心（其作用类似于点质量）的线性运动。正如我之前所说，不受约束的刚体会绕其质心旋转。这意味着我们可以将物体的角运动叠加在其质心的线性运动之上，从而完整地描述物体的整体运动。研究物体在施加力作用下做出的旋转运动的学科称为角动力学。

在二维空间中，角动力学的工作原理几乎与线性动力学相同。每个线性量都有一个对应的角动力学模型，其数学计算过程相当简洁。因此，我们先来研究一下二维角动力学。正如我们将看到的，当我们将讨论扩展到三维空间时，事情会变得有些复杂，但到时候我们再来探讨这个问题！

13.4.5.1 方向和角速度

在二维空间中，每个刚体都可以看作是一层薄片。（一些物理教材将这种物体称为平面薄片。）所有线性运动都发生在 xy 平面上，所有旋转都围绕 z 轴进行。（想象一下木制拼图在桌上冰球桌上滑动的场景。）二维空间中刚体的方向可以用角度 θ 来完整描述，该角度以弧度为单位，相对于某个约定的旋转零点。例如，我们

13.4.4.5 Velocity Verlet

The more commonly used *velocity Verlet* method is a four-step process in which the time step is divided into two parts to facilitate the solution. Given that $\mathbf{a}(t_1) = \frac{1}{m}\mathbf{F}\big(t_1, \mathbf{r}(t_1), \mathbf{v}(t_1)\big)$ is known, we do the following:

1. Calculate $\mathbf{r}(t_1 + \Delta t) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t + \frac{1}{2}\mathbf{a}(t_1)\Delta t^2$.
2. Calculate $\mathbf{v}(t_1 + \frac{1}{2}\Delta t) = \mathbf{v}(t_1) + \frac{1}{2}\mathbf{a}(t_1)\Delta t$.
3. Determine $\mathbf{a}(t_1 + \Delta t) = \mathbf{a}(t_2) = \frac{1}{m}\mathbf{F}\big(t_2, \mathbf{r}(t_2), \mathbf{v}(t_2)\big)$.
4. Calculate $\mathbf{v}(t_1 + \Delta t) = \mathbf{v}(t_1 + \frac{1}{2}\Delta t) + \frac{1}{2}\mathbf{a}(t_1 + \Delta t)\Delta t$.

Notice in the third step that the force function depends on the position and velocity on the *next* time step, $\mathbf{r}(t_2)$ and $\mathbf{v}(t_2)$. We already calculated $\mathbf{r}(t_2)$ in step 1, so we have all the information we need as long as the force is not velocity-dependent. If it is velocity-dependent, then we must approximate the next frame's velocity, perhaps using the explicit Euler method.

13.4.5 Angular Dynamics in Two Dimensions

Up until now, we've focused on analyzing the linear motion of a body's center of mass (which acts as if it were a point mass). As I said earlier, an unconstrained rigid body will rotate about its center of mass. This means that we can layer the angular motion of a body on top of the linear motion of its center of mass in order to arrive at a complete description of the body's overall motion. The study of a body's rotational motion in response to applied forces is called *angular dynamics*.

In two dimensions, angular dynamics works almost identically to linear dynamics. For each linear quantity, there's an angular analog, and the mathematics works out quite neatly. So let's investigate two-dimensional angular dynamics first. As we'll see, when we extend the discussion into three dimensions, things get a bit messier, but we'll burn that bridge when we get to it!

13.4.5.1 Orientation and Angular Speed

In two dimensions, every rigid body can be treated as a thin sheet of material. (Some physics texts refer to such a body as a *plane lamina*.) All linear motion occurs in the xy-plane, and all rotations occur about the z-axis. (Visualize wooden puzzle pieces sliding about on an air hockey table.)

The orientation of a rigid body in 2D is fully described by an angle θ, measured in radians relative to some agreed-upon zero rotation. For example, we

可以指定当赛车在世界空间中正对 x 轴时，θ = 0。这个角度当然是一个时变函数，所以我们将其表示为 θ（t）。

13.4.5.2 角速度和加速度

角速度测量的是物体旋转角度随时间变化的速率。在二维空间中，角速度是一个标量，更准确地说是角速度，因为"速度"一词实际上仅适用于矢量。它用标量函数ω(t)表示，单位为弧度/秒(rad/s)。角速度是方向角θ(t)对时间的导数：

Angular:	Linear:
$\omega(t) = \frac{d\theta(t)}{dt} = \dot{\theta}(t)$	$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t).$

正如我们所料，角加速度，表示为 α(t)，以弧度每平方秒 (rad/s2) 为单位，是角速度的变化率：

Angular:	Linear:
$\alpha(t) = \frac{d\omega(t)}{dt} = \dot{\omega}(t) = \ddot{\theta}(t)$	$\mathbf{a}(t) = \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t) = \ddot{\mathbf{r}}(t).$

13.4.5.3 惯性矩

质量的转动当量称为转动惯量。正如质量描述的是改变质点线速度的难易程度，转动惯量衡量的是改变刚体绕特定轴旋转的角速度的难易程度。如果物体的质量集中在旋转轴附近，它绕该轴旋转会相对容易，因此它的转动惯量会比质量分散在该轴附近的物体小。

由于我们现在关注的是二维角动力学，旋转轴始终为 z ，物体的转动惯量是一个简单的标量。转动惯量通常用符号 I 表示。我们在此不讨论转动惯量的具体计算方法。完整的推导过程请参见 [17]。

13.4.5.4 扭矩

到目前为止，我们假设所有力都作用于刚体的质心。然而，一般来说，力可以作用于物体上的任意点。

might specify that $\theta = 0$ when a race car is facing directly down the positive x-axis in world space. This angle is of course a time-varying function, so we denote it $\theta(t)$.

13.4.5.2 Angular Speed and Acceleration

Angular velocity measures the rate at which a body's rotation angle changes over time. In two dimensions, angular velocity is a scalar, more correctly called angular *speed*, since the term "velocity" really only applies to vectors. It is denoted by the scalar function $\omega(t)$ and measured in radians per second (rad/s). Angular speed is the derivative of the orientation angle $\theta(t)$ with respect to time:

Angular:	Linear:
$\omega(t) = \frac{d\theta(t)}{dt} = \dot{\theta}(t)$	$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt} = \dot{\mathbf{r}}(t).$

And as we'd expect, angular acceleration, denoted $\alpha(t)$ and measured in radians per second squared (rad/s^2), is the rate of change of angular speed:

Angular:	Linear:
$\alpha(t) = \frac{d\omega(t)}{dt} = \dot{\omega}(t) = \ddot{\theta}(t)$	$\mathbf{a}(t) = \frac{d\mathbf{v}(t)}{dt} = \dot{\mathbf{v}}(t) = \ddot{\mathbf{r}}(t).$

13.4.5.3 Moment of Inertia

The rotational equivalent of mass is a quantity known as the *moment of inertia*. Just as mass describes how easy or difficult it is to change the linear velocity of a point mass, the moment of inertia measures how easy or difficult it is to change the angular speed of a rigid body about a particular axis. If a body's mass is concentrated near an axis of rotation, it will be relatively easier to rotate about that axis, and it will hence have a smaller moment of inertia than a body whose mass is spread out away from that axis.

Since we're focusing on two-dimensional angular dynamics right now, the axis of rotation is always z, and a body's moment of inertia is a simple scalar value. Moment of inertia is usually denoted by the symbol I. We won't get into the details of how to calculate the moment of inertia here. For a full derivation, see [17].

13.4.5.4 Torque

Until now, we've assumed that all forces are applied to the center of mass of a rigid body. However, in general, forces can be applied at arbitrary points on a

物体。如果力的作用线穿过物体的质心，那么正如我们已经看到的，该力只会产生线性运动。否则，除了通常引起的线性运动外，该力还会引入一个称为扭矩的旋转力。如图 13.24 所示。

我们可以使用向量积来计算扭矩。首先，我们将力的作用点表示为一个矢量 r ，该矢量从物体的质心延伸到力的作用点。（换句话说，矢量 r 位于物体空间中，其中物体空间的原点定义为质心。）如图 13.25 所示。在位置 r 处施加力 F 所引起的扭矩 N 为

$$\mathbf{N} = \mathbf{r} \times \mathbf{F}. \tag{13.7}$$

公式 (13.7) 表明，力矩随着作用于质心的距离增大而增大。这解释了为什么杠杆可以帮助我们移动重物。它也解释了为什么直接作用于质心的力不会产生扭矩和旋转——在这种情况下，矢量 r 的大小为零。

当两个或多个力作用于刚体时，每个力产生的扭矩矢量可以相加，就像我们可以对力求和一样。因此，通常我们感兴趣的是净扭矩 N net 。

在二维空间中，向量 r 和 F 必须都位于 xy 平面，因此 N 的方向始终是正或负 z 轴。因此，我们将二维扭矩表示为标量 N z ，它只是向量 N 的 z 分量。

扭矩与角加速度和转动惯量有很大关系

图 13.24。左侧，施加于物体中心轴（CM）的力会产生纯线性运动。右侧，施加于偏离中心的力会产生扭矩，从而产生旋转运动和线性运动。

body. If the line of action of a force passes through the body's center of mass, then the force will produce linear motion only, as we've already seen. Otherwise, the force will introduce a rotational force known as a *torque* in addition to the linear motion it normally causes. This is illustrated in Figure 13.24.

We can calculate torque using a cross product. First, we express the location at which the force is applied as a vector **r** extending from the body's center of mass to the point of application of the force. (In other words, the vector **r** is in *body space*, where the origin of body space is defined to be the center of mass.) This is illustrated in Figure 13.25. The torque **N** caused by a force **F** applied at a location **r** is

$$\mathbf{N} = \mathbf{r} \times \mathbf{F}. \tag{13.7}$$

Equation (13.7) implies that torque increases as the force is applied farther from the center of mass. This explains why a lever can help us to move a heavy object. It also explains why a force applied directly through the center of mass produces no torque and no rotation—the magnitude of the vector **r** is zero in this case.

When two or more forces are applied to a rigid body, the torque vectors produced by each one can be summed, just as we can sum forces. So in general we are interested in the net torque, $\mathbf{N}_{\text{net}}$.

In two dimensions, the vectors **r** and **F** must both lie in the xy-plane, so **N** will always be directed along the positive or negative z-axis. As such, we'll denote a two-dimensional torque via the scalar N_z, which is just the z-component of the vector **N**.

Torque is related to angular acceleration and moment of inertia in much

Figure 13.24. On the left, a force applied to a body's CM produces purely linear motion. On the right, a force applied off-center will give rise to a torque, producing rotational motion as well as linear motion.

图 13.25。扭矩是通过计算力在物体空间中的作用点（即相对于质心）与力矢量的叉积来计算的。为了便于说明，这里以二维方式显示矢量；如果可以画出来，扭矩矢量应该指向纸面。

与力与线性加速度和质量的关系相同：

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ N_z(t) = I\alpha(t) = I\dot{\omega}(t) = I\ddot{\theta}(t) & \mathbf{F}(t) = m\mathbf{a}(t) = m\dot{\mathbf{v}}(t) = m\ddot{\mathbf{r}}(t). \end{array} \tag{13.8}$$

13.4.5.5 求解二维角运动方程

对于二维情况，我们可以使用与线性动力学问题完全相同的数值积分技术来求解角运动方程。我们要求解的一对常微分方程如下：

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ N_{\text{net}}(t) = I\dot{\omega}(t) & \mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t) \\ \omega(t) = \dot{\theta}(t) & \mathbf{v}(t) = \dot{\mathbf{r}}(t), \end{array}$$

其近似显式欧拉解为

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ \omega(t_2) = \omega(t_1) + I^{-1}N_{\text{net}}(t_1)\Delta t & \mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t_1)\Delta t \\ \theta(t_2) = \theta(t_1) + \omega(t_1)\Delta t & \mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t. \end{array}$$

当然，我们也可以应用任何其他更精确的数值方法，例如速度 Verlet 方法（为了简洁起见，我在这里省略了线性情况，但请将其与第 13.4.4.5 节中给出的步骤进行比较）：

1. 计算 $\theta(t_1 + \Delta t) = \theta(t_1) + \omega(t_1)\Delta t + \frac{1}{2}\alpha(t_1)\Delta t^2$。
2. 计算 $\omega(t_1 + \frac{1}{2}\Delta t) = \omega(t_1) + \frac{1}{2}\alpha(t_1)\Delta t$。

Figure 13.25. Torque is calculated by taking the cross product between a force's point of application in body space (i.e., relative to the center of mass) and the force vector. The vectors are shown here in two dimensions for ease of illustration; if it could be drawn, the torque vector would be directed into the page.

the same way that force is related to linear acceleration and mass:

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ N_z(t) = I\alpha(t) = I\dot{\omega}(t) = I\ddot{\theta}(t) & \mathbf{F}(t) = m\mathbf{a}(t) = m\dot{\mathbf{v}}(t) = m\ddot{\mathbf{r}}(t). \end{array} \tag{13.8}$$

13.4.5.5 Solving the Angular Equations of Motion in Two Dimensions

For the two-dimensional case, we can solve the angular equations of motion using exactly the same numerical integration techniques we applied to the linear dynamics problem. The pair of ODEs that we wish to solve is as follows:

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ N_{\text{net}}(t) = I\dot{\omega}(t) & \mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t) \\ \omega(t) = \dot{\theta}(t) & \mathbf{v}(t) = \dot{\mathbf{r}}(t), \end{array}$$

and their approximate explicit Euler solutions are

$$\begin{array}{c|c} \text{Angular:} & \text{Linear:} \\ \omega(t_2) = \omega(t_1) + I^{-1}N_{\text{net}}(t_1)\Delta t & \mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t_1)\Delta t \\ \theta(t_2) = \theta(t_1) + \omega(t_1)\Delta t & \mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t. \end{array}$$

Of course, we could apply any of the other more-accurate numerical methods as well, such as the velocity Verlet method (I've omitted the linear case here for compactness, but compare this to the steps given in Section 13.4.4.5):

1. Calculate $\theta(t_1 + \Delta t) = \theta(t_1) + \omega(t_1)\Delta t + \frac{1}{2}\alpha(t_1)\Delta t^2$.
2. Calculate $\omega(t_1 + \frac{1}{2}\Delta t) = \omega(t_1) + \frac{1}{2}\alpha(t_1)\Delta t$.

3. 计算 α (t 1 + Δ t) = α (t 2) = I − 1 N net $\quad (t_2, \theta(t_2), \omega(t_2))$.

4. 计算 ω (t 1 + Δ t) = ω (t 1 + 1 $\quad \Delta t) + \frac{1}{2}\alpha(t_1 + \Delta t)\Delta t$.

13.4.6 三维角动力学

三维角动力学比二维角动力学稍微复杂一些，尽管基本概念非常相似。在下一节中，我将简要概述三维角动力学的工作原理，主要关注那些新手容易混淆的地方。更多信息，请参阅 Glenn Fiedler 关于该主题的系列文章，网址为 http://gafferongames.com/game-physics/physics-in-3d/。另一个有用的资源是卡内基梅隆大学机器人研究所的 David Baraff 撰写的论文《基于物理的建模简介》，网址为 http://www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf。

13.4.6.1 惯性张量

刚体在三个坐标轴上的质量分布可能非常不同。因此，我们应该预期物体在不同轴上具有不同的惯性矩。例如，一根细长的杆应该相对容易绕其长轴旋转，因为所有质量都集中在非常靠近旋转轴的位置。同样，一根杆应该相对难以绕其短轴旋转，因为它的质量分散在离旋转轴较远的地方。事实确实如此，这也是为什么花样滑冰运动员在将四肢紧贴身体时旋转速度更快的原因。

在三维空间中，刚体的转动质量用一个 3 × 3 的矩阵表示，称为惯性张量 。它通常用符号 I 表示（与前面一样，我们在这里不再描述如何计算惯性张量；详情参见 [17]）：

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}.$$

该矩阵对角线上的元素是物体绕其三个主轴 Ixx、Iyy 和 Izz 的惯性矩。非对角线上的元素称为惯性积。当物体绕三个主轴对称时（例如矩形框的情况），惯性积为零。当惯性积非零时，它们往往会产生物理上逼真但又略微不符合直觉的运动，而普通游戏玩家可能认为这些运动本身就是“错误的”。因此，惯性张量通常为

3. Calculate $\alpha(t_1 + \Delta t) = \alpha(t_2) = I^{-1} N_{\text{net}}(t_2, \theta(t_2), \omega(t_2))$.

4. Calculate $\omega(t_1 + \Delta t) = \omega(t_1 + \frac{1}{2}\Delta t) + \frac{1}{2}\alpha(t_1 + \Delta t)\Delta t$.

13.4.6 Angular Dynamics in Three Dimensions

Angular dynamics in three dimensions is a somewhat more complex topic than its two-dimensional counterpart, although the basic concepts are of course very similar. In the following section, I'll give a very brief overview of how angular dynamics works in 3D, focusing primarily on the things that are typically confusing to someone who is new to the topic. For further information, check out Glenn Fiedler's series of articles on the topic, available at http://gafferongames.com/game-physics/physics-in-3d/. Another helpful resource is the paper entitled "An Introduction to Physically Based Modeling" by David Baraff of the Robotics Institute at Carnegie Mellon University, available at http://www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf.

13.4.6.1 The Inertia Tensor

A rigid body may have a very different distribution of mass about the three coordinate axes. As such, we should expect a body to have different moments of inertia about different axes. For example, a long thin rod should be relatively easy to make rotate about its long axis because all the mass is concentrated very close to the axis of rotation. Likewise, the rod should be relatively more difficult to make rotate about its short axis because its mass is spread out farther from the axis. This is indeed the case, and it is why a figure skater spins faster when she tucks her limbs in close to her body.

In three dimensions, the rotational mass of a rigid body is represented by a 3×3 matrix known as its *inertia tensor*. It is usually represented by the symbol **I** (as before, we won't describe how to calculate the inertia tensor here; see [17] for details):

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}.$$

The elements lying along the diagonal of this matrix are the moments of inertia of the body about its three principal axes, I_{xx}, I_{yy} and I_{zz}. The off-diagonal elements are called *products of inertia*. They are zero when the body is symmetrical about all three principal axes (as would be the case for a rectangular box). When they are nonzero, they tend to produce physically realistic yet somewhat unintuitive motions that the average game player would probably think were "wrong" anyway. Therefore, the inertia tensor is often

简化为三元素矢量引擎。 $\begin{bmatrix} I_{xx} \end{bmatrix}$ 我 yy 我 zz 在游戏物理中

13.4.6.2 三维方向

在二维空间中，我们知道刚体的方位可以用一个角度 θ 来描述，该角度测量刚体绕 z 轴的旋转（假设运动发生在 xy 平面上）。在三维空间中，刚体方位可以用三个欧拉角 [θ x θ y θ z] 来表示，每个欧拉角代表刚体绕三个笛卡尔轴之一的旋转。然而，正如我们在第 5 章中看到的，欧拉角容易出现万向节锁问题，并且在数学上难以处理。因此，刚体方位通常用 3 × 3 矩阵 R 或单位四元数 q 来表示。本章我们将只使用四元数形式。

回想一下，四元数是一个四元素向量，其 x、y 和 z 分量可以解释为沿旋转轴的单位向量 u，按半角的正弦缩放，其 w 分量是半角的余弦：

$$\begin{aligned} \mathsf{q} &= \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{q} & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{u} \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}. \end{aligned}$$

物体的方向当然是时间的函数，因此我们应该将其写为 q (t)。
同样，我们需要选择一个任意方向作为零旋转。例如，我们可以默认每个物体的正面位于世界空间的正 z 轴方向，y 轴朝上，x 轴朝左。任何非恒等四元数都会使物体旋转偏离这个规范的世界空间方向。规范方向的选择是任意的，但当然，在游戏中的所有资源中保持一致非常重要。

13.4.6.3 三维空间中的角速度和动量

在三维空间中，角速度是一个矢量，用ω(t)表示。角速度矢量可以表示为一个单位长度的矢量u，它定义了旋转轴，并由物体绕u轴的二维角速度ωu=˙ θu缩放。因此，

$$\boldsymbol{\omega}(t) = \omega_u(t)\mathbf{u} = \dot{\theta}_u(t)\mathbf{u}.$$

在线性动力学中，我们看到，如果没有力作用于物体，则线加速度为零，线速度为常数。在二维角动力学中，这一点同样成立：如果没有扭矩

simplified down to the three-element vector $\begin{bmatrix} I_{xx} & I_{yy} & I_{zz} \end{bmatrix}$ in game physics engines.

13.4.6.2 Orientation in Three Dimensions

In two dimensions, we know that the orientation of a rigid body can be described by a single angle θ, which measures rotation about the z-axis (assuming the motion is taking place in the xy-plane). In three dimensions, a body's orientation could be represented using three Euler angles $\begin{bmatrix} \theta_x & \theta_y & \theta_z \end{bmatrix}$, each representing the body's rotation about one of the three Cartesian axes. However, as we saw in Chapter 5, Euler angles suffer from gimbal lock problems and can be difficult to work with mathematically. Therefore, the orientation of a body is more often represented using either a 3×3 matrix $\mathbf{R}$ or a unit quaternion q. We'll use the quaternion form exclusively in this chapter.

Recall that a quaternion is a four-element vector whose x-, y- and z-components can be interpreted as a unit vector $\mathbf{u}$ lying along the axis of rotation, scaled by the sine of the half-angle and whose w component is the cosine of the half-angle:

$$\begin{aligned} \mathsf{q} &= \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{q} & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{u} \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}. \end{aligned}$$

A body's orientation is of course a function of time, so we should write it $\mathsf{q}(t)$.

Again, we need to select an arbitrary direction to be our zero rotation. For example, we might say that by default, the front of every object will lie along the positive z-axis in world space, with y up and x to the left. Any non-identity quaternion will serve to rotate the object away from this canonical world-space orientation. The choice of the canonical orientation is arbitrary, but of course it's important to be consistent across all assets in the game.

13.4.6.3 Angular Velocity and Momentum in Three Dimensions

In three dimensions, angular velocity is a vector quantity, denoted by $\boldsymbol{\omega}(t)$. The angular velocity vector can be visualized as a unit-length vector $\mathbf{u}$ that defines the axis of rotation, scaled by the two-dimensional angular velocity $\omega_u = \dot{\theta}_u$ of the body about the u-axis. Hence,

$$\boldsymbol{\omega}(t) = \omega_u(t)\mathbf{u} = \dot{\theta}_u(t)\mathbf{u}.$$

In linear dynamics, we saw that if there are no forces acting on a body, then the linear acceleration is zero, and linear velocity is constant. In two-dimensional angular dynamics, this again holds true: If there are no torques

图 13.26 绕其最短轴或最长轴旋转的矩形物体具有恒定的角速度矢量。
然而，当绕其中等尺寸的轴旋转时，角速度矢量的方向会发生剧烈变化。

作用于二维物体上，则角加速度α为零，绕z轴的角速度ω为常数。

不幸的是，在三维空间中情况并非如此。事实证明，即使刚体在没有任何力的情况下旋转，其角速度矢量 ω(t) 也可能不是恒定的，因为旋转轴的方向会不断改变。当你尝试在你面前的半空中旋转一个矩形物体（比如一块木块）时，你可以看到这种效果。如果你抛出木块，让它绕其最短的轴旋转，它就会以稳定的方式旋转。轴的方向大致保持不变。如果你尝试绕其最长的轴旋转木块，也会发生同样的情况。但是，如果你尝试绕剩余的轴（既不是最短的轴也不是最长的轴）旋转木块，旋转将非常不稳定。（试试看！去从婴儿那里偷一块木块，用各种方式旋转它。再想想，完成后一定要把它还回去。）随着物体的旋转，旋转轴本身的方向也会剧烈变化。如图 13.26 所示。

角速度矢量在没有扭矩的情况下会发生变化，这换句话说就是角速度不守恒。然而，一个称为角动量的相关量在没有力的情况下保持不变，因此是守恒的。角动量是线性动量的旋转等价物：

角度：	线性：
$\mathbf{L}(t) = \mathbf{I}\boldsymbol{\omega}(t)$	$\mathbf{p}(t) = m\mathbf{v}(t).$

与线性情况类似，角动量 L（t）是一个三元素矢量。
然而，与线性情况不同，转动质量（惯性张量）不是标量，而是一个 3 × 3 矩阵。因此，表达式 I ω 可以通过以下公式计算：

Figure 13.26. A rectangular object that is spun about its shortest or longest axis has a constant angular velocity vector. However, when spun about its medium-sized axis, the direction of the angular velocity vector changes wildly.

acting on a body in two dimensions, then the angular acceleration α is zero, and the angular speed ω about the z-axis is constant.

Unfortunately, this is *not* the case in three dimensions. It turns out that even when a rigid body is rotating in the absence of all forces, its angular velocity vector $\boldsymbol{\omega}(t)$ may not be constant because the axis of rotation can continually change direction. You can see this effect in action when you try to spin a rectangular object, like a block of wood, in mid-air in front of you. If you throw the block so that it is rotating about its shortest axis, it will spin in a stable way. The orientation of the axis stays roughly constant. The same thing happens if you try to spin the block about its longest axis. But if you try to spin the block around the remaining axis (the one that's neither the shortest nor the longest), the rotation will be utterly unstable. (Try it! Go steal a wooden block from a baby and spin it in various ways. On second thought, make sure to give it back when you're done.) The axis of rotation itself changes direction wildly as the object spins. This is illustrated in Figure 13.26.

The fact that the angular velocity vector can change in the absence of torques is another way of saying that angular velocity is not *conserved*. However, a related quantity called the *angular momentum* does remain constant in the absence of forces and hence *is* conserved. Angular momentum is the rotational equivalent of linear momentum:

Angular:	Linear:
$\mathbf{L}(t) = \mathbf{I}\boldsymbol{\omega}(t)$	$\mathbf{p}(t) = m\mathbf{v}(t).$

Like the linear case, angular momentum $\mathbf{L}(t)$ is a three-element vector. However, unlike the linear case, rotational mass (the inertia tensor) is not a scalar but rather a 3×3 matrix. As such, the expression $\mathbf{I}\boldsymbol{\omega}$ is computed via a

矩阵乘法：

$$\begin{bmatrix} L_x(t) \\ L_y(t) \\ L_z(t) \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x(t) \\ \omega_y(t) \\ \omega_z(t) \end{bmatrix}.$$

由于角速度 ω 不守恒，我们在动力学模拟中不像处理线速度 v 那样将其视为主要量。相反，我们将角动量 L 视为主要量。角速度是次要量，只有在我们确定了模拟每个时间步的 L 值之后才能确定。

13.4.6.4 三维扭矩

在三维空间中，我们仍然将扭矩计算为力作用点的径向位置矢量与力矢量本身的叉积（N = r × F）。公式 (13.8) 仍然成立，但我们始终将其写成角动量的形式，因为角速度不是守恒量：

$$\begin{aligned} \mathbf{N} &= \mathbf{I}\boldsymbol{\alpha}(t) \\ &= \mathbf{I}\frac{d\boldsymbol{\omega}(t)}{dt} \\ &= \frac{d}{dt}\left(\mathbf{I}\boldsymbol{\omega}(t)\right) \\ &= \frac{d\mathbf{L}(t)}{dt}. \end{aligned}$$

13.4.6.5 解三维角运动方程

在求解三维角运动方程时，我们可能会倾向于采用与求解线性运动和二维角运动完全相同的方法。我们可能会猜测运动微分方程应该写成

Angular 3D?	Linear:
$\mathbf{N}_{\text{net}}(t) = \mathbf{I}\dot{\boldsymbol{\omega}}(t)$	$\mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t)$
$\boldsymbol{\omega}(t) = \dot{\boldsymbol{\theta}}(t)$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t),$

使用显式欧拉方法，我们可能会猜测这些 ODE 的近似解看起来像这样：

Angular 3D?	Linear:
$\boldsymbol{\omega}(t_2) = \boldsymbol{\omega}(t_1) + \mathbf{I}^{-1}\mathbf{N}_{\text{net}}(t_1)\Delta t$	$\mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t)\Delta t$
$\boldsymbol{\theta}(t_2) = \boldsymbol{\theta}(t_1) + \boldsymbol{\omega}(t_1)\Delta t$	$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t.$

matrix multiplication:

$$\begin{bmatrix} L_x(t) \\ L_y(t) \\ L_z(t) \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x(t) \\ \omega_y(t) \\ \omega_z(t) \end{bmatrix}.$$

Because the angular velocity $\boldsymbol{\omega}$ is not conserved, we do not treat it as a primary quantity in our dynamics simulations the way we do the linear velocity $\mathbf{v}$. Instead, we treat angular momentum $\mathbf{L}$ as the primary quantity. The angular velocity is a secondary quantity, determined only after we have determined the value of $\mathbf{L}$ at each time step of the simulation.

13.4.6.4 Torque in Three Dimensions

In three dimensions, we still calculate torque as the cross product between the radial position vector of the point of force application and the force vector itself ($\mathbf{N} = \mathbf{r} \times \mathbf{F}$). Equation (13.8) still holds, but we always write it in terms of the angular momentum because angular velocity is not a conserved quantity:

$$\begin{aligned} \mathbf{N} &= \mathbf{I}\boldsymbol{\alpha}(t) \\ &= \mathbf{I}\frac{d\boldsymbol{\omega}(t)}{dt} \\ &= \frac{d}{dt}\left(\mathbf{I}\boldsymbol{\omega}(t)\right) \\ &= \frac{d\mathbf{L}(t)}{dt}. \end{aligned}$$

13.4.6.5 Solving the Equations of Angular Motion in Three Dimensions

When solving the equations of angular motion in three dimensions, we might be tempted to take exactly the same approach we used for linear motion and two-dimensional angular motion. We might guess that the differential equations of motion should be written

Angular 3D?	Linear:
$\mathbf{N}_{\text{net}}(t) = \mathbf{I}\dot{\boldsymbol{\omega}}(t)$	$\mathbf{F}_{\text{net}}(t) = m\dot{\mathbf{v}}(t)$
$\boldsymbol{\omega}(t) = \dot{\boldsymbol{\theta}}(t)$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t),$

and using the explicit Euler method, we might guess that the approximate solutions to these ODEs would look something like this:

Angular 3D?	Linear:
$\boldsymbol{\omega}(t_2) = \boldsymbol{\omega}(t_1) + \mathbf{I}^{-1}\mathbf{N}_{\text{net}}(t_1)\Delta t$	$\mathbf{v}(t_2) = \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t)\Delta t$
$\boldsymbol{\theta}(t_2) = \boldsymbol{\theta}(t_1) + \boldsymbol{\omega}(t_1)\Delta t$	$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t.$

然而，这实际上是不正确的。三维角运动的微分方程与线性和二维角运动的微分方程有两个重要区别：

1. 我们不是求解角速度ω，而是直接求解角动量L。然后，我们利用I和L计算角速度矢量，将其作为次级量。这样做是因为角动量守恒，而角速度不守恒。

2. 在给定角速度求解方向时，我们遇到一个问题：角速度是一个三元素矢量，而方向是一个四元素四元数。如何写出将四元数与矢量联系起来的常微分方程 (ODE)？答案是我们不能，至少不能直接写。但我们可以做的是将角速度矢量转换为四元数形式，然后应用一个看起来有点奇怪的方程，将方向四元数与角速度四元数联系起来。

事实证明，当我们将刚体的方向表示为四元数时，该四元数的导数与刚体的角速度矢量的关系如下。首先，我们构造一个角速度四元数。该四元数包含角速度矢量在 x 、 y 和 z 方向上的三个分量，其 w 分量设置为零：

$$\omega = \begin{bmatrix}\omega_x & \omega_y & \omega_z & 0\end{bmatrix}.$$

现在，将方向四元数与角速度四元数联系起来的微分方程（出于我们在此不讨论的原因）如下：

$$\frac{d\omega(t)}{dt} = \dot{\mathsf{q}}(t) = \tfrac{1}{2}\omega(t)\mathsf{q}(t).$$

这里需要记住的是，ω(t)是如上所述的角速度四元数，而乘积ω(t)q(t)是四元数乘积（详见第 5.4.2.1 节）。

因此，我们实际上需要将运动的 ODE 写如下（请注意，我还根据线性动量重新定义了线性 ODE，以强调两种情况之间的相似性）：

角度 3D:	线性:
$\mathbf{N}_{\text{net}}(t) = \dot{\mathbf{L}}(t)$	$\mathbf{F}_{\text{net}}(t) = \dot{\mathbf{p}}(t)$
$\boldsymbol{\omega}(t) = \mathbf{I}^{-1}\mathbf{L}(t)$	$\mathbf{v}(t) = m^{-}1\mathbf{p}(t)$
$\omega(t) = \begin{bmatrix}\boldsymbol{\omega}(t) & 0\end{bmatrix}$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t).$
$\frac{1}{2}\omega(t)\mathsf{q}(t) = \dot{\mathsf{q}}(t)$	

However, this is *not* actually correct. The differential equations of three-dimensional angular motion differ from their linear and two-dimensional angular counterparts in two important ways:

1. Instead of solving for the angular velocity $\boldsymbol{\omega}$, we solve for the angular momentum $\mathbf{L}$ directly. We then calculate the angular velocity vector as a secondary quantity using $\mathbf{I}$ and $\mathbf{L}$. We do this because angular momentum is conserved, while angular velocity is not.

2. When solving for the orientation given the angular velocity, we have a problem: The angular velocity is a three-element *vector*, while the orientation is a four-element *quaternion*. How can we write an ODE relating a quaternion to a vector? The answer is that we cannot, at least not directly. But what we can do is convert the angular velocity vector into quaternion form and then apply a slightly odd-looking equation that relates the orientation quaternion to the angular velocity quaternion.

It turns out that when we express a rigid body's orientation as a quaternion, the derivative of this quaternion is related to the body's angular velocity vector in the following way. First, we construct an *angular velocity quaternion*. This quaternion contains the three components of the angular velocity vector in x, y and z, with its w-component set to zero:

$$\omega = \begin{bmatrix}\omega_x & \omega_y & \omega_z & 0\end{bmatrix}.$$

Now the differential equation relating the orientation quaternion to the angular velocity quaternion is (for reasons we won't get into here) as follows:

$$\frac{d\omega(t)}{dt} = \dot{\mathsf{q}}(t) = \tfrac{1}{2}\omega(t)\mathsf{q}(t).$$

It's important to remember here that $\omega(t)$ is the angular velocity *quaternion* as described above and that the product $\omega(t)\mathsf{q}(t)$ is a *quaternion* product (see Section 5.4.2.1 for details).

So, we actually need to write the ODEs of motion as follows (note that I've recast the linear ODEs in terms of linear momentum as well, to underscore the similarities between the two cases):

Angular 3D:	Linear:
$\mathbf{N}_{\text{net}}(t) = \dot{\mathbf{L}}(t)$	$\mathbf{F}_{\text{net}}(t) = \dot{\mathbf{p}}(t)$
$\boldsymbol{\omega}(t) = \mathbf{I}^{-1}\mathbf{L}(t)$	$\mathbf{v}(t) = m^{-}1\mathbf{p}(t)$
$\omega(t) = \begin{bmatrix}\boldsymbol{\omega}(t) & 0\end{bmatrix}$	$\mathbf{v}(t) = \dot{\mathbf{r}}(t).$
$\frac{1}{2}\omega(t)\mathsf{q}(t) = \dot{\mathsf{q}}(t)$	

使用显式欧拉方法，三维角微分方程的最终近似解实际上如下：

$$
\begin{aligned}
\mathbf{L}(t_2) &= \mathbf{L}(t_1) + \mathbf{N}_{\text{net}}(t_1)\Delta t && \text{(vectors)}\\
&= \mathbf{L}(t_1) + \Delta t \sum_{\forall i} \left(\mathbf{r}_i \times \mathbf{F}_i(t_1)\right); && \text{(vectors)}\\
\omega(t_2) &= \begin{bmatrix}\mathbf{I}^{-1}\mathbf{L}(t_2) & 0\end{bmatrix}; && \text{(quaternions)}\\
\mathsf{q}(t_2) &= \mathsf{q}(t_1) + \tfrac{1}{2}\omega(t_1)\mathsf{q}(t_1)\Delta t. && \text{(quaternions)}
\end{aligned}
$$

方向四元数 q（t）应定期重新规范化，以逆转浮点误差不可避免的累积的影响。

与往常一样，这里仅使用显式欧拉方法作为示例。在实际发动机中，我们会采用速度 Verlet、RK4 或其他更稳定、更准确的数值方法。

13.4.7 碰撞响应

到目前为止，我们讨论的所有内容都假设刚体既不与任何物体发生碰撞，也不以任何其他方式限制其运动。当刚体相互碰撞时，动力学模拟必须采取措施，确保它们对碰撞做出真实的响应，并且在模拟步骤完成后不会处于相互穿透的状态。这被称为碰撞响应。

13.4.7.1 能源

在讨论碰撞响应之前，我们必须先理解能量的概念。当力使物体移动一段距离时，我们说该力做了功。功表示能量的变化，也就是说，力要么为刚体系统增加能量（例如爆炸），要么从系统带走能量（例如摩擦）。能量有两种形式。物体的势能 V 是它相对于力场（例如重力场或磁场）的位置所具有的能量。（例如，物体距离地球表面越高，它的重力势能就越大。）物体的动能 T 表示由于它相对于系统中其他物体运动而产生的能量。孤立物体系统的总能量 E = V + T 是守恒量，这意味着除非从系统中消耗能量或从系统外部添加能量，否则它将保持不变。

直线运动产生的动能可以写成

$$T_{\text{linear}} = \tfrac{1}{2}mv^2,$$

Using the explicit Euler method, the final approximate solution to the angular ODEs in three dimensions is actually as follows:

$$
\begin{aligned}
\mathbf{L}(t_2) &= \mathbf{L}(t_1) + \mathbf{N}_{\text{net}}(t_1)\Delta t && \text{(vectors)}\\
&= \mathbf{L}(t_1) + \Delta t \sum_{\forall i} \left(\mathbf{r}_i \times \mathbf{F}_i(t_1)\right); && \text{(vectors)}\\
\omega(t_2) &= \begin{bmatrix}\mathbf{I}^{-1}\mathbf{L}(t_2) & 0\end{bmatrix}; && \text{(quaternions)}\\
\mathsf{q}(t_2) &= \mathsf{q}(t_1) + \tfrac{1}{2}\omega(t_1)\mathsf{q}(t_1)\Delta t. && \text{(quaternions)}
\end{aligned}
$$

The orientation quaternion $\mathsf{q}(t)$ should be renormalized periodically to reverse the effects of the inevitable accumulation of floating-point error.

As always, the explicit Euler method is being used here just as an example. In a real engine, we would employ velocity Verlet, RK4 or some other more-stable and more-accurate numerical method.

13.4.7 Collision Response

Everything we've discussed so far assumes that our rigid bodies are neither colliding with anything, nor is their motion constrained in any other way. When bodies collide with one another, the dynamics simulation must take steps to ensure that they respond realistically to the collision and that they are never left in a state of interpenetration after the simulation step has been completed. This is known as *collision response*.

13.4.7.1 Energy

Before we discuss collision response, we must understand the concept of *energy*. When a force moves a body over a distance, we say that the force does *work*. Work represents a change in energy—that is, a force either adds energy to a system of rigid bodies (e.g., an explosion) or it removes energy from the system (e.g., friction). Energy comes in two forms. The potential energy V of a body is the energy it has simply because of where it is relative to a force field such as a gravitational or a magnetic field. (For example, the higher up a body is above the surface of the Earth, the more gravitational potential energy it has.) The kinetic energy of a body T represents the energy arising from the fact that it is moving relative to other bodies in a system. The total energy $E = V + T$ of an isolated system of bodies is a *conserved* quantity, meaning that it remains constant unless energy is being drained from the system or added from outside the system.

The kinetic energy arising from linear motion can be written

$$T_{\text{linear}} = \tfrac{1}{2}mv^2,$$

或者用线性动量和速度矢量来表示：

$$T_{\text{linear}} = \frac{1}{2}\mathbf{p} \cdot \mathbf{v}.$$

类似地，物体旋转运动产生的动能如下：

$$T_{\text{angular}} = \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}.$$

在解决各种物理问题时，能量及其守恒都是极其有用的概念。下一节我们将探讨能量在确定碰撞响应中所起的作用。

13.4.7.2 脉冲碰撞响应

在现实世界中，当两个物体发生碰撞时，会发生一系列复杂的事件。物体会轻微压缩，然后反弹，改变速度，并在此过程中以声音和热量的形式损失能量。大多数实时刚体动力学模拟都使用一个简单的模型来近似所有这些细节，该模型基于对碰撞物体动量和动能的分析，即无摩擦瞬时碰撞的牛顿恢复定律。

它对碰撞做出以下简化假设：

• 碰撞力在极短的时间内起作用，将其转化为我们所说的理想冲量。这会导致物体的速度因碰撞而瞬间改变。

• 物体表面的接触点没有摩擦。
换句话说，碰撞过程中分离物体的冲量垂直于两个表面——碰撞冲量没有切向分量。（当然，这只是一种理想化的情况；我们将在13.4.7.5节讨论摩擦力。）

• 碰撞过程中物体之间复杂的亚分子相互作用的性质可以用一个称为恢复系数的量来近似，通常用符号 ε 表示。该系数描述了碰撞过程中损失的能量。当 $\varepsilon = 1$ 时，碰撞是完全弹性的，没有能量损失。（想象一下两个台球在空中相撞。）当 $\varepsilon = 0$ 时，碰撞是完全非弹性的，也称为完全塑性，两个物体的动能都会损失。碰撞后，物体会粘在一起，继续沿着碰撞前它们相互重心移动的方向运动。（想象一下油灰碎片撞在一起。）

or in terms of the linear momentum and velocity vectors:

$$T_{\text{linear}} = \frac{1}{2}\mathbf{p} \cdot \mathbf{v}.$$

Analogously, the kinetic energy arising from a body's rotational motion is as follows:

$$T_{\text{angular}} = \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}.$$

Energy and its conservation can be extremely useful concepts when solving all sorts of physics problems. We'll see the role that energy plays in the determination of collision responses in the following section.

13.4.7.2 Impulsive Collision Response

When two bodies collide in the real world, a complex set of events takes place. The bodies compress slightly and then rebound, changing their velocities and losing energy to sound and heat in the process. Most real-time rigid body dynamics simulations approximate all of these details with a simple model based on an analysis of the momenta and kinetic energies of the colliding objects, called *Newton's law of restitution for instantaneous collisions with no friction*. It makes the following simplifying assumptions about the collision:

- The collision force acts over an infinitesimally short period of time, turning it into what we call an idealized *impulse*. This causes the velocities of the bodies to change *instantaneously* as a result of the collision.
- There is no friction at the point of contact between the objects' surfaces. This is another way of saying that the impulse acting to separate the bodies during the collision is normal to both surfaces—there is no tangential component to the collision impulse. (This is just an idealization of course; we'll get to friction in Section 13.4.7.5.)
- The nature of the complex submolecular interactions between the bodies during the collision can be approximated by a single quantity known as the *coefficient of restitution*, customarily denoted by the symbol ε. This coefficient describes how much energy is lost during the collision. When $\varepsilon = 1$, the collision is perfectly elastic, and no energy is lost. (Picture two billiard balls colliding in mid-air.) When $\varepsilon = 0$, the collision is perfectly *inelastic*, also known as perfectly *plastic* and the kinetic energy of both bodies is lost. The bodies will stick together after the collision, continuing to move in the direction that their mutual center of mass had been moving before the collision. (Picture pieces of putty being slammed together.)

所有碰撞分析都基于线性动量守恒的思想。因此，对于两个物体 1 和 2，我们可以写成

$$\begin{aligned} \mathbf{p}_1 + \mathbf{p}_2 &= \mathbf{p}_1' + \mathbf{p}_2', \qquad \text{or} \\ m_1\mathbf{v}_1 + m_2\mathbf{v}_2 &= m_1'\mathbf{v}_1' + m_2'\mathbf{v}_2' \end{aligned}$$

其中撇号表示碰撞后的动量和速度。系统的动能也守恒，但我们必须考虑由于热量和声音而损失的能量，因此必须引入一个额外的能量损失项 T lost：

$$\tfrac{1}{2}m_1v_1^2 + \tfrac{1}{2}m_2v_2^2 = \tfrac{1}{2}m_1'v_1'^2 + \tfrac{1}{2}m_2'v_2'^2 + T_{\text{lost}}.$$

如果碰撞是完全弹性的，则能量损失T lost 为零。如果碰撞是完全塑性的，则能量损失等于系统的原始动能，初始动能总和为零，碰撞后物体粘在一起。

为了利用牛顿恢复定律解决碰撞问题，我们对两个物体施加一个理想化的冲量。冲量就像一种在极短时间内作用的力，从而导致被施加物体的速度发生瞬时变化。我们可以用符号Δ p 表示冲量，因为它表示动量的变化（Δ p = m Δ v）。然而，大多数物理教材使用符号 ˆ p（发音为“p-hat”），所以我们也照做。

因为我们假设碰撞中不存在摩擦力，所以冲量矢量在接触点处必定垂直于两个表面。换句话说，ˆ p = ˆ pn，其中 n 是垂直于两个表面的单位矢量。如图 13.27 所示。假设表面法线指向物体 1，则物体 1 受到的冲量为 ˆ p，物体 2 受到的冲量为大小相等但方向相反的冲量 − ˆ p。因此，碰撞后两个物体的动量可以用碰撞前的动量和冲量 ˆ p 表示，如下所示：

$$\begin{aligned} \mathbf{p}_1' &= \mathbf{p}_1 + \hat{\mathbf{p}} & \mathbf{p}_2' &= \mathbf{p}_2 - \hat{\mathbf{p}} \\ m_1\mathbf{v}_1' &= m_1\mathbf{v}_1 + \hat{\mathbf{p}} & m_2\mathbf{v}_2' &= m_2\mathbf{v}_2 - \hat{\mathbf{p}} \\ \mathbf{v}_1' &= \mathbf{v}_1 + \frac{\hat{\mathbf{p}}}{m_1}\mathbf{n} & \mathbf{v}_2' &= \mathbf{v}_2 + \frac{\hat{\mathbf{p}}}{m_2}\mathbf{n}. \end{aligned} \tag{13.9}$$

恢复系数提供了碰撞前后物体相对速度之间的关键关系。假设物体的质心在碰撞前后都有速度，则恢复系数ε定义如下：

$$(\mathbf{v}_2' - \mathbf{v}_1') = \varepsilon(\mathbf{v}_2 - \mathbf{v}_1). \tag{13.10}$$

All collision analysis is based around the idea that linear momentum is conserved. So for two bodies 1 and 2, we can write

$$\begin{aligned} \mathbf{p}_1 + \mathbf{p}_2 &= \mathbf{p}_1' + \mathbf{p}_2', \qquad \text{or} \\ m_1\mathbf{v}_1 + m_2\mathbf{v}_2 &= m_1'\mathbf{v}_1' + m_2'\mathbf{v}_2' \end{aligned}$$

where the primed symbols represent the momenta and velocities after the collision. The kinetic energy of the system is conserved as well, but we must account for the energy lost due to heat and sound by introducing an additional energy loss term T_{lost}:

$$\tfrac{1}{2}m_1v_1^2 + \tfrac{1}{2}m_2v_2^2 = \tfrac{1}{2}m_1'v_1'^2 + \tfrac{1}{2}m_2'v_2'^2 + T_{\text{lost}}.$$

If the collision is perfectly *elastic*, the energy loss T_{lost} is zero. If it is perfectly *plastic*, the energy loss is equal to the original kinetic energy of the system, the primed kinetic energy sum becomes zero and the bodies stick together after the collision.

To resolve a collision using Newton's law of restitution, we apply an idealized *impulse* to the two bodies. An impulse is like a force that acts over an infinitesimally short period of time and thereby causes an instantaneous change in the velocity of the body to which it is applied. We could denote an impulse with the symbol $\Delta\mathbf{p}$, since it is a change in momentum ($\Delta\mathbf{p} = m\Delta\mathbf{v}$). However, most physics texts use the symbol $\hat{\mathbf{p}}$ (pronounced "p-hat") instead, so we'll do the same.

Because we assume that there is no friction involved in the collision, the impulse vector must be normal to both surfaces at the point of contact. In other words, $\hat{\mathbf{p}} = \hat{p}\mathbf{n}$, where $\mathbf{n}$ is the unit vector normal to both surfaces. This is illustrated in Figure 13.27. If we assume that the surface normal points toward body 1, then body 1 experiences an impulse of $\hat{\mathbf{p}}$, and body 2 experiences an equal but opposite impulse $-\hat{\mathbf{p}}$. Hence, the momenta of the two bodies after the collision can be written in terms of their momenta prior to the collision and the impulse $\hat{\mathbf{p}}$ as follows:

$$\begin{aligned} \mathbf{p}_1' &= \mathbf{p}_1 + \hat{\mathbf{p}} & \mathbf{p}_2' &= \mathbf{p}_2 - \hat{\mathbf{p}} \\ m_1\mathbf{v}_1' &= m_1\mathbf{v}_1 + \hat{\mathbf{p}} & m_2\mathbf{v}_2' &= m_2\mathbf{v}_2 - \hat{\mathbf{p}} \\ \mathbf{v}_1' &= \mathbf{v}_1 + \frac{\hat{\mathbf{p}}}{m_1}\mathbf{n} & \mathbf{v}_2' &= \mathbf{v}_2 + \frac{\hat{\mathbf{p}}}{m_2}\mathbf{n}. \end{aligned} \tag{13.9}$$

The *coefficient of restitution* provides the key relationship between the relative velocities of the bodies before and after the collision. Given that the centers of mass of the bodies have velocities before the collision and afterward, the coefficient of restitution ε is defined as follows:

$$(\mathbf{v}_2' - \mathbf{v}_1') = \varepsilon(\mathbf{v}_2 - \mathbf{v}_1). \tag{13.10}$$

图 13.27 在无摩擦碰撞中，冲量作用于接触点处两个表面的法线。这条线由单位法向量 n 定义。

暂时假设物体不能旋转，解方程 (13.9) 和 (13.10) 可得

$$\hat{\mathbf{p}} = \hat{p}\mathbf{n} = \frac{(\varepsilon+1)(\mathbf{v}_2\cdot\mathbf{n} - \mathbf{v}_1\cdot\mathbf{n})}{\dfrac{1}{m_1}+\dfrac{1}{m_2}}\mathbf{n}.$$

请注意，如果恢复系数为 1（完全弹性碰撞），并且物体 2 的质量实际上是无限的（就像混凝土车道一样），那么（1/m 2）= 0，v 2 = 0，并且该表达式简化为另一个物体的速度矢量关于接触法线的反射，正如我们所期望的：

$$\begin{aligned}\hat{\mathbf{p}} &= -2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n};\\ \mathbf{v}_1' &= \frac{\mathbf{p}_1+\mathbf{p}_2}{m_1}\\ &= \frac{m_1\mathbf{v}_1 - 2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n}}{m_1}\\ &= \mathbf{v}_1 - 2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n}.\end{aligned}$$

当我们考虑物体的旋转时，解决方案会变得有些复杂。在这种情况下，我们需要考虑两个物体接触点的速度，而不是它们质心的速度，并且需要计算冲量，以便产生真实的碰撞旋转效果。我们在这里不做详细介绍，但 Chris Hecker 的文章（网址为 http://chrishecker.com/images/e/e7/Gdmphys3.pdf）对碰撞响应的线性和旋转方面进行了出色的描述。碰撞响应背后的理论在 [17] 中有更详细的解释。

Figure 13.27. In a frictionless collision, the impulse acts along a line normal to both surfaces at the point of contact. This line is defined by the unit normal vector **n**.

Solving Equations (13.9) and (13.10) under the temporary assumption that the bodies cannot rotate yields

$$\hat{\mathbf{p}} = \hat{p}\mathbf{n} = \frac{(\varepsilon+1)(\mathbf{v}_2\cdot\mathbf{n} - \mathbf{v}_1\cdot\mathbf{n})}{\dfrac{1}{m_1}+\dfrac{1}{m_2}}\mathbf{n}.$$

Notice that if the coefficient of restitution is one (perfectly elastic collision) and if the mass of body 2 is effectively infinite (as it would be for, say, a concrete driveway), then $(1/m_2) = 0, \mathbf{v}_2 = 0$, and this expression reduces to a reflection of the other body's velocity vector about the contact normal, as we'd expect:

$$\begin{aligned}\hat{\mathbf{p}} &= -2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n};\\ \mathbf{v}_1' &= \frac{\mathbf{p}_1+\mathbf{p}_2}{m_1}\\ &= \frac{m_1\mathbf{v}_1 - 2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n}}{m_1}\\ &= \mathbf{v}_1 - 2m_1(\mathbf{v}_1\cdot\mathbf{n})\mathbf{n}.\end{aligned}$$

The solution gets a bit hairier when we take the rotations of the bodies into account. In this case, we need to look at the velocities of the points of contact on the two bodies rather than the velocities of their centers of mass, and we need to calculate the impulse in such a way as to impart a realistic rotational effect as a result of the collision. We won't get into the details here, but Chris Hecker's article, available at http://chrishecker.com/images/e/e7/Gdmphys3.pdf, does an excellent job of describing both the linear and the rotational aspects of collision response. The theory behind collision response is explained more fully in [17].

13.4.7.3 惩罚部队

另一种碰撞响应方法是在模拟中引入称为惩罚力的假想力。惩罚力的作用类似于连接到两个刚刚相互穿透的物体之间的接触点的刚性阻尼弹簧。这种力会在短暂但有限的时间内引发所需的碰撞响应。使用这种方法，弹簧常数 k 可以有效地控制相互穿透的持续时间，而阻尼系数 b 的作用有点像恢复系数。当 b = 0 时，没有阻尼——没有能量损失，碰撞是完全弹性的。随着 b 的增加，碰撞变得更具塑性。

让我们简要地看一下使用惩罚力方法解决碰撞的优缺点。积极的一面是，惩罚力易于实现和理解。当三个或更多物体相互穿插时，它们也能很好地发挥作用。当一次解决一对碰撞时，这个问题非常难以解决。一个很好的例子是索尼 PS3 的演示，其中大量的橡皮鸭被倒入浴缸——尽管碰撞次数非常多，但模拟仍然很好且稳定。

惩罚力方法是实现这一目标的好方法。
遗憾的是，由于惩罚力响应的是穿透力（即相对位置），而非相对速度，因此力的方向可能与我们直觉预期的方向不一致，尤其是在高速碰撞中。一个典型的例子是一辆汽车迎面撞上一辆卡车。汽车较低，而卡车较高。仅使用惩罚力方法，很容易得出这样的结果：惩罚力是垂直的，而不是我们根据两辆车的速度预期的水平方向。这可能导致卡车车头朝天，而汽车则从卡车下方驶过。

一般来说，惩罚力技术在低速碰撞中效果良好，但在物体快速移动时效果不佳。可以将惩罚力方法与其他碰撞解决方法相结合，以便在大量相互穿透时的稳定性与高速下的响应性和更直观的行为之间取得平衡。

13.4.7.4 使用约束解决碰撞

正如我们将在 13.4.8 节中探讨的那样，大多数物理系统允许对模拟中物体的运动施加各种约束。如果将碰撞视为不允许物体相互穿透的约束，那么只需运行模拟的通用约束求解器即可解决。如果约束求解器速度快且能产生高

13.4.7.3 Penalty Forces

Another approach to collision response is to introduce imaginary forces called *penalty forces* into the simulation. A penalty force acts like a stiff damped spring attached to the contact points between two bodies that have just interpenetrated. Such a force induces the desired collision response over a short but finite period of time. Using this approach, the spring constant k effectively controls the duration of the interpenetration, and the damping coefficient b acts a bit like the restitution coefficient. When $b = 0$, there is no damping—no energy is lost, and the collision is perfectly elastic. As b increases, the collision becomes more plastic.

Let's take a brief look at some of the pros and cons of the penalty force approach to resolving collisions. On the positive side, penalty forces are easy to implement and understand. They also work well when three or more bodies are interpenetrating each other. This problem is very difficult to solve when resolving collisions one pair at a time. A good example is the Sony PS3 demo in which a huge number of rubber duckies are poured into a bathtub—the simulation was nice and stable despite the very large number of collisions. The penalty force method is a great way to achieve this.

Unfortunately, because penalty forces respond to penetration (i.e., relative position) rather than to relative velocity, the forces may not align with the direction we would intuitively expect, especially during a high-speed collision. A classic example is a car driving head-on into a truck. The car is low while the truck is tall. Using only the penalty force method, it is easy to arrive at a situation in which the penalty force is vertical, rather than horizontal as we would expect given the velocities of the two vehicles. This can cause the truck to pop its nose up into the air while the car drives under it.

In general, the penalty force technique works well for low-speed impacts, but it does not work well at all when objects are moving quickly. It is possible to combine the penalty force method with other collision resolution approaches in order to strike a balance between stability in the presence of large numbers of interpenetrations and responsiveness and more-intuitive behavior at high velocities.

13.4.7.4 Using Constraints to Resolve Collisions

As we'll investigate in Section 13.4.8, most physics systems permit various kinds of constraints to be imposed on the motion of the bodies in the simulation. If collisions are treated as constraints that disallow object interpenetration, then they can be resolved by simply running the simulation's general-purpose constraint solver. If the constraint solver is fast and produces high-

高质量的视觉效果，这可以成为解决碰撞的有效方法。

13.4.7.5 摩擦

摩擦力是两个持续接触的物体之间产生的一种力，阻碍它们相对运动。摩擦力有很多种类型。静摩擦力是指当静止物体试图沿表面滑动时感受到的阻力。动摩擦力是指物体实际相对运动时产生的阻力。滑动摩擦力是一种动摩擦力，它阻碍物体沿表面滑动。滚动摩擦力是一种静摩擦力或动摩擦力，作用于轮子或其他圆形物体与其滚动表面的接触点。当表面非常粗糙时，滚动摩擦力刚好足以使轮子滚动而不滑动，这时它就是一种静摩擦力。如果表面比较光滑，轮子可能会打滑，这时就出现了一种动摩擦力。碰撞摩擦力是指两个运动物体碰撞时，在接触点瞬间产生的摩擦力。（这就是我们在 13.4.7.1 节讨论牛顿恢复定律时忽略的摩擦力。）各种约束也可能产生摩擦力。例如，生锈的铰链或轴可能会通过引入摩擦扭矩来阻止其转动。

让我们看一个例子来了解摩擦力的本质。
线性滑动摩擦力与物体重量中作用于其滑动表面的垂直分量成正比。物体的重量就是重力产生的力，G = mg，该力始终向下。该力在与水平面成 θ 角的倾斜表面上的垂直分量就是 GN = mg cos θ。摩擦力 f 为

$$f = \mu mg\cos\theta,$$

其中比例常数 μ 称为摩擦系数。该力沿切向作用于表面，方向与物体试图或实际运动的方向相反。如图 13.28 所示。

图 13.28 还显示了作用于表面的切向重力分量 GT = mg sin θ 。该力倾向于使物体沿平面向下加速，但由于存在滑动摩擦，该力会被 f 抵消。因此，作用于表面的切向净力为

$$F_{\text{net}} = G_T - f = mg(\sin\theta - \mu\cos\theta).$$

如果倾斜角使得括号中的表达式为零，则物体将以恒定速度滑动（如果已经在运动）或处于静止状态。如果

quality visual results, this can be an effective way to resolve collisions.

13.4.7.5 Friction

Friction is a force that arises between two bodies that are in continuous contact, resisting their movement relative to one another. There are a number of types of friction. *Static friction* is the resistance one feels when trying to start a stationary object sliding along a surface. *Dynamic friction* is a resisting force that arises when objects are actually moving relative to one another. *Sliding friction* is a type of dynamic friction that resists movement when an object slides along a surface. *Rolling friction* is a type of static or dynamic friction that acts at the point of contact between a wheel or other round object and the surface it is rolling on. When the surface is very rough, the rolling friction is exactly strong enough to cause the wheel to roll without sliding, and it acts as a form of static friction. If the surface is somewhat smooth, the wheel may slip, and a dynamic form of rolling friction comes into play. *Collision friction* is the friction that acts instantaneously at the point of contact when two bodies collide while moving. (This is the friction force that we ignored when discussing Newton's law of restitution in Section 13.4.7.1.) Various kinds of *constraints* can have friction as well. For example, a rusted hinge or axle might resist being turned by introducing a friction torque.

Let's look at an example to understand the essence of how friction works. Linear sliding friction is proportional to the component of an object's weight that is acting normal to the surface on which it is sliding. The weight of an object is just the force due to gravity, $\mathbf{G} = m\mathbf{g}$, which is always directed downward. The component of this force normal to an inclined surface that makes an angle θ with the horizontal is just $G_N = mg\cos\theta$. The friction force f is then

$$f = \mu mg\cos\theta,$$

where the constant of proportionality μ is called the *coefficient of friction*. This force acts tangentially to the surface, in a direction opposite to the attempted or actual motion of the object. This is illustrated in Figure 13.28.

Figure 13.28 also shows the component of the gravitational force acting tangent to the surface, $G_T = mg\sin\theta$. This force tends to make the object accelerate down the plane, but in the presence of sliding friction, it is counteracted by f. Hence, the net force tangent to the surface is

$$F_{\text{net}} = G_T - f = mg(\sin\theta - \mu\cos\theta).$$

If the angle of inclination is such that the expression in parentheses is zero, the object will slide at a constant speed (if already moving) or be at rest. If the

表达式大于零时，物体将沿表面加速向下运动。如果表达式小于零，物体将减速并最终静止。

13.4.7.6 焊接

当物体在多边形汤（Polygon Soup）上滑动时，还会出现另一个问题。回想一下，多边形汤顾名思义，就是由本质上不相关的多边形（通常是三角形）组成的汤。当物体从多边形汤中的一个三角形滑动到另一个三角形时，碰撞检测系统会产生额外的虚假接触，因为它会认为该物体即将碰到下一个三角形的边。如图 13.29 所示。

这个问题有很多解决方案。一种方法是分析接触集合，并丢弃那些看似虚假的接触，这种方法基于各种启发式算法，并可能结合前一帧中物体接触点的一些信息（例如，如果我们知道物体沿着表面滑动，并且由于物体靠近其当前三角形的边缘而产生了接触法线，则丢弃该接触法线）。Havok 4.5 之前的版本就采用了这种方法。

从 Havok 4.5 开始，我们实现了一项新技术，该技术本质上是用三角形邻接信息来注释网格。因此，碰撞检测系统能够“知道”哪些边是内部边，并能够可靠且快速地丢弃伪碰撞。Havok 将此解决方案描述为焊接，因为实际上多边形汤中三角形的边是相互焊接在一起的。

13.4.7.7 休息、岛屿和睡觉

当能量通过摩擦、阻尼或其他方式从模拟系统中移除时，运动物体最终会静止下来。这似乎是模拟的自然结果——某种会从运动微分方程中“脱离”的东西。不幸的是，在真实的计算机模拟中

图 13.28。摩擦力 f 与物体重量的法向分量成正比。比例常数 μ 称为摩擦系数。

expression is greater than zero, the object will accelerate down the surface. If it is less than zero, the object will decelerate and eventually come to rest.

13.4.7.6 Welding

An additional problem arises when an object is sliding across a polygon soup. Recall that a polygon soup is just what its name implies—a soup of essentially unrelated polygons (usually triangles). As an object slides from one triangle of this soup to the next, the collision detection system will generate additional spurious contacts because it will think that the object is about to hit the edge of the next triangle. This is illustrated in Figure 13.29.

There are a number of solutions to this problem. One is to analyze the set of contacts and discard ones that appear to be spurious, based on various heuristics and possibly some knowledge of the object's contacts on a previous frame (e.g., if we know the object was sliding along a surface and a contact normal arises that is due to the object being near the edge of its current triangle, then discard that contact normal). Versions of Havok prior to 4.5 employed this approach.

Starting with Havok 4.5, a new technique was implemented that essentially annotates the mesh with triangle adjacency information. The collision detection system therefore "knows" which edges are interior edges and can discard spurious collisions reliably and quickly. Havok describes this solution as *welding*, because in effect the edges of the triangles in the poly soup are welded to one another.

13.4.7.7 Coming to Rest, Islands and Sleeping

When energy is removed from a simulated system via friction, damping or other means, moving objects will eventually come to rest. This seems like a natural consequence of the simulation—something that would just "fall out" of the differential equations of motion. Unfortunately, in a real computerized

Figure 13.28. The force of friction f is proportional to the normal component of the object's weight. The proportionality constant μ is called the coefficient of friction.

图 13.29. 当物体在两个相邻三角形之间滑动时，可能会与新三角形的边缘产生虚假接触。

在模拟中，静止从来都不是那么简单。各种因素，例如浮点误差、恢复力计算的不准确性以及数值不稳定性，都可能导致物体永远抖动，而不是像它们应该的那样静止下来。因此，大多数物理引擎使用各种启发式方法来检测物体何时振荡而不是像它们应该的那样静止下来。可以从系统中移除额外的能量，以确保这些物体最终稳定下来，或者一旦它们的平均速度降至阈值以下，就可以直接将它们停止。

当一个物体真正停止运动（达到平衡状态）时，就没有必要继续每帧对其运动方程进行积分。为了优化性能，大多数物理引擎允许模拟中的动态物体进入睡眠状态。这会暂时将它们排除在模拟之外，尽管从碰撞的角度来看，睡眠状态的物体仍然处于活动状态。如果任何力或冲量开始作用于睡眠状态的物体，或者该物体失去维持其平衡的某个接触点，它将被唤醒，以便恢复其动态模拟。

睡眠标准

判断身体是否处于睡眠状态的标准有很多。要在所有情况下都做出稳健的判断并非易事。例如，一个长摆的角动量可能非常低，但在屏幕上仍然可见地移动。

最常用的平衡检测标准包括：

- 身体受到支撑。这意味着它具有三个或更多个接触点（或一个或多个平面接触），使其能够与重力以及可能影响它的任何其他力达到平衡。
- 身体的线性和角动量低于预定的阈值。

Figure 13.29. When an object slides between two adjacent triangles, spurious contacts with the new triangle's edge can be generated.

simulation, coming to rest is never quite that simple. Various factors such as floating-point error, inaccuracies in the calculation of restitution forces and numerical instability can cause objects to jitter forever rather than coming to rest as they should. For this reason, most physics engines use various heuristic methods to detect when objects are oscillating instead of coming to rest as they should. Additional energy can be removed from the system to ensure that such objects eventually settle down, or they can simply be stopped abruptly once their average velocity drops below a threshold.

When an object really does stop moving (finds itself in a state of equilibrium), there is no reason to continue integrating its equations of motion every frame. To optimize performance, most physics engines allow dynamic objects in the simulation to be *put to sleep*. This excludes them from the simulation temporarily, although sleeping objects are still active from a collision standpoint. If any force or impulse begins acting on a sleeping object, or if the object loses one of the contacts that was holding it in equilibrium, it will be awoken so that its dynamic simulation can be resumed.

Sleep Criteria

Various criteria can be used to determine whether or not a body qualifies for sleep. It's not always easy to make this determination in a robust manner for all situations. For example, a long pendulum might have very low angular momentum and yet still be moving visibly on-screen.

The most commonly used criteria for equilibrium detection include:

- The body is *supported*. This means it has three or more contact points (or one or more *planar* contacts) that allow it to attain equilibrium with gravity and any other forces that might be affecting it.
- The body's *linear and angular momentum* are below a predefined threshold.

• 线性和角动量的运行平均值低于预定阈值。

• 物体的总动能（T = 1 2 p · v + 1 2 L · ω）低于预定阈值。动能通常按质量归一化，以便所有物体（无论其质量如何）都可以使用单一阈值。

即将进入睡眠状态的身体的运动可能会逐渐减弱，以便平稳停止而不是突然停止。

模拟岛

Havok 和 PhysX 都通过自动将正在交互或近期可能交互的对象分组到称为“模拟岛”的集合中，进一步优化了性能。每个模拟岛都可以独立于其他所有模拟岛进行模拟——这种方法非常有利于缓存一致性优化和并行处理。

Havok 和 PhysX 都会让整个岛屿进入睡眠状态，而不是单个刚体。这种方法各有利弊。当一组交互对象都进入睡眠状态时，性能提升显然更大。另一方面，即使岛屿中只有一个对象处于唤醒状态，整个岛屿也会处于唤醒状态。总的来说，优点似乎大于缺点，因此我们很可能会在这些 SDK 的未来版本中继续看到模拟岛屿的设计。

13.4.8 约束

不受约束的刚体具有六个自由度 (DOF)：它可以在三维空间中平移，也可以绕三个笛卡尔坐标轴旋转。约束会限制对象的运动，从而部分或完全降低其自由度。约束可用于模拟游戏中各种有趣的行为。以下是一些示例：

• 摆动的枝形吊灯（点对点约束）；
• 一扇可以被踢、被摔、被吹掉铰链的门（铰链约束）；

• 车辆的车轮组件（带有用于悬架的阻尼弹簧的轴约束）；

• 牵引拖车的火车或汽车（刚性弹簧/杆约束）；
• 绳索或链条（由硬弹簧或硬杆组成的链条）；以及

- A *running average* of the linear and angular momentum are below a predefined threshold.
- The total *kinetic energy* of the body ($T = \frac{1}{2}\mathbf{p} \cdot \mathbf{v} + \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}$) is below a predefined threshold. The kinetic energy is usually mass-normalized so that a single threshold can be used for all bodies regardless of their masses.

The motion of a body that is about to go to sleep might be *progressively damped* so that it comes to a smooth stop rather than stopping abruptly.

Simulation Islands

Both Havok and PhysX further optimize their performance by automatically grouping objects that either are interacting or have the potential to interact in the near future into sets called *simulation islands*. Each simulation island can be simulated independently of all the other islands—an approach that is highly conducive to cache coherency optimizations and parallel processing.

Havok and PhysX both put entire islands to sleep rather than individual rigid bodies. This approach has its pros and cons. The performance boost is obviously larger when a whole group of interacting objects can be put to sleep. On the other hand, if even one object in an island is awake, the entire island is awake. Overall, it seems that the pros tend to outweigh the cons, so the simulation island design is one we're likely to continue to see in future versions of these SDKs.

13.4.8 Constraints

An unconstrained rigid body has six degrees of freedom (DOF): It can translate in three dimensions, and it can rotate about the three Cartesian axes. *Constraints* restrict an object's motion, reducing its degrees of freedom either partially or completely. Constraints can be used to model all sorts of interesting behaviors in a game. Here are a few examples:

- a swinging chandelier (point-to-point constraint);
- a door that can be kicked, slammed, blown of its hinges (hinge constraint);
- a vehicle's wheel assembly (axle constraint with damped springs for suspension);
- a train or a car pulling a trailer (stiff spring/rod constraint);
- a rope or chain (chain of stiff springs or rods); and

图 13.31 刚性弹簧约束要求物体 A 上的点与物体 B 上的点之间间隔用户指定的距离。

图 13.30。 A 点对点约束要求物体 A 上的点与物体 B 上的点对齐。

• 布娃娃（模仿人体骨骼中各个关节行为的专门约束）。

在接下来的部分中，我们将简要研究这些以及物理 SDK 通常提供的一些其他最常见的约束。

13.4.8.1 点对点约束

点对点约束是最简单的约束类型。它的作用类似于球窝关节——只要一个物体上的指定点与另一个物体上的指定点对齐，物体就可以以任何方式移动。如图 13.30 所示。

13.4.8.2 硬弹簧

刚性弹簧约束与点对点约束非常相似，不同之处在于它使两点之间保持指定的距离。这种约束就像一根看不见的杆，位于两个约束点之间。图 13.31 展示了这种约束。

13.4.8.3 铰链约束

铰链约束将旋转运动限制为仅绕铰链轴的单个自由度。无限铰链的作用类似于轴，允许受约束的物体完成无限次的完整旋转。通常定义有限铰链，使其只能绕一个允许的轴在预定的角度范围内移动。例如，单向门只能移动 180 度弧度，否则它会穿过相邻的墙壁。同样，双向门被限制为移动 ± 180 度弧度。铰链约束还可以以扭矩的形式赋予一定程度的摩擦力，以抵抗绕铰链轴的旋转。有限铰链约束如图 13.32 所示。

Figure 13.31. A stiff spring constraint requires that a point on body A be separated from a point on body B by a user-specified distance.

Figure 13.30. A point-to-point constraint requires that a point on body A aligns with a point on body B.

- a rag doll (specialized constraints that mimic the behavior of various joints in the human skeleton).

In the sections that follow, we'll briefly investigate these and some of the other most common kinds of constraints typically provided by a physics SDK.

13.4.8.1 Point-to-Point Constraints

A point-to-point constraint is the simplest type of constraint. It acts like a ball-and-socket joint—bodies can move in any way they like, as long as a specified point on one body lines up with a specified point on the other body. This is illustrated in Figure 13.30.

13.4.8.2 Stiff Springs

A stiff spring constraint is a lot like a point-to-point constraint except that it keeps the two points separated by a specified distance. This kind of constraint acts like an invisible rod between the two constrained points. Figure 13.31 illustrates this constraint.

13.4.8.3 Hinge Constraints

A hinge constraint limits rotational motion to only a single degree of freedom, about the hinge's axis. An *unlimited hinge* acts like an axle, allowing the constrained object to complete an unlimited number of full rotations. It's common to define *limited hinges* that can only move through a predefined range of angles about the one allowed axis. For example, a one-way door can only move through a 180 degree arc, because otherwise it would pass through the adjacent wall. Likewise, a two-way door is constrained to move through a ±180 degree arc. Hinge constraints may also be given a degree of friction in the form of a torque that resists rotation about the hinge's axis. A limited hinge constraint is shown in Figure 13.32.

13.4.8.4 棱柱约束

棱柱约束的作用类似于活塞：受约束物体的运动被限制为一个平移自由度。棱柱约束可能允许或不允许绕活塞的平移轴旋转。当然，棱柱约束可以是有限制的，也可以是无限制的，并且可能包含或不包含摩擦力。图 13.33 展示了一个棱柱约束。

13.4.8.5 其他常见约束类型

当然，还有很多其他类型的约束。以下仅举几个例子：

- 平面。物体被限制在二维平面内移动。
- 车轮。这通常是一个具有无限旋转的铰链约束，并与通过弹簧阻尼器组件模拟的某种形式的垂直悬架相结合。
- 滑轮。在这个特殊的约束中，一根假想的绳子穿过滑轮，并连接到两个物体上。物体通过杠杆比沿着绳子移动。

约束可以是可打破的，这意味着一旦施加足够的力，它们就会自动分解。或者，游戏可以随意打开或关闭约束，并使用其自身的标准来判断何时应该打破约束。

13.4.8.6 约束链

由于约束求解器的迭代特性，长链链接体有时难以稳定地模拟。约束链是一组特殊的约束，它包含告知约束求解器对象如何连接的信息。这使得求解器能够以比其他方式更稳定的方式处理约束链。

图 13.32.有限的铰链约束模仿了门的行为。

13.4.8.4 Prismatic Constraints

Prismatic constraints act like a piston: A constrained body's motion is restricted to a single translational degree of freedom. A prismatic constraint may or may not permit rotation about the translation axis of the piston. Prismatic constraints can of course be limited or unlimited and may or may not include friction. A prismatic constraint is illustrated in Figure 13.33.

13.4.8.5 Other Common Constraint Types

Many other types of constraints are possible, of course. Here are just a few examples:

- *Planar*. Objects are constrained to move in a two-dimensional plane.
- *Wheel*. This is typically a hinge constraint with unlimited rotation, coupled with some form of vertical suspension simulated via a spring-damper assembly.
- *Pulley*. In this specialized constraint, an imaginary rope passes through a pulley and is attached to two bodies. The bodies move along the line of the rope via a leverage ratio.

Constraints may be breakable, meaning that after enough force is applied, they automatically come apart. Alternatively, the game can turn the constraint on and off at will, using its own criteria for when the constraint should break.

13.4.8.6 Constraint Chains

Long chains of linked bodies are sometimes difficult to simulate in a stable manner because of the iterative nature of the constraint solver. A *constraint chain* is a specialized group of constraints with information that tells the constraint solver how the objects are connected. This allows the solver to deal with the chain in a more stable manner than would otherwise be possible.

Figure 13.32. A limited hinge constraint mimics the behavior of a door.

图 13.33。棱柱约束的作用类似于活塞。

13.4.8.7 布娃娃

布娃娃是对人体在死亡或失去意识，即完全瘫痪时可能活动方式的物理模拟。布娃娃是通过将一系列刚体连接在一起而创建的，每个刚体对应身体的每个半刚性部位。例如，我们可能会为脚、小腿、大腿、手、上臂、下臂和头部设置胶囊体，也可能为躯干设置一些胶囊体，以模拟脊柱的灵活性。

布娃娃中的刚体通过约束相互连接。布娃娃约束专门用于模拟真实人体关节所能进行的各种运动。我们通常使用约束链来提高模拟的稳定性。

布娃娃模拟始终与动画系统紧密集成。当布娃娃在物理世界中移动时，我们会提取刚体的位置和旋转，并利用这些信息来驱动动画骨架中某些关节的位置和方向。因此，实际上，布娃娃只是一种由物理系统驱动的程序化动画。（有关骨骼动画的更多详细信息，请参阅第 12 章。）当然，实现布娃娃并不像我在这里说的那么简单。首先，布娃娃中的刚体与动画骨架中的关节之间通常不存在一一对应的关系——骨架的关节通常比布娃娃的刚体多。因此，我们需要一个能够将刚体映射到关节的系统（即，一个“知道”布娃娃中每个刚体对应哪个关节的系统）。在布娃娃身体驱动的关节之间可能存在额外的关节，因此映射系统也必须能够确定这些中间关节的正确姿势变换。这并非一门精确的科学。我们必须运用艺术判断和/或一些人体生物力学知识，才能创造出看起来自然的布娃娃。

Figure 13.33. A prismatic constraint acts like a piston.

13.4.8.7 Rag Dolls

A rag doll is a physical simulation of the way a human body might move when it is dead or unconscious and hence entirely limp. Rag dolls are created by linking together a collection of rigid bodies, one for each semi-rigid part of the body. For example, we might have capsules for the feet, calves, thighs, hands, upper and lower arms and head and possibly a few for the torso to simulate the flexibility of the spine.

The rigid bodies in a rag doll are connected to one another via constraints. Rag doll constraints are specialized to mimic the kinds of motions the joints in a real human body can perform. We usually make use of constraint chains to improve the stability of the simulation.

A rag doll simulation is always tightly integrated with the animation system. As the rag doll moves in the physics world, we extract the positions and rotations of the rigid bodies, and we use this information to drive the positions and orientations of certain joints in the animated skeleton. So in effect, a rag doll is really just a form of *procedural animation* that happens to be driven by the physics system. (See Chapter 12 for more details on skeletal animation.)

Of course, implementing a rag doll is not quite as simple as I've made it sound here. For one thing, there's usually not a one-to-one mapping between the rigid bodies in the rag doll and the joints in the animated skeleton—the skeleton usually has more joints than the rag doll has bodies. Therefore, we need a system that can map rigid bodies to joints (i.e., one that "knows" to which joint each rigid body in the rag doll corresponds). There may be additional joints between those that are being driven by the rag doll bodies, so the mapping system must also be capable of determining the correct pose transforms for these intervening joints. This is not an exact science. We must apply artistic judgment and/or some knowledge of human biomechanics in order to achieve a natural-looking rag doll.

图 13.34。在无任何附加力或扭矩的情况下，使用动力布娃娃约束可以使代表下臂的刚体精确跟踪动画肘关节的运动（左图）。如果障碍物阻挡了刚体的运动，它将以逼真的方式偏离动画肘关节的运动（右图）。

13.4.8.8 动力约束

约束也可以是“动力化的”，这意味着动画系统等外部引擎系统可以间接控制布娃娃中刚体的平移和方向。

我们以肘关节为例。肘关节的作用很像一个受限的铰链，自由旋转角度略小于 180 度。（实际上，肘关节也可以轴向旋转，但为了讨论的目的，我们将忽略这一点。）为了满足这一约束，我们将肘关节建模为旋转弹簧。这种弹簧施加的扭矩与弹簧偏离某个预定义静止角的偏转角度成比例，$N = -k(\theta - \theta_{rest})$。现在想象一下从外部改变静止角，比如确保它始终与动画骨架中肘关节的角度相匹配。随着静止角的变化，弹簧将失去平衡，并将施加一个扭矩，该扭矩倾向于将肘关节旋转回与θ_{rest}对齐。在没有任何其他力或扭矩的情况下，刚体将精确跟踪动画骨架中肘关节的运动。但如果引入其他力（例如，小臂与不可移动的物体接触），这些力就会影响肘关节的整体运动，使其在某种程度上偏离动画运动。如图 13.34 所示，这会产生一种错觉，让人感觉自己正尽力以某种方式移动（即动画提供的“理想”运动），但由于物理世界的限制，有时无法做到这一点（例如，当她试图向前摆动手臂时，手臂会被什么东西卡住）。

Figure 13.34. With a powered rag doll constraint, and in the absence of any additional forces or torques, a rigid body representing the lower arm can be made to exactly track the movements of an animated elbow joint (left). If an obstacle blocks the motion of the body, it will diverge from that of the animated elbow joint in a realistic way (right).

13.4.8.8 Powered Constraints

Constraints can also be “powered,” meaning that an external engine system such as the animation system can indirectly control the translations and orientations of the rigid bodies in the rag doll.

Let’s take an elbow joint as an example. An elbow acts pretty much like a limited hinge, with a little less than 180 degrees of free rotation. (Actually, an elbow can also rotate axially, but we’ll ignore that for the purposes of this discussion.) To power this constraint, we model the elbow as a *rotational spring*. Such a spring exerts a torque proportional to the spring’s angle of deflection away from some predefined rest angle, $N = -k(\theta - \theta_{rest})$. Now imagine changing the rest angle externally, say by ensuring that it always matches the angle of the elbow joint in an animated skeleton. As the rest angle changes, the spring will find itself out of equilibrium, and it will exert a torque that tends to rotate the elbow back into alignment with θ_{rest}. In the absence of any other forces or torques, the rigid bodies will exactly track the motion of the elbow joint in the animated skeleton. But if other forces are introduced (for example, the lower arm comes in contact with an immovable object), then these forces will play into the overall motion of the elbow joint, allowing it to diverge from the animated motion in a somewhat realistic manner. As illustrated in Figure 13.34, this provides the illusion of a human who is trying her best to move in a certain way (i.e., the “ideal” motion provided by the animation) but who is sometimes unable to do so due to the limitations of the physical world (e.g., her arm gets caught on something as she tries to swing it forward).

13.4.9 控制刚体的运动

大多数游戏设计都要求对刚体的运动方式进行一定程度的控制，而不仅仅是它们在重力作用下以及与场景中其他物体碰撞时自然移动的方式。例如：

- 通风口对进入其影响轴的任何物体施加向上的力。
- 一辆汽车与一辆拖车相连，并在行驶过程中对拖车施加拉力。
- 牵引光束对不知情的航天器施加力。
- 反重力装置使物体悬浮。
- 河流的流动会产生力场，导致河中漂浮的物体顺流移动。

诸如此类，不胜枚举。大多数物理引擎通常为用户提供多种控制模拟中物体的方式。我们将在以下章节中概述其中最常见的机制。

13.4.9.1 重力

在大多数以地球表面或其他行星（或模拟重力的航天器）为背景的游戏中，重力无处不在。严格来说，重力并非力，而是一种（大致）恒定的加速度，因此它会对所有物体产生同等的影响，无论其质量如何。由于重力加速度的普遍性和特殊性，大多数 SDK 都会通过全局设置来指定其大小和方向。（如果您正在编写一款太空游戏，您可以随时将重力设置为零，以将其从模拟中消除。）

13.4.9.2 施加力量

在游戏物理模拟中，可以对物体施加任意数量的力。力的作用时间总是有限的。（如果力是瞬间作用的，则称为冲量——更多内容请参见下文第 13.4.9.4 节。）游戏中的力通常本质上是动态的——它们的方向和/或大小通常会在每一帧发生变化。因此，大多数物理 SDK 中的力施加函数被设计为在力的作用期间每帧调用一次。此类函数的签名通常类似于：applyForce(const Vector& forceInNewtons)，其中力的持续时间假定为 Δ t。

13.4.9 Controlling the Motions of Rigid Bodies

Most game designs call for a degree of control over the way rigid bodies move over and above the way they would move naturally under the influence of gravity and in response to collisions with other objects in the scene. For example:

- An air vent applies an upward force to any object that enters its shaft of influence.
- A car is coupled to a trailer and exerts a pulling force on it as it moves.
- A tractor beam exerts a force on an unwitting spacecraft.
- An anti-gravity device causes objects to hover.
- The flow of a river creates a force field that causes objects floating in the river to move downstream.

And the list goes on. Most physics engines typically provide their users with a number of ways to exert control over the bodies in the simulation. We'll outline the most common of these mechanisms in the following sections.

13.4.9.1 Gravity

Gravity is ubiquitous in most games that take place on the surface of the Earth or some other planet (or on a spacecraft with simulated gravity). Gravity is technically not a force but rather a (roughly) constant acceleration, so it affects all bodies equally regardless of their mass. Because of its ubiquitous and special nature, the magnitude and direction of the gravitational acceleration is specified via a global setting in most SDKs. (If you're writing a space game, you can always set gravity to zero to eliminate it from the simulation.)

13.4.9.2 Applying Forces

Any number of forces can be applied to the bodies in a game physics simulation. A force always acts over a finite time interval. (If it acted instantaneously, it would be called an *impulse*—more on that in Section 13.4.9.4 below.) The forces in a game are often dynamic in nature—they often change their directions and/or their magnitudes every frame. So the force-application function in most physics SDKs is designed to be called once per frame for the duration of the force's influence. The signature of such a function usually looks something like this: `applyForce(const Vector& forceInNewtons)`, where the duration of the force is assumed to be Δt.

13.4.9.3 施加扭矩

当施加力时，其作用线穿过物体的质心，则不会产生扭矩，只会影响物体的线性加速度。如果施加的力偏离质心，则会同时产生线性加速度和旋转加速度。也可以通过在距质心等距的点施加两个大小相等、方向相反的力来对物体施加纯扭矩。这样一对力引起的线性运动将相互抵消（因为就线性动力学而言，这两个力都作用于质心）。这样就只剩下它们的旋转效应了。像这样的一对产生扭矩的力被称为力偶 (http://en.wikipedia.org/wiki/Couple_(mechanics)。为此目的，可以提供诸如 applyTorque(const Vector& torque) 之类的特殊函数。但是，如果您的物理 SDK 未提供 applyTorque() 函数，您可以编写一个并让它生成合适的力偶。

13.4.9.4 施加脉冲

正如我们在 13.4.7.2 节中看到的，冲量是速度的瞬时变化（或者实际上是动量的变化）。从技术角度来说，冲量是一种作用时间极短的力。然而，在时间步长动力学模拟中，力的最短持续时间为 Δt，这不足以充分模拟冲量。因此，大多数物理 SDK 都提供了一个带有签名的函数，例如 applyImpulse(const Vector& impulse)，用于向物体施加冲量。当然，冲量有两种类型——线性和角度——一个好的 SDK 应该提供同时应用这两种类型的函数。

13.4.10 碰撞/物理步骤

现在我们已经介绍了实现碰撞和物理系统背后的理论和一些技术细节，让我们简单看一下这些系统实际上是如何在每一帧执行更新的。

每个碰撞/物理引擎在其更新步骤中都会执行以下基本任务。不同的物理 SDK 可能以不同的顺序执行这些阶段。不过，我见过最常用的技术大致如下：

1. 将物理世界中作用于物体的力和扭矩向前积分Δt，以确定它们在下一帧的暂定位置和方向。

13.4.9.3 Applying Torques

When a force is applied such that its line of action passes through the center of mass of a body, no torque is generated, and only the body's linear acceleration is affected. If it is applied off-center, it will induce *both* a linear and a rotational acceleration. A *pure torque* can be applied to a body as well by applying two equal and opposite forces to points equidistant from the center of mass. The linear motions induced by such a pair of forces will cancel each other out (since for the purposes of linear dynamics, the forces both act at the center of mass). This leaves only their rotational effects. A pair of torque-inducing forces like this is known as a *couple* (http://en.wikipedia.org/wiki/Couple_(mechanics). A special function such as `applyTorque(const Vector& torque)` may be provided for this purpose. However, if your physics SDK provides no `applyTorque()` function, you can always write one and have it generate a suitable couple instead.

13.4.9.4 Applying Impulses

As we saw in Section 13.4.7.2, an *impulse* is an instantaneous change in velocity (or actually, a change in momentum). Technically speaking, an impulse is a force that acts for an infinitesimal amount of time. However, the shortest possible duration of force application in a time-stepped dynamics simulation is Δt, which is not short enough to simulate an impulse adequately. As such, most physics SDKs provide a function with a signature such as `applyImpulse(const Vector& impulse)` for the purposes of applying impulses to bodies. Of course, impulses come in two flavors—linear and angular—and a good SDK should provide functions for applying both types.

13.4.10 The Collision/Physics Step

Now that we've covered the theory and some of the technical details behind implementing a collision and physics system, let's take a brief look at how these systems actually perform their updates every frame.

Every collision/physics engine performs the following basic tasks during its update step. Different physics SDKs may perform these phases in different orders. That said, the technique I've seen used most often goes something like this:

1. The forces and torques acting on the bodies in the physics world are integrated forward by Δt in order to determine their tentative positions and orientations next frame.

2. 调用碰撞检测库来确定物体之间是否由于试探性移动而产生了新的接触。（为了利用时间一致性，物体通常会跟踪它们的接触。因此，在模拟的每个步骤中，碰撞引擎只需确定先前的接触是否丢失以及是否添加了新的接触。）

3. 碰撞的解决通常通过施加冲量或惩罚力，或作为后续约束求解步骤的一部分进行。根据 SDK 的不同，此阶段可能包含或不包含连续碰撞检测（CCD，也称为撞击时间检测或 TOI）。

4. 约束求解器满足约束条件。

在步骤 4 结束时，一些物体可能已经偏离了在步骤 1 中确定的暂定位置。这种移动可能会导致物体之间发生更多穿透，或导致其他先前满足的约束被破坏。因此，步骤 1 到 4（有时仅执行步骤 2 到 4，具体取决于碰撞和约束的解析方式）会重复执行，直到 (a) 所有碰撞都已成功解析且所有约束都已满足，或 (b) 超过预定义的最大迭代次数。在后一种情况下，求解器实际上会“放弃”，希望在模拟的后续帧中问题能够自然解决。这有助于通过在多个帧上分摊碰撞和约束解析的成本来避免性能峰值。但是，如果误差过大、时间步长过长或不一致，则可能导致行为看起来不正确。可以将惩罚力融入模拟中，以便随着时间的推移逐步解决这些问题。

13.4.10.1 约束求解器

约束求解器本质上是一种迭代算法，它试图通过最小化物理世界中物体的实际位置和旋转与约束定义的理想位置和旋转之间的误差，同时满足大量约束条件。因此，约束求解器本质上是一种迭代误差最小化算法。

让我们首先看一下约束求解器在一对物体通过单个铰链约束连接的简单情况下是如何工作的。在物理模拟的每个步骤中，数值积分器都会为这两个物体找到新的试探性变换。然后，约束求解器评估它们的

2. The collision detection library is called to determine if any new contacts have been generated between any of the objects as a result of their tentative movement. (The bodies normally keep track of their contacts in order to take advantage of temporal coherency. Hence, at each step of the simulation, the collision engine need only determine whether any previous contacts have been lost and whether any new contacts have been added.)

3. Collisions are resolved, often by applying impulses or penalty forces or as part of the constraint-solving step below. Depending on the SDK, this phase may or may not include continuous collision detection (CCD, otherwise known as time of impact detection or TOI).

4. Constraints are satisfied by the constraint solver.

At the conclusion of step 4, some of the bodies may have moved away from their tentative positions as determined in step 1. This movement may cause additional interpenetrations between objects or cause other previously satisfied constraints to be broken. Therefore, steps 1 through 4 (or sometimes only 2 through 4, depending on how collisions and constraints are resolved) are repeated until either (a) all collisions have been successfully resolved and all constraints are satisfied, or (b) a predefined maximum number of iterations has been exceeded. In the latter case, the solver effectively “gives up,” with the hope that things will resolve themselves naturally during subsequent frames of the simulation. This helps to avoid performance spikes by amortizing the cost of collision and constraint resolution over multiple frames. However, it can lead to incorrect-looking behavior if the errors are too large or if the time step is too long or is inconsistent. Penalty forces can be blended into the simulation in order to gradually resolve these problems over time.

13.4.10.1 The Constraint Solver

A constraint solver is essentially an iterative algorithm that attempts to satisfy a large number of constraints simultaneously by *minimizing the error* between the actual positions and rotations of the bodies in the physics world and their ideal positions and rotations as defined by the constraints. As such, constraint solvers are essentially iterative error-minimization algorithms.

Let’s take a look first at how a constraint solver works in the trivial case of a single pair of bodies connected by a single hinge constraint. During each step of the physics simulation, the numerical integrator will find new tentative transforms for the two bodies. The constraint solver then evaluates their

相对位置，并计算它们共享旋转轴的位置和方向之间的误差。如果检测到任何误差，求解器会移动物体，以最小化或消除误差。由于系统中没有其他物体，该步骤的第二次迭代应该不会发现新的接触，约束求解器将发现单铰链约束现在已得到满足。因此，循环可以退出，无需进一步迭代。

当必须同时满足多个约束时，可能需要更多次迭代。在每次迭代中，数值积分器有时会使物体与其约束不对齐，而约束求解器则会使其重新对齐。如果运气好，并且采用精心设计的方法来最小化约束求解器中的错误，这个反馈循环最终应该会稳定下来，得到一个有效的解。然而，解可能并不总是精确的。这就是为什么在使用物理引擎的游戏中，你有时会看到看似不可能的行为，比如拉伸的链条（在链环之间打开小缝隙）、短暂穿透的物体或瞬间超出其允许范围的铰链。约束求解器的目标是最小化错误——它并不总是能够完全消除错误。

13.4.10.2 发动机之间的差异

当然，上述描述过于简化了物理/碰撞引擎每一帧的实际运作方式。不同物理 SDK 中各个计算阶段的执行方式及其相对顺序可能有所不同。例如，某些类型的约束被建模为力和扭矩，这些力和扭矩由数值积分步骤处理，而不是由约束求解器求解。碰撞可以在积分步骤之前运行，而不是之后。碰撞可以通过多种不同的方式解决。我们的目的仅仅是让您大致了解这些系统的工作原理。要详细了解任何一个 SDK 的运作方式，您需要阅读其文档，并且可能还需要查看其源代码（假设相关代码可供您阅读）。好奇且勤奋的读者可以通过下载并试用 Open Dynamics Engine (ODE) 和/或 PhysX 来获得良好的开端，因为这两个 SDK 都是免费的。您还可以从 ODE 的 wiki 中学到很多知识，网址为 http://opende.sourceforge.net/wiki/index.php/Main_Page。

relative positions and calculates the error between the positions and orientations of their shared axis of rotation. If any error is detected, the solver moves the bodies in such a way as to minimize or eliminate it. Since there are no other bodies in the system, the second iteration of the step should discover no new contacts, and the constraint solver will find that the one hinge constraint is now satisfied. Hence the loop can exit without further iterations.

When more than one constraint must be satisfied simultaneously, more iterations may be required. During each iteration, the numerical integrator will sometimes tend to move the bodies out of alignment with their constraints, while the constraint solver tends to put them back into alignment. With luck, and a carefully designed approach to minimizing error in the constraint solver, this feedback loop should eventually settle into a valid solution. However, the solution may not always be exact. This is why, in games with physics engines, you sometimes witness seemingly impossible behaviors, like chains that stretch (opening up little gaps between the links), objects that interpenetrate briefly or hinges that momentarily move beyond their allowable ranges. The goal of the constraint solver is to minimize error—it's not always possible to eliminate it completely.

13.4.10.2 Variations between Engines

The description given above is of course an over-simplification of what really goes on in a physics/collision engine every frame. The way in which the various phases of computation are performed, and their order relative to one another, may vary from physics SDK to physics SDK. For example, some kinds of constraints are modeled as forces and torques that are taken care of by the numerical integration step rather than being resolved by the constraint solver. Collision may be run before the integration step rather than after. Collisions may be resolved in any number of different ways. Our goal here is merely to give you a taste of how these systems work. For a detailed understanding of how any one SDK operates, you'll want to read its documentation and probably also inspect its source code (presuming the relevant bits are available for you to read). The curious and industrious reader can get a good start by downloading and experimenting with Open Dynamics Engine (ODE) and/or PhysX, as these two SDKs are available for free. You can also learn a great deal from ODE's wiki, which is available at http://opende.sourceforge.net/wiki/index.php/Main_Page.

图 13.35。刚体通过游戏对象与其视觉表示相链接。通常会提供可选的直接渲染路径，以便可视化刚体的位置，从而方便调试。

13.5 将物理引擎集成到您的游戏中

显然，碰撞/物理引擎本身用处不大——它必须集成到你的游戏引擎中。在本节中，我们将讨论碰撞/物理引擎与游戏代码其余部分之间最常见的接口点。

13.5.1 链接游戏对象和刚体

碰撞/物理世界中的刚体和可碰撞体只不过是抽象的数学描述。为了使它们在游戏环境中发挥作用，我们需要以某种方式将它们与屏幕上的视觉呈现联系起来。通常，我们不会直接绘制刚体（除非出于调试目的）。相反，刚体用于描述构成虚拟游戏世界的逻辑对象的形状、大小和物理行为。我们将在第16章深入讨论游戏对象，但目前，我们将依赖于我们对游戏对象的直观概念——游戏世界中的逻辑实体，例如角色、车辆、武器、浮动道具等等。因此，物理世界中的刚体与其屏幕上的视觉呈现之间的联系通常是间接的，逻辑游戏对象充当连接两者的枢纽。如图13.35所示。

一般来说，游戏对象在碰撞/物理世界中由零个或多个刚体表示。以下列表描述了三种可能的情况：

- 零刚体。游戏对象在物理上没有任何刚体

Figure 13.35. Rigid bodies are linked to their visual representations by way of game objects. An optional direct rendering path is usually provided so that the locations of the rigid bodies can be visualized for debugging purposes.

13.5 Integrating a Physics Engine into Your Game

Obviously, a collision/physics engine is of little use by itself—it must be integrated into your game engine. In this section, we'll discuss the most common interface points between the collision/physics engine and the rest of the game code.

13.5.1 Linking Game Objects and Rigid Bodies

The rigid bodies and collidables in the collision/physics world are nothing more than abstract mathematical descriptions. In order for them to be useful in the context of a game, we need to link them in some way to their visual representations on-screen. Usually, we don't draw the rigid bodies directly (except for debugging purposes). Instead, the rigid bodies are used to describe the shape, size, and physical behavior of the logical objects that make up the virtual game world. We'll discuss game objects in depth in Chapter 16, but for the time being, we'll rely on our intuitive notion of what a game object is—a logical entity in the game world, such as a character, a vehicle, a weapon, a floating power-up and so on. So the linkage between a rigid body in the physics world and its visual representation on-screen is usually indirect, with the logical game object serving as the hub that links the two together. This is illustrated in Figure 13.35.

In general, a game object is represented in the collision/physics world by zero or more rigid bodies. The following list describes three possible scenarios:

- *Zero rigid bodies*. Game objects without any rigid bodies in the physics

游戏世界中的物体看起来就像不是实体，因为它们根本没有碰撞表示。玩家或非玩家角色无法与之互动的装饰性物体，例如飞过头顶的鸟儿，或游戏世界中那些看得见却永远无法触及的部分，可能没有碰撞。这种情况也适用于由于某些原因需要手动处理碰撞检测（没有碰撞/物理引擎的帮助）的物体。

• 一个刚体。大多数简单的游戏对象只需用一个刚体来表示。在这种情况下，刚体的可碰撞体形状的选择应与游戏对象视觉呈现的形状非常接近，并且刚体的位置和方向也与游戏对象本身的位置和方向完全匹配。

• 多个刚体。一些复杂的游戏对象在碰撞/物理世界中由多个刚体表示。例如，角色、机械、车辆或任何由多个可相对运动的实体部件组成的对象。这类游戏对象通常使用骨架（即仿射变换的层次结构）来追踪其组成部分的位置（当然也可以使用其他方法）。刚体通常与骨架的关节相连，使得每个刚体的位置和方向与其中一个关节的位置和方向相对应。骨架中的关节可能由动画驱动，在这种情况下，关联的刚体只是随波逐流。或者，物理系统可以驱动刚体的位置，从而间接控制关节的位置。关节到刚体的映射可能是一对一的，也可能不是——有些关节可能完全由动画控制，而另一些关节则链接到刚体。

当然，游戏对象和刚体之间的连接必须由引擎管理。通常，每个游戏对象都会管理自己的刚体，在必要时创建和销毁它们，根据需要将它们添加到物理世界或从物理世界中移除，并维护每个刚体的位置与游戏对象和/或其某个关节的位置之间的连接。对于由多个刚体组成的复杂游戏对象，可以使用某种包装类来管理它们。这使得游戏对象无需处理管理一组刚体的具体细节，并允许不同类型的游戏对象以一致的方式管理其刚体。

world act as though they are not solid, because they have no collision representation at all. Decorative objects with which the player or non-player characters cannot interact, such as birds flying overhead or portions of the game world that can be seen but never reached, might have no collision. This scenario can also apply to objects whose collision detection is handled manually (without the help of the collision/physics engine) for some reason.

- *One rigid body*. Most simple game objects need only be represented by a single rigid body. In this case, the shape of the rigid body's collidable is chosen to closely approximate the shape of the game object's visual representation, and the rigid body's position and orientation exactly match the position and orientation of the game object itself.
- *Multiple rigid bodies*. Some complex game objects are represented by multiple rigid bodies in the collision/physics world. Examples include characters, machinery, vehicles or any object that is composed of multiple solid pieces that can move relative to one another. Such game objects usually make use of a skeleton (i.e., a hierarchy of affine transforms) to track the locations of their component pieces (although other means are certainly possible as well). The rigid bodies are usually linked to the joints of the skeleton in such a way that the position and orientation of each rigid body corresponds to the position and orientation of one of the joints. The joints in the skeleton might be driven by an animation, in which case the associated rigid bodies simply come along for the ride. Alternatively, the physics system might drive the locations of rigid bodies and hence indirectly control the locations of the joints. The mapping from joints to rigid bodies may or may not be one-to-one—some joints might be controlled entirely by animation, while others are linked to rigid bodies.

The linkage between game objects and rigid bodies must be managed by the engine, of course. Typically, each game object will manage its own rigid bodies, creating and destroying them when necessary, adding and removing them from the physics world as needed, and maintaining the connection between each rigid body's location and the location of the game object and/or one of its joints. For complex game objects consisting of multiple rigid bodies, a wrapper class of some kind may be used to manage them. This insulates the game objects from the nitty-gritty details of managing a collection of rigid bodies and allows different kinds of game objects to manage their rigid bodies in a consistent way.

13.5.1.1 物理驱动的物体

如果我们的游戏有刚体动力学系统，那么我们大概希望游戏中至少部分物体的运动完全由模拟驱动。这类游戏对象被称为物理驱动对象。碎片、爆炸的建筑物、从山坡上滚落的岩石、空弹匣和弹壳——这些都是物理驱动对象的例子。

通过逐步模拟，物理驱动的刚体会与其游戏对象关联，然后向物理系统查询刚体的位置和方向。之后，该变换会应用于整个游戏对象、游戏对象内的关节或其他数据结构。

示例：建造一个带可拆卸门的保险箱

当物理驱动的刚体与骨架的关节连接时，刚体通常会受到约束，以产生所需的运动。举个例子，让我们看看如何建模一个带有可拆卸门的保险箱。

直观地讲，我们假设保险箱由一个三角形网格和两个子网格组成，一个用于外壳，一个用于门。我们使用一个双关节骨架来控制这两个部件的运动。根关节绑定到保险箱外壳，子关节绑定到门，这样旋转门关节时，门子网格就会以合适的方式摆动打开和关闭。

保险箱的碰撞几何体也被分解成两个独立的部分，一个代表外壳，一个代表门。这两个部分用于在碰撞/物理世界中创建两个完全独立的刚体。保险箱外壳的刚体连接到骨架中的根关节，门的刚体连接到门关节。然后，在物理世界中添加一个铰链约束，以确保在模拟两个刚体的动力学时，门体相对于外壳正确摆动。代表外壳和门的两个刚体的运动用于更新骨架中两个关节的变换。一旦动画系统生成了骨架的矩阵调色板，渲染引擎就会在物理世界中刚体的位置绘制外壳和门的子网格。

如果在某个时刻需要炸开门，可以打破约束，并向刚体施加冲量，使其飞出。在人类玩家看来，门和房屋似乎是独立的物体。但实际上，它仍然是一个游戏对象，一个带有两个关节和两个刚体的三角形网格。

13.5.1.1 Physics-Driven Bodies

If our game has a rigid body dynamics system, then presumably we want the motions of at least some of the objects in the game to be driven entirely by the simulation. Such game objects are called *physics-driven* objects. Bits of debris, exploding buildings, rocks rolling down a hillside, empty magazines and shell casings—these are all examples of physics-driven objects.

A physics-driven rigid body is linked to its game object by stepping the simulation and then querying the physics system for the body's position and orientation. This transform is then applied either to the game object as a whole or to a joint or some other data structure within the game object.

Example: Building a Safe with a Detachable Door

When physics-driven rigid bodies are linked to the joints of a skeleton, the bodies are often *constrained* to produce a desired kind of motion. As an example, let's look at how a safe with a detachable door might be modeled.

Visually, let's assume that the safe consists of a single triangle mesh with two submeshes, one for the housing and one for the door. A two-joint skeleton is used to control the motions of these two pieces. The root joint is bound to the housing of the safe, while the child joint is bound to the door in such a way that rotating the door joint causes the door submesh to swing open and shut in a suitable way.

The collision geometry for the safe is broken into two independent pieces as well, one for the housing and one for the door. These two pieces are used to create two totally separate rigid bodies in the collision/physics world. The rigid body for the safe's housing is attached to the root joint in the skeleton, and the door's rigid body is linked to the door joint. A hinge constraint is then added to the physics world to ensure that the door body swings properly relative to the housing when the dynamics of the two rigid bodies are simulated. The motions of the two rigid bodies representing the housing and the door are used to update the transforms of the two joints in the skeleton. Once the skeleton's matrix palette has been generated by the animation system, the rendering engine will end up drawing the housing and door submeshes in the locations of the rigid bodies within the physics world.

If the door needs to be blown off at some point, the constraint can be broken, and impulses can be applied to the rigid bodies to send them flying. Visibly, it will appear to the human player that the door and the housing have become separate objects. But in reality, it's still a single game object and a single triangle mesh with two joints and two rigid bodies.

13.5.1.2 游戏驱动的实体

在大多数游戏中，游戏世界中的某些物体需要以非物理的方式移动。这些物体的运动可能由动画或样条曲线路径决定，也可能由人类玩家控制。我们通常希望这些物体参与碰撞检测（例如，能够将物理驱动的物体推开），但我们不希望物理系统以任何方式干扰它们的运动。为了适应这类物体，大多数物理 SDK 提供了一种特殊类型的刚体，称为游戏驱动刚体。（Havok 称之为“关键帧”刚体。）

游戏驱动的物体不受重力影响。物理系统也认为它们具有无限大的质量（通常用零质量表示，因为这对于物理驱动的物体来说是无效的质量）。无限大质量的假设确保了模拟中的力和碰撞脉冲永远不会改变游戏驱动物体的速度。

为了在物理世界中移动游戏驱动的物体，我们不能简单地将其位置和方向设置为每帧都与相应游戏对象的位置一致。这样做会引入不连续性，而物理模拟将很难解决这些不连续性问题。（例如，一个物理驱动的物体可能会突然穿透一个游戏驱动的物体，但它没有关于游戏驱动物体动量的信息来解析碰撞。）因此，游戏驱动的物体通常使用冲量来移动——速度的瞬时变化，当将其积分到时间步长时，会在时间步长结束时将物体定位到所需的位置。大多数物理 SDK 都提供了一个便捷函数，可以计算在下一帧达到所需位置和方向所需的线性和角度冲量。移动游戏驱动的物体时，我们必须小心，在它应该停止时将其速度归零。否则，物体将永远沿着其上一个非零轨迹继续运动。

示例：动画保险箱门

让我们继续之前那个带有可拆卸门的保险箱的例子。想象一下，我们希望一个角色走到保险箱前，拨动密码，打开门，存入一些钱，然后关上并锁上门。之后，我们希望另一个角色以一种不太文明的方式——炸开保险箱门——来获取钱。为此，保险箱的建模需要添加一个用于转盘的子网格和一个允许转盘旋转的关节。然而，转盘不需要刚体，除非

13.5.1.2 Game-Driven Bodies

In most games, certain objects in the game world need to be moved about in a non-physical way. The motions of such objects might be determined by an animation or by following a spline path, or they might be under the control of the human player. We often want these objects to participate in collision detection—to be capable of pushing the physics-driven objects out of their way, for example—but we do not want the physics system to interfere with their motion in any way. To accommodate such objects, most physics SDKs provide a special type of rigid body known as a *game-driven body*. (Havok calls these “key framed” bodies.)

Game-driven bodies do not experience the effects of gravity. They are also considered to be infinitely massive by the physics system (usually denoted by a mass of zero, since this is an invalid mass for a physics-driven body). The assumption of infinite mass ensures that forces and collision impulses within the simulation can never change the velocity of a game-driven body.

To move a game-driven body around in the physics world, we cannot simply set its position and orientation every frame to match the location of the corresponding game object. Doing so would introduce discontinuities that would be very difficult for the physical simulation to resolve. (For example, a physics-driven body might find itself suddenly interpenetrating a game-driven body, but it would have no information about the game-driven body’s momentum with which to resolve the collision.) As such, game-driven bodies are usually moved using impulses—instantaneous changes in velocity that, when integrated forward in time, will position the bodies in the desired places at the end of the time step. Most physics SDKs provide a convenience function that will calculate the linear and angular impulses required in order to achieve a desired position and orientation on the next frame. When moving a game-driven body, we do have to be careful to zero out its velocity when it is supposed to stop. Otherwise, the body will continue forever along its last nonzero trajectory.

Example: Animated Safe Door

Let’s continue our example of the safe with a detachable door. Imagine that we want a character to walk up to the safe, dial the combination, open the door, deposit some money and close and lock the door again. Later, we want a different character to get the money in a rather less-civilized manner—by blowing the door off the safe. To do this, the safe would be modeled with an additional submesh for the dial and an additional joint that allows the dial to be rotated. No rigid body is required for the dial, however, unless of course

我们希望它在门爆炸时飞走。

在人物打开和关闭保险箱的动画序列中，可以将刚体设置为游戏驱动模式。动画现在驱动关节，关节又驱动刚体。之后，当保险箱门被炸开时，我们可以将刚体切换到物理驱动模式，打破铰链约束，施加冲量，然后观察保险箱门飞出去的情景。

你可能已经注意到了，在这个例子中，铰链约束其实并不需要。只有当门在某个时刻保持打开状态，并且我们希望看到门在保险箱被移动或门被撞击时自然摆动时，才需要铰链约束。

13.5.1.3 固定体

大多数游戏世界由静态几何体和动态物体组成。为了模拟游戏世界的静态组件，大多数物理 SDK 都提供了一种特殊的刚体，称为固定刚体。固定刚体的行为有点像游戏驱动刚体，但它们根本不参与动力学模拟。实际上，它们只是碰撞刚体。这种优化可以显著提升大多数游戏的性能，尤其是那些在大型静态□□世界中仅包含少量动态物体的游戏。

13.5.1.4 Havok 的运动类型

在 Havok 中，所有类型的刚体都由类 hkp RigidBody 的实例表示。每个实例都包含一个指定其运动类型的字段。运动类型告诉系统刚体是固定的、游戏驱动的（Havok 称之为“关键帧”）还是物理驱动的（Havok 称之为“动态的”）。如果使用固定运动类型创建刚体，则其类型永远不能改变。否则，可以在运行时动态更改刚体的运动类型。此功能非常有用。例如，角色手中的物体是游戏驱动的。但是，一旦角色掉落或扔出该物体，它就会变为物理驱动，以便动力学模拟可以接管其运动。在 Havok 中，只需在释放时更改运动类型即可轻松实现这一点。

运动类型还可以为 Havok 提供一些关于动态物体惯性张量的提示。因此，“动态”运动类型被细分为“球体惯性动态”、“箱体惯性动态”等等子类别。根据物体的运动类型，Havok 可以根据对惯性张量内部结构的假设来决定应用各种优化。

we want it to fly off when the door explodes.

During the animated sequence of the person opening and closing the safe, its rigid bodies can be put into game-driven mode. The animation now drives the joints, which in turn drive the rigid bodies. Later, when the door is to be blown off, we can switch the rigid bodies into physics-driven mode, break the hinge constraint, apply the impulse and watch the door fly.

As you've probably already noticed, the hinge constraint is not actually needed in this particular example. It would only be required if the door is to be left open at some point and we want to see the door swinging naturally in response to the safe being moved or the door being bumped.

13.5.1.3 Fixed Bodies

Most game worlds are composed of both static geometry and dynamic objects. To model the static components of the game world, most physics SDKs provide a special kind of rigid body known as a *fixed body*. Fixed bodies act a bit like game-driven bodies, but they do not take part in the dynamics simulation at all. They are, in effect, collision-only bodies. This optimization can give a big performance boost to most games, especially those whose worlds contain only a small number of dynamic objects moving around within a large static world.

13.5.1.4 Havok's Motion Type

In Havok, all types of rigid body are represented by instances of the class `hkpRigidBody`. Each instance contains a field that specifies its *motion type*. The motion type tells the system whether the body is fixed, game-driven (what Havok calls "key framed") or physics-driven (what Havok calls "dynamic"). If a rigid body is created with the fixed motion type, its type can never be changed. Otherwise, the motion type of a body can be changed dynamically at runtime. This feature can be incredibly useful. For example, an object that is in a character's hand would be game-driven. But as soon as the character drops or throws the object, it would be changed to physics-driven so the dynamics simulation can take over its motion. This is easily accomplished in Havok by simply changing the motion type at the moment of release.

The motion type also doubles as a way to give Havok some hints about the inertia tensor of a dynamic body. As such, the "dynamic" motion type is broken into subcategories such as "dynamic with sphere inertia," "dynamic with box inertia" and so on. Using the body's motion type, Havok can decide to apply various optimizations based on assumptions about the internal structure of the inertia tensor.

13.5.2 更新模拟

当然，物理模拟必须定期更新，通常每帧更新一次。这不仅仅涉及逐步执行模拟（数值积分、碰撞求解和应用约束）。游戏对象与其刚体之间的连接也必须保持。如果游戏需要对任何刚体施加任何力或冲量，也必须每帧执行此操作。要完全更新物理模拟，需要执行以下步骤：

• 更新游戏驱动刚体。物理世界中所有游戏驱动刚体的变换都会更新，以便它们与游戏世界中对应物（游戏对象或关节）的变换相匹配。

• 更新幻影。幻影形状就像一个游戏驱动的可碰撞体，没有对应的刚体。它用于执行某些类型的碰撞查询。所有幻影的位置都会在物理步骤之前更新，以便在运行碰撞检测时它们位于正确的位置。

• 更新力、施加冲量并调整约束。游戏施加的所有力都会更新。所有由此帧发生的游戏事件引起的冲量都会应用。如有必要，会调整约束。（例如，可能会检查一个易碎铰链以确定它是否已损坏；如果已损坏，则指示物理引擎移除该约束。）

• 逐步进行模拟。我们在 13.4.10 节中看到，碰撞引擎和物理引擎都必须定期更新。这包括对运动方程进行数值积分，以找到下一帧所有刚体的物理状态，运行碰撞检测算法，以添加和移除物理世界中所有刚体的接触，解决碰撞并应用约束。根据 SDK 的不同，这些更新阶段可能隐藏在单个原子 step() 函数后面，或者可以单独运行它们。

• 更新物理驱动的游戏对象。所有物理驱动对象的变换都从物理世界中提取，并更新相应游戏对象或关节的变换以进行匹配。

• 查询模型。物理步骤之后读取每个模型形状的接触点并用于做出决策。

• 执行碰撞投射查询。射线投射和形状投射以同步或异步方式启动。当这些查询的结果

13.5.2 Updating the Simulation

The physics simulation must of course be updated periodically, usually once per frame. This does not merely involve stepping the simulation (numerically integrating, resolving collisions and applying constraints). The linkages between the game objects and their rigid bodies must be maintained as well. If the game needs to apply any forces or impulses to any of the rigid bodies, this must also be done every frame. The following steps are required to completely update the physics simulation:

- *Update game-driven rigid bodies*. The transforms of all game-driven rigid bodies in the physics world are updated so that they match the transforms of their counterparts (game objects or joints) in the game world.
- *Update phantoms*. A phantom shape acts like a game-driven collidable with no corresponding rigid body. It is used to perform certain kinds of collision queries. The locations of all phantoms are updated prior to the physics step, so that they will be in the right places when collision detection is run.
- *Update forces, apply impulses and adjust constraints*. Any forces being applied by the game are updated. Any impulses caused by game events that occurred this frame are applied. Constraints are adjusted if necessary. (For example, a breakable hinge might be checked to determine if it has been broken; if so, the physics engine is instructed to remove the constraint.)
- *Step the simulation*. We saw in Section 13.4.10 that the collision and physics engines must both be updated periodically. This involves *numerically integrating* the equations of motion to find the physical state of all bodies on the next frame, running the *collision detection* algorithm to add and remove contacts from all rigid bodies in the physics world, *resolving collisions* and *applying constraints*. Depending on the SDK, these update phases may be hidden behind a single atomic `step()` function, or it may be possible to run them individually.
- *Update physics-driven game objects*. The transforms of all physics-driven objects are extracted from the physics world, and the transforms of the corresponding game objects or joints are updated to match.
- *Query phantoms*. The contacts of each phantom shape are read after the physics step and used to make decisions.
- *Perform collision cast queries*. Ray casts and shape casts are kicked off, either synchronously or asynchronously. When the results of these queries

变得可用时，各种引擎系统都会使用它们来做出决策。

这些任务通常按上述顺序执行，射线和形状投射除外，理论上它们可以在游戏循环的任何时间执行。显然，在步骤之前更新游戏驱动的物体并施加力和冲量是合理的，这样模拟才能“看到”效果。同样，物理驱动的游戏对象也应始终在步骤之后更新，以确保我们使用最新的物体变换。渲染通常发生在游戏循环中的所有其他操作之后。这确保了我们在特定时刻渲染的游戏世界视图始终一致。

13.5.2.1 定时冲突查询

为了向碰撞系统查询最新信息，我们需要在帧内物理步骤运行完毕后运行碰撞查询（射线和形状投射）。然而，物理步骤通常在帧的末尾运行，此时游戏逻辑已经做出了大部分决策，并且所有游戏驱动的物理实体的新位置都已确定。那么，碰撞查询应该何时运行呢？

这个问题很难回答。我们有很多选择，大多数游戏最终都会使用其中的部分或全部：

- 根据上一帧的状态做出决策。在很多情况下，可以根据上一帧的碰撞信息做出正确的决策。例如，我们可能想知道玩家在上一帧是否站在某个物体上，以便决定他是否应该在这一帧开始坠落。在这种情况下，我们可以在物理步骤之前安全地运行碰撞查询。

- 接受一帧的延迟。即使我们真的想知道这一帧发生了什么，我们也许能够容忍碰撞查询结果中有一帧的延迟。通常只有当所讨论的物体移动速度不是太快时，情况才会如此。例如，我们可能会将一个物体向前移动，然后想知道该物体现在是否在玩家的视线范围内。这种查询中的一帧错误可能不会被玩家注意到。如果是这种情况，我们可以在物理步骤之前运行碰撞查询（返回来自前一帧的碰撞信息），然后使用这些结果，就好像它们是当前帧结束时碰撞状态的近似值一样。

- 在物理步骤之后运行查询。另一种方法是在物理步骤之后运行某些查询。当需要执行决策时，这种方法是可行的。

become available, they are used by various engine systems to make decisions.

These tasks are usually performed in the order shown above, with the exception of ray and shape casts, which can theoretically be done at any time during the game loop. Clearly it makes sense to update game-driven bodies and apply forces and impulses prior to the step, so that the effects will be “seen” by the simulation. Likewise, physics-driven game objects should always be updated after the step, to ensure that we're using the most up-to-date body transforms. Rendering typically happens after everything else in the game loop. This ensures that we are rendering a consistent view of the game world at a particular instant in time.

13.5.2.1 Timing Collision Queries

In order to query the collision system for up-to-date information, we need to run our collision queries (ray and shape casts) after the physics step has run during the frame. However, the physics step is usually run toward the end of the frame, after the game logic has made most of its decisions and the new locations of any game-driven physics bodies have been determined. When, then, should collision queries be run?

This question does not have an easy answer. We have a number of options, and most games end up using some or all of them:

- *Base decisions on last frame's state*. In many cases, decisions can be made correctly based on last frame's collision information. For example, we might want to know whether or not the player was standing on something last frame, in order to decide whether or not he should start falling this frame. In this case, we can safely run our collision queries prior to the physics step.
- *Accept a one-frame lag*. Even if we really want to know what is happening *this frame*, we may be able to tolerate a one-frame lag in our collision query results. This is usually only true if the objects in question aren't moving too fast. For example, we might move one object forward in time and then want to know whether or not that object is now in the player's line of sight. A one-frame-off error in this kind of query may not be noticeable to the player. If this is the case, we can run the collision query prior to the physics step (returning collision information from the previous frame) and then use these results as if they were an *approximation* to the collision state at the end of the current frame.
- *Run the query after the physics step*. Another approach is to run certain queries after the physics step. This is feasible when the decisions being

基于查询结果的渲染可以推迟到帧的后期。例如，依赖于碰撞查询结果的渲染效果可以通过这种方式实现。

13.5.2.2 单线程更新

一个非常简单的单线程游戏循环可能看起来像这样：

```
F32 dt = 1.0f/30.0f;

for (;;) // main game loop
{
    g_hidManager->poll();

    g_gameObjectManager->preAnimationUpdate(dt);
    g_animationEngine->updateAnimations(dt);
    g_gameObjectManager->postAnimationUpdate(dt);

    g_physicsWorld->step(dt);
    g_animationEngine->updateRagDolls(dt);

    g_gameObjectManager->postPhysicsUpdate(dt);
    g_animationEngine->finalize();

    g_effectManager->update(dt);

    g_audioEngine->udate(dt);

    // etc.

    g_renderManager->render();

    dt = calcDeltaTime();
}
```

在这个例子中，我们的游戏对象分三个阶段更新：一次在动画运行之前（例如，在此期间它们可以排队新的动画），一次在动画系统计算出最终的局部姿势和暂定的全局姿势之后（但在生成最终的全局姿势和矩阵调色板之前），一次在物理系统步进之后。

- 所有游戏驱动刚体的位置通常在 preAnimationUpdate() 或 postAnimationUpdate() 中更新。每个游戏驱动刚体的变换都会设置为与其所属游戏对象或其骨骼中关节的位置相匹配。

made based on the results of the query can be deferred until late in the frame. For example, a rendering effect that depends on the results of a collision query could be implemented this way.

13.5.2.2 Single-Threaded Updating

A very simple single-threaded game loop might look something like this:

```
F32 dt = 1.0f/30.0f;

for (;;) // main game loop
{
    g_hidManager->poll();

    g_gameObjectManager->preAnimationUpdate(dt);
    g_animationEngine->updateAnimations(dt);
    g_gameObjectManager->postAnimationUpdate(dt);

    g_physicsWorld->step(dt);
    g_animationEngine->updateRagDolls(dt);

    g_gameObjectManager->postPhysicsUpdate(dt);
    g_animationEngine->finalize();

    g_effectManager->update(dt);

    g_audioEngine->udate(dt);

    // etc.

    g_renderManager->render();

    dt = calcDeltaTime();
}
```

In this example, our game objects are updated in three phases: once before animation runs (during which they can queue up new animations, for example), once after the animation system has calculated final local poses and a tentative global pose (but before the final global pose and matrix palette has been generated) and once after the physics system has been stepped.

- The locations of all game-driven rigid bodies are generally updated in `preAnimationUpdate()` or `postAnimationUpdate()`. Each game-driven body's transform is set to match the location of either the game object that owns it or a joint in the owner's skeleton.

• 通常在后 PhysicsUpdate() 中读取每个物理驱动刚体的位置，并用于更新游戏对象或其骨架中某个关节的位置。

一个重要的考虑因素是物理模拟步进的频率。大多数数值积分器、碰撞检测算法和约束求解器在步进间隔时间 (Δt) 恒定时运行最佳。通常，最好以理想的 1/30 秒或 1/60 秒时间增量步进物理/碰撞 SDK，然后控制整个游戏循环的帧速率。如果游戏帧速率低于目标值，最好让物理效果在视觉上减慢，而不是尝试调整模拟时间步长以匹配实际帧速率。

13.5.3 游戏中碰撞和物理的示例应用

为了使我们对碰撞和物理的讨论更加具体，让我们从高层次来看一些常见的例子，了解碰撞和/或物理模拟在实际游戏中的常用方法。

13.5.3.1 简单的刚体游戏对象

许多游戏包含简单的物理模拟对象，例如武器、可以拾取和投掷的石头、空弹匣、家具、架子上可以射击的物品等等。这些对象可以通过创建自定义游戏对象类并赋予其物理世界中刚体的引用来实现（例如，如果我们使用 Havok，则为 hkpRigidBody）。或者，我们可以创建一个附加组件类来处理简单的刚体碰撞和物理，从而允许将此功能添加到引擎中几乎任何类型的游戏对象中。

简单的物理对象通常会在运行时改变其运动类型。当它们被角色握在手中时，由游戏驱动；当它们被掉落并自由落体时，由物理驱动。

13.5.3.2 子弹痕迹

无论您是否赞同游戏暴力，事实是，各种形式的激光枪和投射武器是大多数游戏的重要组成部分。
让我们看看这些通常是如何实现的。
有时，投射物会使用射线投射来实现。在武器发射的那一帧，我们会发射射线，确定被击中的物体，并立即将冲击力传递给受影响的物体。

不幸的是，射线投射方法没有考虑抛射体的飞行时间。它也没有考虑轻微向下的轨迹

- The location of each physics-driven rigid body is generally read in `post-PhysicsUpdate()` and used to update the location of either the game object or one of the joints in its skeleton.

One important concern is the frequency with which you are stepping the physics simulation. Most numerical integrators, collision detection algorithms and constraint solvers operate best when the time between steps (Δt) is constant. It's usually a good idea to step your physics/collision SDK with an ideal 1/30 second or 1/60 second time delta and then govern the frame rate of your overall game loop. If your game drops below its target frame rate, it's better to let the physics slow down visually than to try to adjust the simulation time step to match the actual frame rate.

13.5.3 Example Uses of Collision and Physics in a Game

To make our discussion of collision and physics more concrete, let's take a high-level look at a few common examples of how collision and/or physics simulations are commonly used in real games.

13.5.3.1 Simple Rigid Body Game Objects

Many games include simple physically simulated objects like weapons, rocks that can be picked up and thrown, empty magazines, furniture, objects on shelves that can be shot and so on. Such objects might be implemented by creating a custom game object class and giving it a reference to a rigid body in the physics world (e.g., `hkpRigidBody` if we're using Havok). Or we might create an add-on component class that handles simple rigid body collision and physics, allowing this feature to be added to virtually any type of game object in the engine.

Simple physics objects usually change their motion type at runtime. They are game-driven when being held in a character's hand and physics-driven when in free fall after having been dropped.

13.5.3.2 Bullet Traces

Whether or not you approve of game violence, the fact remains that laser guns and projectile weapons of one form or another are a big part of most games. Let's look at how these are typically implemented.

Sometimes projectiles are implemented using ray casts. On the frame that the weapon is fired, we shoot off a ray cast, determine what object was hit and immediately impart the impact to the affected object.

Unfortunately, the ray cast approach does not account for the travel time of the projectile. It also does not account for the slight downward trajectory

重力影响造成的。如果这些细节对游戏很重要，我们可以使用真实的刚体来建模射弹，这些刚体会随着时间推移在碰撞/物理世界中移动。这对于移动速度较慢的射弹尤其有用，例如投掷的物体或火箭。顽皮狗的《最后生还者》中投掷的砖块就采用了这种方法。

在实施激光束和射弹时，有很多问题需要考虑和处理。下面讨论一些最常见的问题。

子弹射线投射

当使用光线投射检查子弹命中情况时，就会出现一个问题：光线是来自摄像机焦点，还是来自玩家角色手中枪尖？这在第三人称射击游戏中尤其成问题，因为从玩家枪中射出的光线通常与从摄像机焦点穿过屏幕中心准星的光线不一致。这会导致准星似乎位于目标上方，但第三人称角色显然位于障碍物后方，从他的视角无法射击目标的情况。通常必须使用各种“技巧”来确保玩家感觉自己正在射击目标，同时保持屏幕上的视觉效果逼真。

碰撞与可见几何体之间的不匹配

碰撞几何体与可见几何体之间的不匹配可能会导致玩家能够透过小裂缝或其他物体边缘看到目标，但碰撞几何体是实体，因此子弹无法击中目标。（这通常只对玩家角色而言是个问题。）解决此问题的一个方法是使用渲染查询而非碰撞查询来确定射线是否真正击中了目标。例如，在某个渲染过程中，我们可以生成一个纹理，其中每个像素都存储了其对应游戏对象的唯一标识符。然后，我们可以查询该纹理，以确定敌方角色或其他合适的目标是否占据了武器准星下方的像素。

在动态环境中瞄准

如果射弹需要有限的时间才能到达目标，那么 AI 控制的角色可能需要“引导”他们的射击。

caused by the influence of gravity. If these details are important to the game, we can model our projectiles using real rigid bodies that move through the collision/physics world over time. This is especially useful for slower-moving projectiles, like thrown objects or rockets. The thrown bricks in Naughty Dog's *The Last of Us* used such an approach.

There are plenty of issues to consider and deal with when implementing laser beams and projectiles. A few of the most common ones are discussed below.

Bullet Ray Casting

When using ray casting to check for bullet hits, the question arises: Does the ray come from the camera focal point or from the tip of the gun in the player character's hands? This is especially problematic in a *third-person* shooter, where the ray coming out of the player's gun usually does not align with the ray coming from the camera focal point through the reticle in the center of the screen. This can lead to situations in which the reticle appears to be on top of a target, yet the third-person character is clearly behind an obstacle and would not be able to shoot that target from his point of view. Various "tricks" must usually be employed to ensure that the player feels like he or she is shooting what he or she is aiming at while maintaining plausible visuals on the screen.

Mismatches between Collision and Visible Geometry

Mismatches between collision geometry and visible geometry can lead to situations in which the player can see the target through a small crack or just over the edge of some other object, and yet the collision geometry is solid, so the bullet cannot reach the target. (This is usually only a problem for the player character.) One solution to this problem is to use a render query instead of a collision query to determine if the ray actually hit the target. For example, during one of the rendering passes, we could generate a texture in which each pixel stores the unique identifier of the game object to which it corresponds. We can then query this texture to determine whether or not an enemy character or other suitable target occupies the pixel(s) underneath the weapon's reticle.

Aiming in a Dynamic Environment

AI-controlled characters may need to "lead" their shots if projectiles take a finite amount of time to reach their targets.

冲击效果

当子弹击中目标时，我们可能想要触发声音或粒子效果、贴上贴花或执行其他任务。

在虚幻引擎中，这是通过物理材质系统实现的。可见几何体不仅可以用视觉材质标记，还可以用物理材质标记。前者定义表面的外观，后者定义其对物理交互的反应，包括撞击声、子弹“哑弹”粒子效果、贴花等等。（更多详情，请参阅 http://udn.epicgames.com/Three/PhysicalMaterialSystem.html。）

在顽皮狗，我们使用非常类似的系统：碰撞几何体可以用多边形属性（简称 PAT）标记，这些属性定义了某些物理行为，例如脚步声。但子弹撞击的处理方式有所不同，因为我们需要它们直接与可见几何体（而非粗糙的碰撞几何体）交互。因此，可见材质可以使用可选的子弹效果标记，该效果定义了针对可能撞击该表面的每种射弹类型，应该放置哪种子弹爆破片、撞击声和贴花。

13.5.3.3 手榴弹

游戏中的手榴弹有时会被设计成自由移动的物理物体。然而，这会导致严重的失控。通过对手榴弹施加各种人为的力或冲量，可以重新获得一些控制。例如，我们可以在手榴弹第一次弹跳时施加极大的空气阻力，以限制它与目标弹跳的距离。

有些游戏团队甚至完全手动控制手榴弹的运动。手榴弹的轨迹弧度可以预先计算，使用一系列射线投射来确定投掷后会击中哪个目标。轨迹甚至可以通过某种屏幕显示方式显示给玩家。当手榴弹被投掷时，游戏会沿着弧线移动它，然后精确控制弹跳，使其不会离目标太远，同时又能保持画面的自然。

13.5.3.4 爆炸

在游戏中，爆炸通常包含几个组成部分：某种视觉效果，如火球和烟雾，模拟爆炸声音及其对世界上物体的影响的音频效果，以及影响其后任何物体的不断增加的伤害半径。

Impact Effects

When bullets hit their targets, we may want to trigger a sound or a particle effect, lay down a decal or perform other tasks.

In the Unreal engine, this is accomplished via a system of *physical materials*. Visible geometry can be tagged not only with visual materials, but with physical materials as well. The former defines how the surface looks, the latter defines how it reacts to physical interactions, including impact sounds, bullet “squib” particle effects, decals and so on. (See http://udn.epicgames.com/Three/PhysicalMaterialSystem.html for more details.)

At Naughty Dog, we use a very similar system: Collision geometry can be tagged with *polygon attributes* (PATs for short), which define certain physical behaviors like footstep sounds. But bullet impacts are treated in a special way, because we need them to interact directly with visible geometry rather than the crude collision geo. As such, visible materials can be tagged with an optional *bullet effect* that defines which bullet squib, impact sound and decal should be laid down for each possible type of projectile that might impact that surface.

13.5.3.3 Grenades

Grenades in games are sometimes implemented as free-moving physics objects. However, this leads to a significant loss of control. Some control can be regained by imposing various artificial forces or impulses on the grenade. For example, we could apply an extreme air drag once the grenade bounces for the first time, in an attempt to limit the distance it can bounce away from its target.

Some game teams actually go so far as to manage the grenade’s motion entirely manually. The arc of a grenade’s trajectory can be calculated beforehand, using a series of ray casts to determine what target it would hit if released. The trajectory can even be shown to the player via some kind of on-screen display. When the grenade is thrown, the game moves it along its arc and can then carefully control the bounce so that it never goes too far away from its target, while still looking natural.

13.5.3.4 Explosions

In a game, an explosion typically has a few components: some kind of visual effect like a fireball and smoke, audio effects to mimic the sound of the explosion and its impacts with objects in the world, and a growing damage radius that affects any objects in its wake.

当一个物体处于爆炸半径内时，其生命值通常会降低，我们通常也希望赋予其一些动作来模拟冲击波的效果。这可以通过动画来实现。（例如，角色对爆炸的反应可能最好以这种方式呈现。）我们或许还希望让冲击反应完全由动态模拟驱动。我们可以让爆炸对其半径内的任何合适物体施加脉冲来实现这一点。计算这些脉冲的方向非常容易——它们通常是径向的，通过将从爆炸中心到受冲击物体中心的矢量标准化，然后根据爆炸强度缩放该矢量（并且可能随着与爆炸中心距离的增加而衰减）。

爆炸也可能与其他引擎系统相互作用。例如，我们可能想给动画植被系统赋予一种“力量”，使草、植物和树木在爆炸的冲击波作用下瞬间弯曲。

13.5.3.5 可破坏物体

可破坏物体在许多游戏中很常见。这些物体之所以特殊，是因为它们一开始处于未损坏状态，必须看起来像一个完整的物体，但却必须能够破碎成许多碎片。我们可能希望碎片逐个破碎，让物体逐渐“消磨”；也可能只需要一次灾难性的爆炸。

像 DMM 这样的可变形体模拟可以自然地处理破坏。
然而，我们也可以使用刚体动力学来实现可破坏物体。这通常是通过将模型分割成多个可破坏的部分，并为每个部分分配一个单独的刚体来实现的。例如，Havok Destruction 就采用了这种方法。

出于性能优化和/或视觉质量的考虑，我们可能会决定使用特殊的“未损坏”版本的视觉和碰撞几何体，每个模型都构建为单个实体块。当物体需要开始分解时，可以将此模型替换为损坏的版本。在其他情况下，我们可能希望始终将物体建模为单独的块。例如，如果物体是一堆砖块或一堆锅碗瓢盆，这可能是合适的。

要建模一个多块物体，我们可以简单地堆叠一堆刚体，然后让物理模拟来处理。这可以在高质量的物理引擎中实现（尽管有时很难做到）。然而，我们可能需要一些好莱坞风格的效果，而这些效果是无法通过简单的刚体堆叠来实现的。

When an object finds itself in the radius of an explosion, its health is typically reduced, and we often also want to impart some motion to mimic the effect of the shock wave. This might be done via an animation. (For example, the reaction of a character to an explosion might best be done this way.) We might also wish to allow the impact reaction to be driven entirely by the dynamics simulation. We can accomplish this by having the explosion apply impulses to any suitable objects within its radius. It's pretty easy to calculate direction of these impulses—they are typically radial, calculated by normalizing the vector from the center of the explosion to the center of the impacted object and then scaling this vector by the magnitude of the explosion (and perhaps falling off as the distance from the epicenter increases).

Explosions may interact with other engine systems as well. For example, we might want to impart a "force" to the animated foliage system, causing grass, plants and trees to momentarily bend as a result of the explosion's shock wave.

13.5.3.5 Destructible Objects

Destructible objects are commonplace in many games. These objects are peculiar because they start out in an undamaged state in which they must appear to be a single cohesive object, and yet they must be capable of breaking into many separate pieces. We may want the pieces to break off one by one, allowing the object to be "whittled down" gradually, or we may only require a single catastrophic explosion.

Deformable body simulations like DMM can handle destruction naturally. However, we can also implement breakable objects using rigid body dynamics. This is typically done by dividing a model into a number of breakable pieces and assigning a separate rigid body to each one. This is the approach taken by Havok Destruction, for example.

For reasons of performance optimization and/or visual quality, we might decide to use special "undamaged" versions of the visual and collision geometry, each of which is constructed as a single solid piece. This model can be swapped out for the damaged version when the object needs to start breaking apart. In other cases, we may want to model the object as separate pieces at all times. This might be appropriate if the object is a stack of bricks or a pile of pots and pans, for example.

To model a multi-piece object, we could simply stack a bunch of rigid bodies and let the physics simulation take care of it. This can be made to work in good-quality physics engines (although it's not always trivial to get right). However, we may want some Hollywood-style effects that cannot be achieved with a simple stack of rigid bodies.

例如，我们可能想要定义对象的结构。有些部件可能是不可摧毁的，比如墙基或汽车底盘。其他部件可能是非结构性的——它们被子弹或其他物体击中时会掉落。还有一些部件可能是结构性的——如果它们被击中，它们不仅会掉落，还会将力传递给位于其上方的其他部件。有些部件可能是爆炸性的——当它们被击中时，它们会引发二次爆炸或将损坏扩散到整个结构。我们可能希望某些部件作为某些角色的有效掩护点，而不是其他角色。这意味着我们的可破坏物体系统可能与掩护系统有一些联系。

我们或许还希望可破碎物体拥有生命值的概念。损伤可能会逐渐累积，直至最终完全崩塌；或者，每个碎片也可能拥有生命值，需要多次射击或撞击才能破碎。此外，还可以使用约束，使破碎的碎片能够悬挂在物体上，而不是完全脱离物体。

我们可能还希望我们的结构需要时间才能完全坍塌。例如，如果一座长桥的一端遭遇爆炸，坍塌应该缓慢地从一端蔓延到另一端，使桥看起来巨大无比。这又是一个物理系统不会免费提供的功能的例子——它会同时唤醒模拟岛屿上的所有刚体。这类效果可以通过合理使用游戏驱动的运动类型来实现。

13.5.3.6 角色机制

对于像保龄球、弹球或《疯狂弹珠》这样的游戏来说，“主角”是一个在虚拟游戏世界中滚动的球。对于这类游戏，我们可以在物理模拟中将球建模为自由移动的刚体，并通过在游戏过程中施加力和冲量来控制它的运动。

然而，在基于角色的游戏中，我们通常不会采用这种方法。人形或动物角色的运动通常过于复杂，难以通过力和脉冲进行充分控制。相反，我们通常将角色建模为一组由游戏驱动的胶囊状刚体，每个刚体都链接到角色动画骨架中的一个关节。这些刚体主要用于子弹命中检测或产生次要效果，例如角色的手臂撞到桌子上的物体。由于这些刚体是由游戏驱动的，它们无法避免与物理世界中不可移动的物体相互穿透，因此动画师需要确保角色的动作看起来逼真。

为了在游戏世界中移动角色，大多数游戏使用球体或胶囊体来探测所需运动的方向。碰撞是

For example, we may want to define the structure of the object. Some pieces might be *indestructible*, like the base of a wall or the chassis of a car. Others might be *non-structural*—they just fall off when hit by bullets or other objects. Still other pieces might be *structural*—if they are hit, not only do they fall, but they also impart forces to other pieces lying on top of them. Some pieces could be *explosive*—when they are hit, they create secondary explosions or propagate damage throughout the structure. We may want some pieces to act as valid *cover* points for some characters, but not others. This implies that our breakable object system may have some connections to the cover system.

We might also want our breakable objects to have a notion of health. Damage might build up until eventually the whole thing collapses, or each piece might have a health, requiring multiples shots or impacts before it is allowed to break. Constraints might also be employed to allow broken pieces to hang off the object rather than coming away from it completely.

We may also want our structures to take time to collapse completely. For example, if a long bridge is hit by an explosion at one end, the collapse should slowly propagate from one end to the other so that the bridge looks massive. This is another example of a feature the physics system won't give you for free—it would just wake up all rigid bodies in the simulation island simultaneously. These kinds of effects can be implemented through judicious use of the game-driven motion type.

13.5.3.6 Character Mechanics

For a game like bowling, pinball or *Marble Madness*, the "main character" is a ball that rolls around in an imaginary game world. For this kind of game, we could very well model the ball as a free-moving rigid body in the physics simulation and control its movements by applying forces and impulses to it during gameplay.

In character-based games, however, we usually don't take this kind of approach. The locomotion of a humanoid or animal character is usually far too complex to be controlled adequately with forces and impulses. Instead, we usually model characters as a set of game-driven capsule-shaped rigid bodies, each one linked to a joint in the character's animated skeleton. These bodies are primarily used for bullet hit detection or to generate secondary effects such as when a character's arm bumps an object off a table. Because these bodies are game-driven, they won't avoid interpenetrations with immovable objects in the physics world, so it is up to the animator to ensure that the character's movements appear believable.

To move the character around in the game world, most games use sphere or capsule casts to probe in the direction of desired motion. Collisions are

手动解决。这让我们可以做一些很酷的事情，比如：

- 当角色以斜角撞到墙壁时，他会沿着墙壁滑动；
- 允许角色“弹出”低矮的路缘而不是被卡住；
- 防止角色走下低矮路缘时进入“坠落”状态；
- 防止角色走上过于陡峭的斜坡（大多数游戏都有一个截止角度，超过这个角度角色就会向后滑落，而无法走上斜坡）；
- 调整动画以适应碰撞。

举个例子，如果角色以大约 90 度角直接撞到墙上，我们可以让角色一直“月球漫步”撞到墙上，或者放慢动画速度。我们还可以做得更巧妙，比如播放一个动画，让角色伸出手触摸墙壁，然后保持静止，直到移动方向改变。

Havok 提供了一个角色控制器系统来处理这些任务。如图 13.36 所示，Havok 的系统将角色建模为一个胶囊模型，该模型每帧都会移动以寻找潜在的新位置。系统会为角色维护一个碰撞接触流形（即一组经过清理以消除噪点的接触平面）。可以逐帧分析该流形，以确定如何以最佳方式移动角色、调整动画等等。

13.5.3.7 相机碰撞

在许多游戏中，摄像机会跟随玩家的角色或车辆在游戏世界中移动，并且通常只能由玩家旋转或进行有限度的控制。在这类游戏中，务必确保摄像机不会穿透场景中的几何体，因为这会破坏真实感。因此，在许多游戏中，摄像机系统是碰撞引擎的另一个重要客户端。

大多数相机碰撞系统的基本思想是，在虚拟相机周围放置一个或多个球体幻影或球体投射查询，这些幻影或查询可以检测相机何时接近碰撞。系统可以通过某种方式调整相机的位置和/或方向，以避免在相机实际穿过目标物体之前发生潜在的碰撞。

resolved manually. This allows us to do cool stuff like:

- having the character slide along walls when he runs into them at an oblique angle;
- allowing the character to “pop up” over low curbs rather than getting stuck;
- preventing the character from entering a “falling” state when he walks off a low curb;
- preventing the character from walking up slopes that are too steep (most games have a cutoff angle after which the character will slide back rather than being able to walk up the slope); and
- adjusting animations to accommodate collisions.

As an example of this last point, if the character is running directly into a wall at a roughly 90 degree angle, we can let the character “moonwalk” into the wall forever, or we can slow down his animation. We can also do something even more slick, like playing an animation in which the character sticks out his hand and touches the wall and then idles sensibly until the movement direction changes.

Havok provides a character controller system that handles many of these things. In Havok’s system, illustrated in Figure 13.36, a character is modeled as a capsule phantom that is moved each frame to find a potential new location. A collision contact manifold (i.e., a collection of contact planes, cleaned up to eliminate noise) is maintained for the character. This manifold can be analyzed each frame in order to determine how best to move the character, adjust animations and so on.

13.5.3.7 Camera Collision

In many games, the camera follows the player’s character or vehicle around in the game world, and it can often be rotated or controlled in limited ways by the person playing the game. It’s important in such games to never permit the camera to interpenetrate geometry in the scene, as this would break the illusion of realism. The camera system is therefore another important client of the collision engine in many games.

The basic idea behind most camera collision systems is to surround the virtual camera with one or more sphere phantoms or sphere cast queries that can detect when it is getting close to colliding with something. The system can respond by adjusting the camera’s position and/or orientation in some way to avoid the potential collision before the camera actually passes through the object in question.

图 13.36 Havok 的角色控制器将角色建模为一个胶囊状的幻影。该幻影维持一个降噪的碰撞流形（一组接触平面），可供游戏用于做出运动决策。

这听起来很简单，但实际上却是一个非常棘手的问题，需要大量的反复试验才能解决。为了让您了解这需要付出多少努力，许多游戏团队都会安排一名专职工程师在整个项目期间负责相机系统。我们无法在此深入探讨相机碰撞检测和分辨率，但以下列表应该可以让您了解一些最需要注意的问题：

- 在各种情况下，放大镜头以避免碰撞都很有效。在第三人称游戏中，你可以将镜头拉近到第一人称视角，而不会造成太多麻烦（除了确保镜头在此过程中不会穿透角色的头部）。

- 通常情况下，为了应对碰撞而大幅改变摄像机的水平角度并非明智之举，因为这往往会干扰玩家相对于摄像机的控制。然而，根据玩家当时的预期操作，进行一定程度的水平调整可能会取得良好的效果。如果玩家正在瞄准目标，而你为了避免碰撞而打断她的瞄准，她会很生气。但如果她只是在世界中移动，那么改变摄像机方向可能会感觉非常自然。因此，你可能希望仅当主角不在场时才允许调整摄像机的水平角度。

Figure 13.36. Havok's character controller models a character as a capsule-shaped phantom. The phantom maintains a noise-reduced collision manifold (a collection of contact planes) that can be used by the game to make movement decisions.

This sounds simple enough, but it is actually an incredibly tricky problem requiring a great deal of trial and error to get right. To give you a feel for how much effort can be involved, many game teams have a dedicated engineer working on the camera system for the entire duration of the project. We can't possibly cover camera collision detection and resolution in any depth here, but the following list should give you a sense of some of the most pertinent issues to be aware of:

- Zooming the camera in to avoid collisions works well in a wide variety of situations. In a third-person game, you can zoom all the way in to a first-person view without causing too much trouble (other than making sure the camera doesn't interpenetrate the character's head in the process).
- It's usually a bad idea to drastically change the horizontal angle of the camera in response to collisions, as this tends to mess with camera-relative player controls. However, some degree of horizontal adjustment can work well, depending on what the player is expected to be doing at the time. If she is aiming at a target, she'll be angry with you if you throw off her aim to bring the camera out of collision. But if she's just locomoting through the world, the change in camera orientation may feel entirely natural. Because of this, you might want to allow adjustments to the horizontal angle of the camera only when the main character is not

在激烈的战斗中。

• 您可以在一定程度上调整摄像机的垂直角度，但重要的是不要调整太多，否则玩家将看不到地平线，最终俯视玩家角色的头顶！

• 某些游戏允许摄像机沿着垂直平面上的圆弧移动，该圆弧可能由样条曲线描述。这使得单个 HID 控件（例如左拇指摇杆的垂直偏转）能够以直观的方式同时控制摄像机的缩放和垂直角度。（顽皮狗引擎中的摄像机就是这样工作的。）当摄像机与场景中的物体发生碰撞时，它可以自动沿着相同的圆弧移动以避免碰撞，圆弧也可以水平压缩，或者可以采取许多其他方法。

• 不仅要考虑摄像机前后方的情况，还要考虑摄像机前方的情况。例如，如果一根柱子或其他角色挡在摄像机和玩家角色之间，会发生什么？在某些游戏中，碰撞的物体会变成半透明；在其他游戏中，摄像机会放大或旋转以避免碰撞。这可能会让游戏玩家感到不舒服！你如何处理这些情况，可能会影响游戏的视觉质量。

即使考虑到这些以及其他许多可能出现的问题，你的相机可能看起来或感觉起来仍然不对劲！在实现相机碰撞系统时，一定要预留充足的时间进行反复试验。

13.5.3.8 布娃娃集成

在 13.4.8.7 节中，我们学习了如何使用特殊类型的约束将一组刚体连接在一起，以模拟瘫痪（死亡或失去意识）的人体行为。在本节中，我们将探讨在游戏中集成布娃娃物理时出现的一些问题。

正如我们在 13.5.3.6 节中看到的，有意识角色的总体动作通常是通过执行形状转换或在游戏世界中移动幻影形状来确定的。角色身体的详细动作通常由动画驱动。游戏驱动的刚体有时会附加到四肢上，用于武器瞄准或允许角色撞倒世界中的其他物体。

当角色失去意识时，布娃娃系统就会启动。角色的四肢被建模为胶囊状刚体，通过约束连接，并与角色动画骨架中的关节相连。

in the heat of a battle.

- You can adjust the vertical angle of the camera to some degree, but it's important not to do too much of this, or the player will lose track of the horizon and end up looking down onto the top of the player character's head!
- Some games allow the camera to move along an arc lying in a vertical plane, perhaps described by a spline. This permits a single HID control such as the vertical deflection of the left thumb stick to control both the zoom and the vertical angle of the camera in an intuitive way. (The camera in the Naughty Dog engine works this way.) When the camera comes into collision with objects in the world, it can be automatically moved along this same arc to avoid the collision, the arc might be compressed horizontally, or any number of other approaches might be taken.
- It's important to consider not only what's behind and beside the camera but what is in front of it as well. For example, what should happen if a pillar or another character comes between the camera and the player character? In some games, the offending object becomes translucent; in others, the camera zooms in or swings around to avoid the collision. This may or may not feel good to the person playing the game! How you handle these kinds of situations can make or break the perceived quality of your game.

Even after taking account of these and many other problematic situations, your camera may not look or feel right! Always budget plenty of time for trial and error when implementing a camera collision system.

13.5.3.8 Rag Doll Integration

In Section 13.4.8.7, we learned how special types of constraints can be used to link a collection of rigid bodies together to mimic the behavior of a limp (dead or unconscious) human body. In this section, we'll investigate a few of the issues that arise when integrating rag doll physics into your game.

As we saw in Section 13.5.3.6, the gross movements of a conscious character are usually determined by performing shape casts or moving a phantom shape around in the game world. The detailed movements of the character's body are typically driven by animations. Game-driven rigid bodies are sometimes attached to the limbs for the purposes of weapons targeting or to allow the character to knock over other objects in the world.

When a character becomes unconscious, the rag doll system kicks in. The character's limbs are modeled as capsule-shaped rigid bodies connected via constraints and linked to joints in the character's animated skeleton. The

物理系统模拟这些身体的运动，我们更新骨骼关节以进行匹配，从而允许物理移动角色的身体。

用于布娃娃物理的刚体集合可能与角色活着时肢体上的刚体集合不同。这是因为这两个碰撞模型的要求截然不同。当角色活着时，它的刚体由游戏驱动，所以我们不关心它们是否相互穿透。事实上，我们通常希望它们重叠，这样就不会出现任何敌方角色可能射穿的孔洞。但是当角色变成布娃娃时，刚体不能相互穿透就很重要，因为这会导致碰撞解决系统传递巨大的冲量，从而倾向于使肢体向外爆炸！出于这些原因，角色有意识还是无意识时，拥有完全不同的碰撞/物理表现形式是很常见的。

另一个问题是如何从意识状态过渡到无意识状态。动画生成的姿势和物理生成的姿势之间的简单 LERP 混合通常效果不佳，因为物理姿势很快就会偏离动画姿势。（两个完全不相关的姿势之间的混合通常看起来不自然。）因此，我们可能需要在过渡期间使用动力约束（参见第 13.4.8.8 节）。

角色在有意识时（即其刚体由游戏驱动时），通常会穿透背景几何体。这意味着，当角色转换为布娃娃（物理驱动）模式时，刚体可能位于另一个固体物体内部。这可能会产生巨大的冲击力，导致游戏中布娃娃的行为看起来相当狂野。为了避免这些问题，最好谨慎编写死亡动画，尽可能避免角色的肢体发生碰撞。在游戏驱动模式下，通过幻影或碰撞回调检测碰撞也很重要，这样当角色身体的任何部位接触到固体物体时，您就可以立即将其切换到布娃娃模式。

即使采取了这些措施，布娃娃仍然容易卡在其他物体内。单面碰撞对于让布娃娃看起来更美观来说是一个非常重要的因素。如果肢体的一部分嵌入墙内，它往往会被推出墙外，而不是卡在墙内。然而，即使是单面碰撞也无法解决所有问题。例如，当角色快速移动或转换为布娃娃时，布娃娃中的一个刚体可能会卡在薄墙的另一侧。这会导致角色悬在半空中，而不是正常落地。

另一个有用的布娃娃功能是无意识的

physics system simulates the motions of these bodies, and we update the skeletal joints to match, thereby allowing physics to move the character's body.

The set of rigid bodies used for rag doll physics might not be the same ones affixed to the character's limbs when it was alive. This is because the two collision models have very different requirements. When the character is alive, its rigid bodies are game-driven, so we don't care if they interpenetrate. And in fact, we usually want them to overlap, so there aren't any holes through which an enemy character might shoot. But when the character turns into a rag doll, it's important that the rigid bodies do not interpenetrate, as this would cause the collision resolution system to impart large impulses that would tend to make the limbs explode outward! For these reasons, it's actually quite common for characters to have entirely different collision/physics representations depending on whether they're conscious or unconscious.

Another issue is how to transition from the conscious state to the unconscious state. A simple LERP blend between animation-generated and physics-generated poses usually doesn't work very well, because the physics pose very quickly diverges from the animation pose. (A blend between two totally unrelated poses usually doesn't look natural.) As such, we may want to use powered constraints during the transition (see Section 13.4.8.8).

Characters often interpenetrate background geometry when they are conscious (i.e., when their rigid bodies are game-driven). This means that the rigid bodies might be inside another solid object when the character transitions to rag doll (physics-driven) mode. This can give rise to huge impulses that cause rather wild-looking rag doll behavior in-game. To avoid these problems, it is best to author death animations carefully, so that the character's limbs are kept out of collision as best as possible. It's also important to detect collisions via phantoms or collision callbacks during the game-driven mode so that you can drop the character into rag doll mode the moment any part of his body touches something solid.

Even when these steps are taken, rag dolls have a tendency to get stuck inside other objects. Single-sided collision can be an incredibly important feature when trying to make rag dolls look good. If a limb is partly embedded in a wall, it will tend to be pushed out of the wall rather than staying stuck inside it. However, even single-sided collision doesn't solve all problems. For example, when the character is moving quickly or if the transition to rag doll isn't executed properly, one rigid body in the rag doll can end up on the far side of a thin wall. This causes the character to hang in mid-air rather than falling properly to the ground.

Another rag doll feature that can be useful is the ability for unconscious

角色恢复意识并重新站起来。为了实现这一点，我们需要一种方法来搜索合适的“站起来”动画。我们希望找到一个动画，其在第 0 帧时的姿势与布娃娃静止后的姿势最接近（这通常是完全不可预测的）。这可以通过仅匹配几个关键关节（例如大腿和上臂）的姿势来实现。另一种方法是使用动力约束，手动引导布娃娃在静止时进入适合站起来的姿势。

最后，我们应该提一下，设置布娃娃的约束可能很棘手。我们通常希望肢体能够自由活动，但又不至于做出任何生物力学上不可能的动作。这也是在制作布娃娃时经常使用特殊约束的原因之一。不过，你不应该想当然地认为你的布娃娃不费吹灰之力就能看起来很棒。像 Havok 这样的高质量物理引擎提供了丰富的内容创作工具，允许艺术家在 Maya 等 DCC 软件包中设置约束，然后实时测试它们，看看它们在游戏中的效果。

总而言之，让布娃娃的物理效果在游戏中发挥作用并不难，但要让它看起来美观，可就费劲了！就像游戏编程中的许多事情一样，最好预留充足的时间进行反复试验，尤其是在你第一次使用布娃娃的时候。

13.6 高级物理特性

带有约束的刚体动力学模拟可以覆盖游戏中种类繁多的物理驱动效果。然而，这样的系统显然存在局限性。最近的研发正在寻求将物理引擎扩展到受约束的刚体之外。以下仅列举几个例子：

- 可变形体。随着硬件性能的提升和更高效算法的开发，物理引擎开始支持可变形体。DMM 就是此类引擎的一个很好的例子。

- 布料。布料可以建模为由刚性弹簧连接的点质量片。众所周知，布料的制作难度很大，因为布料与其他物体的碰撞、模拟的数值稳定性等诸多问题都存在。即便如此，许多游戏和第三方物理 SDK（例如 Havok）现在都提供了强大且性能良好的布料模拟，可用于游戏和其他实时应用。

characters to regain consciousness and get back up. To implement this, we need a way to search for a suitable “stand up” animation. We want to find an animation whose pose on frame zero most closely matches the rag doll’s pose after it has come to rest (which is totally unpredictable in general). This can be done by matching the poses of only a few key joints, like the upper thighs and the upper arms. Another approach is to manually guide the rag doll into a pose suitable for getting up by the time it comes to rest, using powered constraints.

As a final note, we should mention that setting up a rag doll’s constraints can be a tricky business. We generally want the limbs to move freely but without doing anything biomechanically impossible. This is one reason specialized types of constraints are often used when constructing rag dolls. Nonetheless, you shouldn’t assume that your rag dolls will look great without some effort. High-quality physics engines like Havok provide a rich set of content creation tools that allow an artist to set up constraints within a DCC package like Maya and then test them in real time to see how they might look in-game.

All in all, getting rag doll physics to *work* in your game isn’t particularly difficult, but getting it to look *good* can take a lot of work! As with many things in game programming, it’s a good idea to budget plenty of time for trial and error, especially when it’s your first time working with rag dolls.

13.6 Advanced Physics Features

A rigid body dynamics simulation with constraints can cover an amazing range of physics-driven effects in a game. However, such a system clearly has its limitations. Recent research and development is seeking to expand physics engines beyond constrained rigid bodies. Here are just a few examples:

- *Deformable bodies*. As hardware capabilities improve and more-efficient algorithms are developed, physics engines are beginning to provide support for deformable bodies. DMM is an excellent example of such an engine.
- *Cloth*. Cloth can be modeled as a sheet of point masses, connected by stiff springs. Cloth is notoriously difficult to get right, as many difficulties arise with respect to collision between cloth and other objects, numerical stability of the simulation, etc. That being said, many games and third-party physics SDKs like Havok now provide powerful and well-behaved cloth simulations for use in games and other real-time applications.

• 头发。头发可以用大量小型物理模拟薄膜来建模，或者也可以采用更简单的方法，将布料纹理贴图模拟成头发的样子，并调整布料模拟，使角色的头发以逼真的方式摆动。《神秘海域：失落的遗产》中克洛伊的头发就是这样设计的。头发模拟和渲染仍然是一个活跃的研究领域，游戏中头发的质量必将持续改进。

• 水面模拟和浮力。游戏中已经存在一段时间的水面模拟和浮力。浮力可以通过特殊系统（本身不是物理引擎的一部分）实现，也可以通过物理模拟中的力进行建模。水面的有机运动通常只是渲染效果，根本不影响物理模拟。从物理学的角度来看，水面通常被建模为一个平面。对于水面的大位移，整个平面可能会移动。然而，一些游戏团队和研究人员正在突破这些模拟的极限，允许动态水面、波峰波浪、逼真的水流模拟等等。上帝游戏《From Dust》中的水就是一个很好的例子。

• 通用流体动力学模拟。目前，流体动力学主要存在于专门的模拟库中。然而，这是一个活跃的研究和开发领域，一些游戏已经利用流体模拟来制作一些令人惊叹的视觉效果。例如，《小小大星球》系列利用二维流体模拟来制作烟雾和火焰效果；而 PhysX SDK 则提供了基于位置的三维流体模拟，可以产生令人惊叹的逼真效果。

• 基于物理的音频合成。当物理模拟的物体发生碰撞、弹跳、滚动和滑动时，能够生成合适的音频来增强模拟的可信度至关重要。这些声音可以通过控制播放预先录制的音频片段在游戏中创建。但动态合成此类声音正逐渐成为一种可行的替代方案，并且目前是一个活跃的研究领域。

• 通用 GPU (GPGPU)。随着 GPU 的功能越来越强大，人们开始将其强大的并行处理能力用于图形处理以外的任务。通用 GPU (GPGPU) 计算的一个常见应用是碰撞和物理模拟。例如，顽皮狗的布料模拟引擎已移植到 PlayStation 4 上，完全在 GPU 上运行。

- *Hair*. Hair can be modeled by a large number of small physically simulated filments, or a simpler approach can be used in which sheets of cloth are texture-mapped to look like hair, and the cloth simulation is tuned to make the character's hair move in a believable way. This is how Chloe's hair in *Uncharted: The Lost Legacy* works. Hair simulation and rendering remains an active area of research, and the quality of hair in games will certainly continue to improve.
- *Water surface simulations and buoyancy*. Games have been doing water surface simulations and buoyancy for some time now. Buoyancy can be implemented via a special-case system (not part of the physics engine per se), or it can be modeled via forces within the physics simulation. Organic movement of the water surface is often a rendering effect only and does not affect the physics simulation at all. From the point of view of physics, the water surface is often modeled as a plane. For large displacements in the water surface, the entire plane might be moved. However, some game teams and researchers are pushing the limits of these simulations, allowing for dynamic water surfaces, waves that crest, realistic current simulations and more. One good example is the water in the god game *From Dust*.
- *General fluid dynamics simulations*. Right now, fluid dynamics falls primarily into the realm of specialized simulation libraries. However, this is an active area of research and development, and some games already make use of fluid simulations to produce some astounding visual effects. For example, the *LittleBigPlanet* series makes use of a 2D fluid simulation for its smoke and fire effects; and the PhysX SDK offers a 3D *position based* fluid simulation that produces stunningly realistic results.
- *Physically based audio synthesis*. When physically simulated objects collide, bounce, roll and slide, it's important to be able to generate appropriate audio to reinforce the believability of the simulation. These sounds can be created in games via controlled playback of pre-recorded audio clips. But dynamic synthesis of such sounds is becoming a viable alternative, and is currently an active area of research.
- *GPGPU*. As GPUs become more and more powerful, there has been a shift toward harnessing their awesome parallel processing power for tasks other than graphics. One obvious application of general-purpose GPU (GPGPU) computing is for collision and physics simulation. For example, Naughty Dog's cloth simulation engine was ported to run entirely on the GPU for PlayStation 4.

14
声音的

如果你曾经在静音状态下观看恐怖电影，你就会明白音频对于沉浸感有多么重要。（如果没有，不妨试试！它真的能让你耳目一新。）无论是电影还是电子游戏，音效都能让你体验扣人心弦、令人动容、令人难忘的多媒体体验，而不是乏味无聊的无聊体验。

现代游戏让玩家沉浸在逼真（或半逼真但风格化）的虚拟环境中。图形引擎的任务是尽可能准确、可信地重现玩家身临其境时所看到的内容（同时保持游戏的艺术风格）。同样，音频引擎的任务是准确、可信地重现玩家身临其境时所听到的内容（同时保持游戏的虚构和音调风格）。如今，声音程序员使用“音频渲染引擎”一词来强调其与图形渲染引擎的诸多相似之处。

在本章中，我们将探讨为 3A 游戏制作音频的理论和实践。我们将介绍一个叫做信号处理理论的数学领域，它几乎涵盖了数字音频技术的各个方面，包括数字录音和播放、滤波、混响和其他数字信号处理器 (DSP) 效果。我们将从软件工程的角度探索游戏音频，研究一些广泛使用的

14
Audio

If you've ever watched a horror film with your speakers muted, you know just how important audio is to immersiveness. (If not, try it! It's a real eye-opener.) Be it a film or a video game, sound can quite literally make the difference between a gripping, emotional, unforgettable multimedia experience and a lackluster yawnfest.

Modern games immerse the player in a realistic (or a semi-realistic but stylized) virtual environment. The graphics engine is charged with the task of reproducing as accurately and believably as possible what the player would actually see, if he or she were present within this virtual environment (while remaining true to the art style of the game). In exactly the same sense, the audio engine is charged with the task of accurately and believably reproducing what the player would actually hear, if he or she were present in the game world (while remaining true to the fiction and tonal style of the game). Sound programmers today use the term audio *rendering* engine to underscore its many parallels with the graphics rendering engine.

In this chapter, we're going to explore both the theory and practice of creating audio for a triple-A game. We'll introduce an area of mathematics called *signal processing theory* that underlies almost every aspect of digital audio technology, including digital sound recording and playback, filtering, reverb and other digital signal processor (DSP) effects. We'll explore game audio from a software engineering standpoint, by investigating a number of widely used

音频 API，分解典型音频渲染引擎的组件，并了解音频系统如何与其他游戏引擎系统互连。我们还将了解顽皮狗的热门游戏《最后生还者》中环境声学建模和角色对话的处理方式。所以，抓紧车，双手始终放在车内，享受这充满噪音的旅程吧！

14.1 声音的物理学

声音是一种在空气（或其他可压缩介质）中传播的压缩波。声波会相对于平均大气压产生交替的空气压缩和减压区域（也称为稀薄化）。因此，我们用压力（压强）来测量声波的振幅。在国际单位制中，压力以帕斯卡（缩写为Pa）为单位。1帕斯卡等于1牛顿作用于1平方米面积上的力。

$(1\ \mathrm{Pa} = 1\ \mathrm{N/m^2} = 1\ \mathrm{kg/(m \cdot s^2)})$.
瞬时声压是环境大气压（就我们的目的而言，视为常数）加上某一特定时刻声波引起的扰动：

$$p_{\mathrm{inst}} = p_{\mathrm{atmos}} + p_{\mathrm{sound}}。$$

当然，声音是一种动态现象——声压会随时间变化。我们可以绘制瞬时声压随时间变化的函数图，即 $p_{\mathrm{inst}}(t)$。在信号处理理论（几乎涵盖数字音频技术各个方面的数学基础）中，这种随时间变化的函数被称为信号。图 14.1 展示了一个典型的声波信号 $p(t)$，它围绕平均大气压振荡。

图 14.1. 信号 $p(t)$ 可用于模拟声音随时间变化的声压。

audio APIs, breaking down the components that comprise a typical audio rendering engine and learning how the audio system is interconnected with other game engine systems. We'll also see how environmental acoustic modeling and character dialog were handled in Naughty Dog's popular game *The Last of Us*. So hold on tight, keep your hands inside the car at all times and enjoy the noisy ride!

14.1 The Physics of Sound

Sound is a compression wave that travels through the air (or some other compressible medium). A sound wave gives rise to alternating regions of air compression and decompression (also known as *rarefaction*) relative to the average atmospheric pressure. As such, we measure the *amplitude* of a sound wave in units of *pressure*. In SI units, pressure is measured in Pascals, abbreviated Pa. One Pascal is the force of one Newton applied over an area of one square meter $(1\ \mathrm{Pa} = 1\ \mathrm{N/m^2} = 1\ \mathrm{kg/(m \cdot s^2)})$.

The *instantaneous acoustic pressure* is the ambient atmospheric pressure (considered a constant for our purposes) plus the perturbation caused by the sound wave at one specific instant in time:

$$p_{\mathrm{inst}} = p_{\mathrm{atmos}} + p_{\mathrm{sound}}.$$

Of course, sound is a dynamic phenomenon—the sound pressure varies over time. We can plot the instantaneous sound pressure as a function of time, $p_{\mathrm{inst}}(t)$. In *signal processing theory*—the area of mathematics that underlies virtually every aspect of digital audio technology—such a time-varying function is called a *signal*. Figure 14.1 illustrates a typical sound wave signal $p(t)$, oscillating about the average atmospheric pressure.

Figure 14.1. A signal $p(t)$ can be used to model the time-varying acoustic pressure of a sound.

图 14.2 任意周期信号的周期 T 是波形中重复模式之间的最短时间。

14.1.1 声波的性质

当乐器演奏一个长而稳定的音符时，产生的声压信号具有周期性，这意味着波形由该特定乐器特有的重复模式组成。任何重复模式的周期T描述的是该模式连续出现之间的最短时间。例如，对于正弦声波，周期测量的是连续波峰或波谷之间的时间。在国际单位制（SI）中，周期通常以秒（s）为单位。如图14.2所示。

波的频率恰好是其周期的倒数（f = 1/T）。频率的单位是赫兹 (Hz)，即“每秒周期数”。严格来说，“周期”是一个无量纲量，因此赫兹是秒的倒数（Hz = 1/s）。

许多科学家和数学家会使用一个称为角频率的量，通常用符号ω表示。角频率就是以弧度/秒而不是周期/秒为单位测量的振荡速率。由于一圈完整的圆周旋转为2π弧度，因此ω=2πf=2π/T。角频率在分析正弦波时非常有用，因为二维的圆周运动投影到一维轴上时会产生正弦运动。

周期信号（例如正弦波）沿时间轴向左或向右移动的量称为其相位。相位是一个相对术语。例如，sin(t)实际上只是cos(t)的一个版本，沿t轴相移了+π2（即sin(t)=cos(t-π2)）。同样，cos(t)只是sin(t)相移了-π2（即cos(t)=sin(t+π2)）。相位如图14.3所示。

声波在介质中传播的速度 v 取决于介质的材质和物理特性，包括相态（固体、气体或液体）、温度、压力和密度。在 20° C 干燥

Figure 14.2. The period T of an arbitrary periodic signal is the minimum time between repeated patterns in the waveform.

14.1.1 Properties of Sound Waves

When a musical instrument plays a long steady note, the resulting sound pressure signal is *periodic*, meaning the waveform consists of a *repeating pattern* characteristic to that particular kind of instrument. The *period* T of any repeating pattern describes the minimum amount of time that passes between successive instances of the pattern. For example, for a sinusoidal sound wave the period measures the time between successive peaks or troughs. In SI units, period is typically measured in seconds (s). This is illustrated in Figure 14.2.

The *frequency* of a wave is just the inverse of its period ($f = 1/T$). Frequency is measured in Hertz (Hz), which means “cycles per second.” A “cycle” is technically a dimensionless quantity, so the Hertz is the inverse of the second (Hz $= 1/$s).

Many scientists and mathematicians make use of a quantity known as the *angular frequency*, typically denoted by the symbol ω. The angular frequency is just the rate of oscillation measured in *radians* per second instead of *cycles* per second. Since one complete circular rotation is 2π radians, $\omega = 2\pi f = 2\pi/T$. Angular frequency is very useful when analyzing sinusoidal waves, because a circular motion in two dimensions gives rise to a sinusoidal motion when projected onto a single dimensional axis.

The amount by which a periodic signal such as a sine wave is *shifted* left or right along the time axis is known as its *phase*. Phase is a relative term. For example, $\sin(t)$ is really just a version of $\cos(t)$ that has been *phase-shifted* by $+\frac{\pi}{2}$ along the t axis (i.e., $\sin(t) = \cos(t - \frac{\pi}{2})$). Likewise $\cos(t)$ is just $\sin(t)$ phase-shifted by $-\frac{\pi}{2}$ (i.e., $\cos(t) = \sin(t + \frac{\pi}{2})$). Phase is illustrated in Figure 14.3.

The *speed* v at which a sound wave propagates through its medium depends upon the material and physical properties of the medium, including phase (solid, gas or liquid), temperature, pressure and density. In 20°C dry

图 14.3。正弦函数和余弦函数只是彼此的相移版本。

空气中，声音的速度约为 343.2 米/秒，即 767.7 英里/小时或 1235.6 公里/小时。

正弦波的波长λ测量的是连续波峰或波谷之间的空间距离。它部分取决于波的频率，但由于它是空间测量，因此也取决于波速。具体来说，λ = v / f，其中v是波速（以米/秒为单位），f是频率（以赫兹或1/秒为单位）。分子和分母中的秒相互抵消，最终得到的波长以米为单位。

14.1.2 感知响度和分贝

为了判断我们听到的声音的“响度”，我们的耳朵会在短暂的滑动时间窗口内不断平均传入声音信号的幅度。这种平均效应可以用一个称为有效声压的量来很好地描述。有效声压的定义是特定时间间隔内测得的瞬时声压的均方根 (RMS)。

如果我们在时间上等间隔地进行一系列 n 次离散声压测量 pi ，则 RMS 声压 p rms 将是

$$p_{\text{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} p_i^2}. \tag{14.1}$$

然而，我们的耳朵会持续测量声压，而不是在离散的时间点进行测量。假设我们连续测量瞬时声压，从时刻 T 1 开始，一直持续到时刻 T 2 ，那么公式 (14.1) 中的求和运算将变成如下的积分：

$$p_{\text{rms}} = \sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2} (p(t))^2\, dt}. \tag{14.2}$$

Figure 14.3. The sine and cosine functions are just phase-shifted versions of one another.

air, the speed of sound is approximately 343.2 m/s, which is 767.7 mph or 1235.6 km/h.

The *wavelength* λ of a sinusoidal wave measures the spatial distance between successive peaks or troughs. It depends in part on the frequency of the wave, but because it is a *spatial* measurement it also depends on the speed of the wave. Specifically, $\lambda = v/f$ where v is the speed of the wave (measured in m/s) and f is the frequency (measured in Hz or 1/s). The seconds in the numerator and denominator cancel one another, and we are left with wavelength measured in units of meters.

14.1.2 Perceived Loudness and the Decibel

In order to judge the “loudness” of the sounds we hear, our ears continuously *average* the amplitude of the incoming sound signal over a short, sliding time window. This averaging effect is modeled well by a quantity known as the *effective sound pressure*. This is defined as the *root mean square* (RMS) of the instantaneous sound pressure measured over a specific interval of time.

If we were to take a series of n discrete sound pressure measurements p_i, equally spaced in time, the RMS sound pressure p_{rms} would be

$$p_{\text{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} p_i^2}. \tag{14.1}$$

However, our ears take pressure measurements continuously, rather than at discrete points in time. If we imagine measuring the instantaneous sound pressure continuously, starting at time T_1 and lasting until time T_2, the summation in Equation (14.1) would become an integral as follows:

$$p_{\text{rms}} = \sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2} (p(t))^2\, dt}. \tag{14.2}$$

然而，故事并没有就此结束。感知响度实际上与声强I成正比，而声强I本身又与RMS声压的平方成正比：

$$I \propto p_{\text{rms}}^2.$$

人类能够感知到非常广泛的声压变化——从纸张落地的飘动声到飞机突破一马赫的轰鸣声。为了管理如此宽的动态范围，我们通常以分贝 (dB) 为单位来测量声强。分贝是一个对数单位，表示两个值之间的比率。通过采用对数刻度，分贝可以用相对较窄的数值范围来表示很宽的测量范围。1 分贝实际上是贝尔的十分之一，贝尔是为了纪念亚历山大 · 格雷厄姆 · 贝尔而命名的单位。

当声强以分贝为单位时，称为声压级 (SPL)，用符号 L p 表示。声压级定义为声音的声强（即声压的平方）与参考强度 p ref 之比，参考强度 p ref 代表人类听觉的下限。因此，我们有：

$$\begin{aligned} L_p &= 10\log_{10}\left(\frac{p_{\text{rms}}^2}{p_{\text{ref}}^2}\right)\ \text{dB} \\ &= 20\log_{10}\left(\frac{p_{\text{rms}}}{p_{\text{ref}}}\right)\ \text{dB}, \end{aligned}$$

其中 20 的出现是因为当我们对数取平方时，它变成了乘以 2。空气中常用的参考声压为 p ref = 20 μ Pa (RMS)。更多关于声压、声音物理学和人类听觉感知的信息，请参阅 [6]。

顺便说一句，如果你感觉对数有点生疏，以下恒等式或许能帮助你复习一下。在公式 (14.3) 中，b、x 和 y 是正实数，且 b ⁄ = 1，c 和 d 是任意实数，c = log bx，d = log by（或者换句话说，bc = x，bd = y）。

$$\begin{aligned} \log_b x &= c && \text{when} && b^c = x \quad \text{(definition)}; \\ \log_b 1 &= 0 && \text{because} && b^0 = 1; \\ \log_b b &= 1 && \text{because} && b^1 = b; \\ \log_b(x\cdot y) &= \log_b x + \log_b y && \text{because} && b^c\cdot b^d = b^{c+d}; \\ \log_b(x/y) &= \log_b x - \log_b y && \text{because} && b^c/b^d = b^{c-d}; \\ \log_b x^d &= d\log_b x && \text{because} && (b^c)^d = b^{cd}. \end{aligned} \tag{14.3}$$

However, the story doesn't end here. Perceived loudness is actually proportional to the *acoustic intensity* I, which is itself proportional to the *square* of the RMS sound pressure:

$$I \propto p_{\text{rms}}^2.$$

Humans can perceive a very wide range of sound pressure variations—from the flutter of a piece of paper falling to the ground to the boom of an aircraft breaking mach one. In order to manage such a wide dynamic range, we normally measure sound intensity in units of *decibels* (dB). The decibel is a *logarithmic* unit that expresses the *ratio* between two values. By employing a logarithmic scale, the decibel allows a wide range of measurements to be represented by a relatively narrow range of values. A decibel is actually one-tenth of a *bel*, a unit named in honor of Alexander Graham Bell.

When sound intensity is measured in decibels, it is called *sound pressure level* (SPL) and represented by the symbol L_p. Sound pressure level is defined as the ratio of the acoustic intensity (i.e., the squared pressure) of a sound relative to a reference intensity p_{ref} that represents the lower limit of human hearing. So we have:

$$\begin{aligned} L_p &= 10\log_{10}\left(\frac{p_{\text{rms}}^2}{p_{\text{ref}}^2}\right)\ \text{dB} \\ &= 20\log_{10}\left(\frac{p_{\text{rms}}}{p_{\text{ref}}}\right)\ \text{dB}, \end{aligned}$$

where the 20 arises because when we take the square outside the logarithm, it becomes a multiplication by two. The commonly used reference sound pressure in air is $p_{\text{ref}} = 20\ \mu$Pa (RMS). For more information on sound pressure, the physics of sound and human aural perception, see [6].

By the way, if you're feeling a bit rusty, the following identities may help to refresh your memory on logarithms. In Equations (14.3), b, x and y are positive real numbers with $b \neq 1$, c and d are any real numbers, $c = \log_b x$, and $d = \log_b y$ (or written another way, $b^c = x$ and $b^d = y$).

$$\begin{aligned} \log_b x &= c && \text{when} && b^c = x \quad \text{(definition)}; \\ \log_b 1 &= 0 && \text{because} && b^0 = 1; \\ \log_b b &= 1 && \text{because} && b^1 = b; \\ \log_b(x\cdot y) &= \log_b x + \log_b y && \text{because} && b^c\cdot b^d = b^{c+d}; \\ \log_b(x/y) &= \log_b x - \log_b y && \text{because} && b^c/b^d = b^{c-d}; \\ \log_b x^d &= d\log_b x && \text{because} && (b^c)^d = b^{cd}. \end{aligned} \tag{14.3}$$

图 14.4. 人耳在 2 至 5 kHz 的频率范围内最为敏感。随着频率降低或超出此范围，需要越来越大的声压才能产生相同的“响度”感知。

14.1.2.1 等响度曲线

人耳对不同频率的声波的响应并不相同。人耳在2至5 kHz的频率范围内最敏感。随着频率降低或超出此范围，需要越来越大的声强（即声压）才能产生相同的“响度”感知。

图 14.4 显示了许多等响度曲线，每个曲线对应不同的感知响度级。这些曲线表明，要达到相同的感知响度，低频和高频所需的压力比中频所需压力更大。换句话说，如果我们保持声压波的振幅不变而改变频率，人耳实际上会感觉到低频和高频比中频“更弱”。最低的等响度曲线代表最安静的可听音调，也称为绝对听觉阈值。最高的等响度曲线穿过人类的痛阈，大约位于可听声音的 120 dB 水平。

有关等响度轮廓及其所基于的 Fletcher-Munson 曲线的更多信息，请参阅 https://bit.ly/2HfCjCs。

Figure 14.4. The human ear is most sensitive in the frequency range between 2 and 5 kHz. As the frequency decreases or increases beyond this range, more and more acoustic pressure is required to produce the same perception of “loudness.”

14.1.2.1 Equal-Loudness Contours

The human ear does not have the same response to sound waves of different frequencies. The human ear is most sensitive in the frequency range between 2 and 5 kHz. As the frequency decreases or increases beyond this range, more and more acoustic intensity (i.e., pressure) is required to produce the same perception of “loudness.”

Figure 14.4 shows a number of equal-loudness countours, each corresponding to a different perceived loudness level. These curves show that more pressure is required at low and high frequencies to achieve the same perceived loudness than is needed at mid-range frequencies. Or put another way, if we were to keep the amplitude of an acoustic pressure wave the same while varying the frequency, the human ear would actually perceive the lower and higher frequencies as “less loud” than the mid-range frequencies. The lowest equal-loudness contour represents the quietest audible tone and is also known as the absolute threshold of hearing. The highest contour passes through the human threshold of pain, which lies roughly at the 120 dB level for audible sounds.

For more information on equal-loudness contours, and the Fletcher-Munson curves on which they are based, see https://bit.ly/2HfCjCs.

14.1.2.2 可听频段

一个典型的成年人可以听到频率低至 20 Hz 到高至 20,000 Hz (20 kHz) 的声音（尽管上限通常会随着年龄的增长而降低）。等响度轮廓有助于解释为什么人耳只能感知这个有限“频带”内的声音。随着频率变低或变高，需要越来越大的声压才能产生相同的感知响度。当频率接近人耳听觉的下限或上限时，等响度轮廓会渐近垂直，这意味着我们需要一个实际上无限的声压才能产生任何响度感知。换句话说，在可听频带之外，人类对音频的感知会下降到几乎为零。

14.1.3 声波传播

与任何类型的波一样，声压波在空间中传播，可以被表面吸收或反射，在拐角处和狭窄的“缝隙”处发生衍射，并在穿过不同传输介质的边界时发生折射。声波不表现出极化 1，因为声压振荡发生在波传播方向上（这被称为纵波），而不是像光波那样垂直于波传播方向（横波）。在游戏中，我们通常会模拟虚拟声波的吸收、反射，有时还会模拟衍射（例如，在拐角处略微弯曲），但我们通常会忽略折射效应，因为这些效应不易被人类听众察觉。

14.1.3.1 随距离衰减

在一个空气完全静止的开放空间中，假设一个声源向各个方向均匀辐射，它产生的声压波的强度会随着距离的增加而下降，遵循 1/r2 定律，而压力则遵循 1/r 定律。

$$p(r) \propto \frac{1}{r};$$
$$I(r) \propto \frac{1}{r^2}.$$

这里，r 测量听众或麦克风与声源的径向距离，压力和强度都表示为 r 的函数。

1 固体中的声波可以是横向的，因此可以表现出极化。

14.1.2.2 The Audible Frequency Band

A typical adult can hear sounds with frequencies as low as 20 Hz and as high as 20,000 Hz (20 kHz) (although the upper limit generally decreases with age). The equal-loudness contours help to explain why the human ear can perceive sounds only within this limited “band” of frequencies. As the frequency becomes lower or higher, more and more acoustic pressure is required to produce the same perceived loudness. As the frequency approaches the lower or upper limits of human hearing, the countours become asympotically vertical, meaning we’d need an effectively infinite acoustic pressure to produce any perception of loudness at all. Or put another way, human audio perception drops off to effectively zero outside the audible frequency band.

14.1.3 Sound Wave Propagation

Like any kind of wave, an acoustic pressure wave propagates through space and can be *absorbed* or *reflected* by surfaces, *diffracted* around corners and through narrow “slits,” and *refracted* as it passes across the boundary between different transmission media. Sound waves exhibit no *polarization*[1] because the acoustic pressure oscillation occurs in the direction of wave travel (this is known as a *longitudinal* wave), rather than perpendicular to it as with light waves (a *transverse* wave). In games, we typically model the absorption, reflection and sometimes the diffraction (e.g., bending slightly around corners) of our virtual sound waves, but we generally ignore refraction effects because these effects are not easily noticed by a human listener.

14.1.3.1 Fall-Off with Distance

In an open space with otherwise perfectly still air, and assuming a sound source that radiates equally in all directions, the intensity of the sound pressure wave it produces falls off with distance, following a $1/r^2$ law, while pressure follows a $1/r$ law.

$$p(r) \propto \frac{1}{r};$$
$$I(r) \propto \frac{1}{r^2}.$$

Here, r measures the radial distance of the listener or microphone from the sound source, and both pressure and intensity are expressed as functions of r.

[1] Sound waves in solids can be transverse and therefore can exhibit polarization.

图 14.5. 三种类型的声源及其声辐射模式（为便于说明，采用二维表示）。从左到右：全向性、锥形和定向性。

更准确地说，开放空间中球形辐射（全向）声波的声压级可以写成如下形式：

$$L_p(r) = L_p(0) + 10\log_{10}\left(\frac{1}{4\pi r^2}\right)\ \text{dB}$$
$$= L_p(0) - 10\log_{10}\ \ 4\pi r^2\ \ \ \text{dB},$$

其中 L p (r) 是听者处的 SPL，是其与声源径向距离的函数，L p (0) 表示声源的未衰减或“自然”声强。

声源并非总是全向的。例如，当一面大而平坦的墙壁反射声波时，它就像一个纯定向声源——反射波沿单一方向传播，压力波前基本平行。

扩音器向特定方向投射声音，但具有锥形衰减，这意味着声波的强度沿着投射“锥体”的中心线最大，但随着听众与该中心线之间的角度增加而衰减。

图 14.5 显示了各种声音辐射模式。

14.1.3.2 大气吸收

声压会随着距离以 1/ r 的速率衰减，这是因为随着波形几何级数扩展，能量会耗散。这种衰减对所有频率的声音的影响相同。由于大气吸收能量，声强也会随着距离的增加而衰减。大气吸收效应在整个频谱范围内并不均匀。一般而言，吸收效应会随着声音频率的增加而增强。

这让我想起高中时听过的一个故事：夜晚，一位女士走在一条安静的乡村街道上。她听到一连串断断续续的

Figure 14.5. Three types of sound sources and their sound radiation patterns (in two dimensions for ease of illustration). From left to right: omnidirectional, conical and directional.

More precisely, the sound pressure level for a spherically radiating (omnidirectional) sound wave in open space can be written as follows:

$$L_p(r) = L_p(0) + 10\log_{10}\left(\frac{1}{4\pi r^2}\right)\ \text{dB}$$
$$= L_p(0) - 10\log_{10}\left(4\pi r^2\right)\ \text{dB},$$

where $L_p(r)$ is the SPL at the listener as a function of its radial distance from the sound source, and $L_p(0)$ represents the unattenuated or “natural” sound intensity of the source.

Sound sources are not always omnidirectional. For example, when a large, flat wall reflects sound waves, it acts like a purely *directional* sound source—the reflected waves propagate in a single direction, and the pressure wavefronts are essentially parallel.

A bullhorn projects sound in a particular direction but with a *conical* fall-off, meaning that the intensity of the sound waves is maximum along the centerline of the projection “cone,” but falls off as the angle between the listener and this centerline increases.

Various sound radiation patterns are illustrated in Figure 14.5.

14.1.3.2 Atmospheric Absorption

The $1/r$ fall-off of sound pressure with distance arises because energy is dissipated as the waveform expands geometrically. This fall-off affects sounds of all frequencies equally. Sound intensity also falls off with distance due to energy absorption by the atmosphere. Atmospheric absorption effects are *not* uniform across the entire frequency spectrum. In general, the absorption effect becomes greater as the frequency of the sound increases.

I’m reminded of a story I heard when I was in high school: A woman was walking down a quiet village street at night. She heard a sporadic sequence of

低沉的音调，音符之间有着长长的、无声的间隙。她很好奇是什么发出了这些奇怪的音调，于是朝它们走去。随着她走动，音调变得越来越大，音调之间的间隔似乎也越来越短。走了几分钟后，这些音调汇成了一首美妙的乐曲。女人来到一扇敞开的窗户前，发现一位中提琴手正在里面练习。乐手停下演奏说“你好”，女人问他为什么几分钟前一直在胡乱拉琴。他回答说：“我没有胡乱拉琴——我一直在拉这首曲子。”当然，女人听到的声音的解释是，由于大气吸收，低频声音比高频声音能传得更远。你可以在http://www.sfu.ca/sonic-studio/handbook/Sound_Propagation.html上了解更多关于声波传播的知识。

其他因素也会影响声波在介质中传播时的强度。一般来说，衰减取决于距离、频率、温度和湿度。请访问 http://sengpielaudio.com/calculator-air.htm，这是一个在线计算器，可以让你试验这些因素的影响。

14.1.3.3 相移和干扰

当多个声波在空间中重叠时，它们的振幅会叠加——这被称为叠加。考虑两个频率相同的周期性声波。（最简单的例子是两个正弦波。）如果这两个波同相——即它们的波峰和波谷对齐——那么这两个波将相互增强，最终产生的波的振幅比任何一个原始波都要大。同样，如果这两个波异相，一个波的波峰可能会抵消另一个波的波谷，反之亦然，最终产生的波的振幅较低（甚至为零）。

当多个波相互作用时，我们称之为干涉。相长干涉描述的是波之间相互加强，振幅增大的情况。相消干涉描述的是波之间相互抵消，导致振幅降低的情况。

波的频率对这一现象有重要影响：
如果两束波的频率非常接近，干涉只会增加或减少整体振幅。如果频率差异很大，就会产生一种称为“拍频”的效应，即频率差异导致两束波的同相和异相周期交替出现，从而产生振幅较高和较低的周期交替出现。

干扰可能发生在两个完全不相关的声音信号之间，也可能发生在单个声音信号从源到接收端经过多条路径时。

low tones with long, silent gaps between them. Curious what might be making these strange tones, she walked toward them. As she walked, the tones became louder and the spaces between the tones seemed to get shorter. After a few minutes' walk, the tones had resolved into a beautiful piece of music. The woman arrived at an open window to discover a viola player practicing within. The musician stopped playing to say "hello," and the woman asked him why he had been playing random notes a few minutes before. He replied, "I haven't been playing random notes—I've been playing this piece the whole time." The explanation for what the woman heard, of course, is that lower-frequency sounds can be heard over longer distances than higher-frequency sounds due to atmospheric absorption. You can learn more about sound wave propagation at http://www.sfu.ca/sonic-studio/handbook/Sound_Propagation.html.

Other factors also affect the intensity of sound waves as they propagate through their medium. In general, fall-off depends on distance, frequency, temperature and humidity. See http://sengpielaudio.com/calculator-air.htm for an online calculator that lets you experiment with the effects of these factors.

14.1.3.3 Phase Shift and Interference

When multiple sound waves overlap in space, their amplitudes add together—this is called *superposition*. Consider two periodic sound waves with the same frequency. (The simplest example would be two sinusoids.) If the waves are *in phase*—that is, their peaks and troughs line up—then the waves will *positively reinforce* each other, and the result is a wave with larger amplitude than either of the original waves. Likewise, if the waves are *out of phase*, the peaks of one wave can tend to *cancel* the troughs of the other and vice versa, and the result is a wave with lower (or even zero) amplitude.

When multiple waves interact, we call this *interference*. *Constructive interference* describes the case in which the waves reinforce one another and the amplitude increases. *Destructive interference* occurs when the waves cancel each other out, resulting in lower amplitude.

The frequency of the waves has an important effect on this phenomenon: If the frequencies of the two waves match closely, the interference simply increases or decreases the overall amplitude. If the frequencies differ significantly, we can get an effect called *beating*, wherein the frequency difference causes alternating periods of the waves being in and out of phase, resulting in alternating periods of higher and lower amplitude.

Interference can occur between two totally unrelated sound signals, or it can occur if a single sound signal takes multiple paths from the source to the

听众。在后一种情况下，路径长度的差异会引入相移，这可能会导致建设性或破坏性干扰，具体取决于相移的量。

梳状滤波

干扰会导致一种称为梳状滤波的效应。这是由于声波在表面反射时，会几乎完全抵消或完全增强某些频率而引起的。其结果是频率响应（参见第 14.2.5.7 节）中会出现许多窄峰和窄谷，绘制出来看起来有点像梳子（因此得名）。这种效应会对音频重现和录制产生很大的影响——有时它是一种不受欢迎的伪像，有时却被当作一种工具。梳状滤波的存在也是为什么花钱进行房间声学处理通常比花钱购买高端音响设备更划算的关键原因之一：如果房间出现梳状效应，那么您试图从设备中获得平坦响应就是在浪费时间。请访问 http://www.realtraps.com/video_comb.htm，观看 Ethan Winer 制作的精彩视频。

14.1.3.4 混响和回声

在任何含有声音反射表面的环境中，听众通常会从声源接收到三种声波：

• 直达（干）声。从声源通过直接、畅通的路径到达听众的声波统称为直达声或干声。

• 早期反射（回声）。声波通过间接路径到达听者，在被周围表面反射并被部分吸收后，由于路径较长，需要更长的时间才能到达听者。因此，直达声波和反射波之间会有延迟。到达耳朵的第一组反射声波仅与一两个表面相互作用。因此，它们是相对“干净”的信号，我们将它们感知为声音或回声的独特新“副本”。

• 后期混响（尾声）。声波在聆听空间中反射多次后，会相互叠加和干扰，以至于大脑无法再分辨出清晰的回声。这被称为后期混响或弥散尾声。反射面的特性会导致波的振幅衰减不同程度。由于反射声波

listener. In the latter case, the difference in path lengths introduces a phase shift that can cause either constructive or destructive interference, depending on the amount of the phase shift.

Comb Filtering

Interference can lead to an effect known as *comb filtering*. This is caused when sound waves reflect off surfaces in such a way as to either almost completely cancel or completely reinforce certain frequencies. The result is a *frequency response* (see Section 14.2.5.7) with lots of narrow peaks and troughs, which when plotted look a bit like a comb (hence the name). This effect can have a big impact on audio reproduction and recording—sometimes it is an undesirable artifact, and sometimes it is used as a tool. The existence of comb filtering is also one of the key reasons why it is generally better to spend money on acoustic room treatment than to spend money on high-end audio equipment: If the room exhibits comb effects, you're wasting your time trying to get a flat response from your gear. See http://www.realtraps.com/video_comb.htm for a great video by Ethan Winer on the subject.

14.1.3.4 Reverb and Echo

In any environment containing sound-reflective surfaces, a listener generally receives three kinds of sound waves from a sound source:

- *Direct (dry)*. Sound waves that arrive at the listener via a direct, unobstructed path from the source are collectively known as *direct* or *dry* sound.
- *Early reflections (echo)*. Sound waves that arrive at the listener via an indirect path, after being reflected from and partially absorbed by surrounding surfaces, take a longer time to reach the listener because their path is longer. As such, there will be a *delay* between the arrival of the direct sound waves and the arrival of the reflected waves. The first group of reflected sound waves arriving at the ear have only interacted with one or two surfaces. As such, they are relatively "clean" signals, and we perceive them as distinct new "copies" of the sound or *echos*.
- *Late reverberations (tail)*. Once the sound waves have bounced around the listening space more than a few times, they superimpose and interfere with one another so much that the brain can no longer detect distinct echos. These are known as *late reverberations* or the *diffuse tail*. The properties of the reflective surfaces cause the amplitudes of the waves to be attenuated by varying amounts. And because the reflected sound waves

图 14.6. 直达声波、早期反射和晚期混响。

当声波被延迟时，相移就会发生，导致声波相互干扰。这会导致某些频率相对于其他频率衰减。当我们谈论一个空间的声学效果时，我们主要谈论的是后期混响对声音感知到的“质量”或“音色”的影响。

回声、尾音与干声结合在一起，形成了所谓的湿声。图 14.6 展示了一次突然的拍手声的干湿成分。

早期反射和晚期混响为大脑提供了丰富的线索，让我们可以了解自己所处空间的类型。预延迟是直达声波到达和第一个反射波到达之间的时间间隔。通过预延迟，大脑可以判断出我们所在房间或空间的大致大小。衰减是反射声波消失所需的时间。这可以告诉我们的大脑周围环境吸收了多少声音，从而间接地告诉我们一些有关构成我们所在空间的材料的信息。例如，一间铺有瓷砖的小浴室会产生后期混响，其预延迟非常短（由于其尺寸小），衰减很长（由于瓷砖能够有效反射声波，吸收很少）。像纽约中央车站（又名大中央车站）这样的大型花岗岩墙壁房间会有更长的预延迟和更多的回声，但衰减将与瓷砖浴室的衰减相似。

如果我们在浴室里挂上窗帘，或者墙壁上铺的是木板而不是瓷砖，那么预延迟将保持不变，但

Figure 14.6. Direct sound waves, early reflections and late reverberations.

are delayed, phase shifts occur causing the waves to interfere with one another. This causes certain frequencies to be attenuated relative to the others. When we speak of the *acoustics* of a space, we are largely speaking about the effects of late reverberations on the perceived “quality” or “timbre” of the sound.

Collectively, the echos and the tail combine with the dry sound to create what is known as *wet* sound. Figure 14.6 illustrates the wet and dry components of a single abrupt clap.

The early reflections and late reverberations provide the brain with a wealth of cues that tell us quite a lot about the type of space we’re in. The *pre-delay* is the time interval between the arrival of the direct sound waves and the arrival of the very first reflected waves. From the pre-delay, the brain can determine the approximate size of the room or space in which we are listening. The *decay* is the time it takes for the reflected sound waves to die away. This tells our brains how much of the sound has been absorbed by the surroundings, and so indirectly tells us something about the materials that make up the space we’re in. For example, a small tiled bathroom would produce late reverberations with a very short pre-delay (due to its small size) and a long decay (due to the tile’s ability to reflect sound waves efficiently, with little absorption). A large granite-walled room like Grand Central Terminal (a.k.a. Grand Central Station) in New York City will have a much longer pre-delay and a lot more echos, but the decay will be similar to that of the tiled bathroom.

If we were to hang curtains in that bathroom, or if the walls were covered with wood panels instead of tile, the pre-delay would remain the same, but

衰减以及其他因素，例如密度（单个反射在时间上的间隔有多近）和扩散（反射密度随时间增加的速率），都会发生变化。这解释了为什么一个人即使蒙着眼睛也能猜出自己身在何处，或者盲人如何仅靠拐杖就能学会导航。声音为我们提供了大量关于周围环境的信息！

混响（reverb）一词用于描述声音中湿成分的质量。在录音的早期，音响工程师几乎无法控制混响，完全依赖于录音房间的形状和结构。后来，简单的人工混响设备被发明出来，从比尔·普特曼（Universal Audio 创始人）在浴室里使用扬声器和麦克风，到使用长金属板或弹簧在声音信号中引入延迟，再到现代数字技术。如今，数字信号处理器 (DSP) 芯片和/或软件不仅用于在录制的音效和音乐中重现自然混响效果，还用于为录音增添各种自然界中通常听不到的有趣效果。我们将在 14.2 节中学习更多关于数字信号处理的知识。您可以在 http://www.uaudio.com/blog/the-basics-of-reverb 上阅读更多关于混响的内容。

消声室是专门设计用来完全消除反射声波的房间。其原理是，在房间的墙壁、地板和天花板上铺上厚厚的波纹泡沫垫，吸收几乎所有反射声波。这样，只有直达声（干声）才能到达听众或麦克风。消声室中的声音音色完全“死寂”。消声室适合录制不含混响的“纯净”声音。这种纯净的声音通常非常适合输入到数字信号处理流程中，为声音设计师提供最大的灵活性来控制声音的音色。

14.1.3.5 运动中的声音：多普勒效应

如果你曾经站在铁路道口，看着火车经过，你就会听到多普勒效应。当火车靠近你时，声音听起来音调较高，而当火车驶向远方时，声音听起来音调较低。声波在空气中以大致恒定的速度传播，但声源（在本例中是火车）也在移动。与火车同向传播的声波会被“挤压”在一起，而与火车运动方向相反的声波则会“扩散”，每个声波的扩散程度与空气中的声速和火车在空气中的速度之差成正比。因此，被挤压的声波的频率会增加，因为波峰和波谷之间的距离

the decay, along with other factors such as *density* (how closely spaced in time the individual reflections are) and *diffusion* (the rate at which the reflections increase in density over time), would change. This explains how a person can guess where they are even when blindfolded, or how the blind can learn to navigate with only a cane to aid them. Sound provides us with a *lot* of information about our surroundings!

The term *reverb* is used to describe the quality of a sound in terms of its wet components. In the early days of audio recording, sound engineers had little control over reverb, relying entirely on the shape and construction of the room in which the recording was made. Later, simple artificial reverb devices were created, from the use of a speaker and microphone in a bathroom by Bill Putman Sr. (founder of Universal Audio), to the use of a long metal plate or spring to introduce a delay in a sound signal, to modern digital techniques. Today, digital signal processor (DSP) chips and/or software are used not only to recreate natural reverb effects in recorded sound effects and music, but also to augment recordings with all sorts of interesting effects that are not normally heard in nature. We'll learn more about digital signal processing in Section 14.2. You can read more about reverb at http://www.uaudio.com/blog/the-basics-of-reverb.

An *anechoic chamber* is a room especially designed to entirely eliminate reflected sound waves. This is accomplished by lining the walls, floor and ceiling of the room with thick corrugated foam padding that absorbs essentially all of the reflected sound waves. As a result, only the direct (dry) sound reaches the listener or microphone. Sound in an anechoic chamber has a completely "dead" timbre. Anechoic chambers are useful for recording "pure" sounds that contain no reverb. Such pure sounds are often perfect candidates for input into a digital signal processing pipeline, giving a sound designer maximum flexibility to control the timbre of the sound.

14.1.3.5 Sound in Motion: The Doppler Effect

If you've ever stood at a railway crossing when a train goes by, you've heard the *Doppler effect* in action. The sound of the train seems higher pitched when it is approaching you, and becomes lower pitched as it races off into the distance. Sound waves travel at a roughly constant speed through the air, but the sound source (in this case, the train) is also moving. The sound waves that are moving in the same direction as the train become "squashed together," and the waves that are moving opposite to the motion of the train become "spread out," each by an amount proportional to the difference between the speed of sound in air and the speed of the train through the air. The frequency of the squashed waves is therefore increased, because the space between the peaks and troughs

声波的振动频率被有效降低，导致声音音调更高。同样，扩散波的频率也会降低，导致声音音调更低。多普勒效应以奥地利物理学家克里斯蒂安·多普勒的名字命名，他于1842年发现了这一现象。

当听者移动而声源静止时，也会出现多普勒效应。一般来说，多普勒频移取决于听者与声源之间的相对速度（以矢量表示）。在一个维度上，多普勒频移相当于频率的变化，可以量化如下：

$$f' = \left(\frac{c + v_l}{c + v_s}\right) f,$$

其中，f 为原始频率，f′ 为听者处的多普勒频移（观测到的频率），c 为空气中的声速，vl 和 vs 分别为听者的速度和声源的速度。如果声源的速度相对于声速非常小，我们可以用以下公式近似地计算出两者之间的关系：

$$\begin{aligned} f' &= \left(\frac{1 + (v_l - v_s)}{c}\right) f \\ &= \left(\frac{1 + \Delta v}{c}\right) f. \end{aligned}$$

该表达式使相对速度Δv显而易见。通过查看以下GIF动画，可以轻松直观地了解多普勒效应：http://en.
wikipedia.org/wiki/File:Dopplereffectsourcemovingrightatmach0.7.gif。

14.1.4 位置感知

人类的听觉系统已经进化到能够相当准确地感知声音在周围空间中的位置。许多因素影响着我们对声音位置的感知：

- 距离衰减可以让我们大致了解声源的距离。为了做到这一点，我们必须对近距离听到的声音的响度有所了解，以此作为“基准”。

- 大气吸收会导致声源远离听众时，声音中较高的频率会衰减。这可以作为一个重要的线索，帮助我们感知远处正常音量说话的人和近处降低音量说话的人之间的差异。

of the sound waves has been effectively reduced, resulting in a higher-pitched sound. Likewise, the frequency of the spread-out waves is decreased, resulting in a lower-pitched sound. The Doppler effect was named after the Austrian physicist Christian Doppler, who identified it in 1842.

The Doppler effect also occurs when the listener is moving but the sound source is stationary. In general, the Doppler shift is dependent upon the *relative velocity* (as a vector) between the listener and the sound source. In one dimension, the Doppler shift amounts to a change in frequency, and can be quantified as follows:

$$f' = \left(\frac{c + v_l}{c + v_s}\right) f,$$

where f is the original frequency, f' is the Doppler-shifted (observed) frequency at the listener, c is the speed of sound in air and v_l and v_s are the speeds of the listener and sound source, respectively. If the speeds of the sound sources are very small relative to the speed of sound, we can approximate this relationship as follows:

$$\begin{aligned} f' &= \left(\frac{1 + (v_l - v_s)}{c}\right) f \\ &= \left(\frac{1 + \Delta v}{c}\right) f. \end{aligned}$$

This expression makes the relative velocity, Δv, apparent. The Doppler effect can be easily visualized by looking at the following animated GIF: http://en.
wikipedia.org/wiki/File:Dopplereffectsourcemovingrightatmach0.7.gif.

14.1.4 Perception of Position

The human auditory system has evolved to allow a reasonably accurate perception of the position of sounds in the space around us. A number of factors contribute to our perception of sound position:

- *Fall-off with distance* provides us with a rough idea of how far away the source of a sound is. In order for this to work, we must have some idea of the loudness of the sound when heard at close range to serve as a “baseline.”
- *Atmospheric absorption* causes the higher frequencies in a sound to drop out as the source moves farther away from the listener. This can serve as an important cue in perceiving the difference between, for example, a person speaking at normal volume but far away and a person speaking at reduced volume up close.

• 拥有左右两只耳朵，这赋予我们大量的位置信息。右侧的声音在右耳听起来会比在左耳更响亮。此外，由于头部一侧的声音到达另一侧耳朵所需的时间略长，因此会产生大约一毫秒的耳间时间差 (ITD)。最后，头部本身会阻挡声音，因此靠近声源一侧的耳朵会听到比靠近近侧耳朵略微低沉的声音。这被称为耳间强度差 (IID)。

• 耳朵的形状也有影响。我们的耳朵略微向前倾斜，所以来自我们身后的声音相对于来自我们前方的声音会稍微减弱一些。

• 头部相关传递函数 (HRTF) 是耳朵褶皱（耳廓）对来自不同方向的声音的微小影响的数学模型。

14.2 声音的数学

信号处理与系统理论是数学领域的核心，几乎是所有现代音频技术的核心。它还广泛应用于各种其他技术和工程领域，包括图像处理和机器视觉、航空学、电子学、流体动力学等等。本节将快速回顾信号与系统理论的关键概念，因为这将有助于我们理解本章后面一些更高级的游戏音频主题。（它也是数学理论中一个重要的领域，可以让任何游戏程序员受益——所以，这又有什么关系呢！）关于该主题的深入探讨，请参阅[41]。

14.2.1 信号

信号是一个或多个独立变量的函数，通常描述某种物理现象的行为。在14.1节中，我们使用信号p(t)来表示音频压缩波随时间变化的声压。当然，还有很多其他类型的信号。信号v(t)可以表示麦克风随时间产生的电压，而w(t)可以模拟管道系统中随时间变化的水压，或者我们可以使用f(t)来表示生态系统中狐狸种群的变化。

在研究信号理论时，我们经常将独立变量称为“时间”，并用符号 t 表示——当然，独立变量

- *Having two ears*, one on the left and one on the right, gives us a great deal of positional information. A sound that is to our right will sound louder in the right ear than in the left. An *interaural time difference* (ITD) of approximately one millisecond also arises, because a sound to one side of your head will take just a little bit longer to reach the opposite ear. Finally, the head itself obstructs sounds, so the ear opposite to the sound source will perceive a slightly muffled version of the sound reaching the near ear. This is known as interaural intensity difference (IID).
- *Ear shape* has an effect as well. Our ears are cupped slightly forward, so sounds coming from behind us are very slightly muffled relative to those coming from in front of us.
- *The head-related transfer function* (HRTF) is a mathematical model of the minute effects that the folds of our ears (the *pinnae*) have on sounds coming from different directions.

14.2 The Mathematics of Sound

Signal processing and systems theory is the area of mathematics that lies at the heart of virtually all modern audio technology. It is also extensively used in a wide variety of other technological and engineering endeavors, including image processing and machine vision, aeronautics, electronics, fluid dynamics, and the list goes on. In this section, we'll embark on a whirlwind tour of the key concepts in signals and systems theory, because it will help us to understand some of the more advanced topics in game audio later in the chapter. (It's also an important area of mathematical theory that can benefit any game programmer—so what the heck!) An in-depth treatment of the topic can be obtained from [41].

14.2.1 Signals

A *signal* is any function of one or more independent variables, typically describing the behavior of some kind of physical phenomenon. In Section 14.1, we used the signal $p(t)$ to represent the time-varying acoustic pressure of an audio compression wave. Of course, many other kinds of signals are possible. A signal $v(t)$ might represent the voltage produced by a microphone over time, while $w(t)$ might model the time-varying water pressure in a system of pipes, or we could use $f(t)$ to represent the varying population of foxes in an ecosystem.

In studying signal theory, we often refer to the independent variable as “time” and represent it with the symbol t—but of course the independent vari-

可以表示其他量，并且可能有多个独立变量。例如，可以将二维灰度图像视为信号 i(x,y)，其中两个独立变量 x 和 y 代表正交坐标轴，信号值 i 代表灰度图像每个像素的强度。彩色图像也可以类似地用三个信号表示：r(x,y) 代表红色通道，g(x,y) 代表绿色通道，b(x,y) 代表蓝色通道。

14.2.1.1 保持离散，持续

上面的二维图像示例揭示了两种基本信号之间的重要区别：连续信号和离散信号。

- 如果自变量是实数（t∈R），我们称该信号为连续时间信号。在本章中，我们将使用符号t来表示连续的“时间”，并使用圆括号表示函数符号（例如x(t)），以提醒我们处理的是连续时间信号。

- 如果自变量是整数（n∈I），我们称该信号为离散时间信号。我们使用符号n表示离散“时间”，并使用方括号表示函数符号（例如x[n]），以提醒我们处理的是离散时间信号。请注意，离散时间信号的值可能仍然是实数（x[n]∈R）——“离散时间信号”一词唯一规定的是自变量是整数（n∈I）。

在图 14.1 中，我们看到可以将连续时间信号可视化为普通函数图，横轴为时间 t，纵轴为信号值 p(t)。我们可以用类似的方式绘制离散时间信号 x[n]，尽管该函数的值仅针对自变量 n 的整数值定义（参见图 14.7）。一种常见的理解离散时间信号的方式是将其视为连续时间信号的采样版本。采样过程（也称为数字化或模数转换）是数字音频录制和播放的核心。有关采样的更多信息，请参见第 14.3.2.1 节。

14.2.2 操作信号

在接下来的讨论中，理解一些通过改变信号自变量来操纵信号的基本方法至关重要。例如，为了反映 t = 0 时的信号，我们只需在信号方程中将 t 替换为 − t。要将整个信号向右时移（即，在

able might represent some other quantity, and there may be more than one independent variable. For example, one can think of a 2D greyscale image as a signal $i(x, y)$, where the two independent variables, x and y, represent the orthogonal coordinate axes, and the signal value i represents the intensity of the greyscale image at each pixel. A color image could be similarly represented by three signals, $r(x, y)$ for the red channel, $g(x, y)$ for the green channel and $b(x, y)$ for the blue channel.

14.2.1.1 Be Discrete, Continuously

The 2D image examples above bring to light an important distinction between two fundamental kinds of signal: *continuous* and *discrete*.

- If the independent variable is a *real number* ($t \in \mathbb{R}$), we call the signal a *continuous-time signal*. In this chapter, we'll use the symbol t to represent continuous "time," and we'll use round parentheses for our functional notation (e.g., $x(t)$) to remind us that we're working with a continuous-time signal.
- If the independent variable(s) is an *integer* ($n \in \mathbb{I}$), we call the signal a *discrete-time signal*. We'll use the symbol n to represent discrete "time," and we'll use square brackets for our functional notation (e.g., $x[n]$) to remind us that we're working with a discrete-time signal. Note that the *value* of a discrete-time signal might still be a real number ($x[n] \in \mathbb{R}$)—the only thing the term "discrete-time signal" prescribes is that the *independent* variable is an integer ($n \in \mathbb{I}$).

In Figure 14.1, we saw that we can visualize a continuous-time signal as an ordinary function plot, with time t on the horizontal axis and the signal value $p(t)$ on the vertical axis. We can plot a discrete-time signal $x[n]$ in similar fashion, although the function's values are only defined for integer values of the independent variable n (see Figure 14.7). One common way to think of a discrete-time signal is as a *sampled* version of a continuous-time signal. The sampling process (also known as *digitization* or *analog-to-digital conversion*) lies at the heart of digital audio recording and playback. See Section 14.3.2.1 for more information on sampling.

14.2.2 Manipulating Signals

It will be important in the following discussions for us to understand some basic ways to manipulate a signal by making changes to its independent variable. For example, to *reflect* a signal about $t = 0$, we simply replace t with $-t$ in the signal's equation. To *time-shift* the entire signal to the *right* (i.e., in the

为了使时间向左/负方向移动 s ，我们在信号方程中用 t − s 代替 t 。（时间向左/负方向移动是通过将 t 代替 t + s 来实现的。）我们还可以通过缩放自变量来扩展或压缩信号的域。这些简单的变换如图 14.8 所示。

14.2.3 线性时不变（LTI）系统

在信号处理理论中，系统被定义为将输入信号转换为新输出信号的任何设备或过程。系统的数学概念可用于描述、分析和操作音频处理中出现的许多实际系统，包括麦克风、扬声器、模数转换器、混响单元、均衡器和滤波器，甚至房间的声学效果。

举一个简单的例子，放大器是一个将输入信号幅度增加A倍（称为放大器增益）的系统。给定输入信号x(t)，这样的放大系统将产生一个输出信号

$y(t) = Ax(t)$.
时不变系统是指输入信号的时间偏移会导致输出信号发生相同时间偏移的系统。换句话说，系统的行为不会随时间而改变。

线性系统具有叠加特性。这意味着，如果输入信号由其他信号的加权和组成，则输出将是各个输出的加权和，这些输出是当其他每个信号独立地输入系统时产生的。

线性时不变 (LTI) 系统非常有用，原因有二。

图 14.7. 离散时间信号 x [n] 的值仅针对 n 的整数值进行定义。

positive direction) by a distance s, we replace t with $t - s$ in the signal's equation. (Time shifting to the *left*/*negative* direction is accomplished by replacing t with $t + s$.) We can also expand or compress the domain of the signal by scaling the independent variable. These simple transformations are illustrated in Figure 14.8.

14.2.3 Linear Time-Invariant (LTI) Systems

In the context of signal processing theory, a *system* is defined as any device or process that transforms an *input* signal into a new *output* signal. The mathematical concept of a system can be used to describe, analyze and manipulate many real-world systems that arise in audio processing, including microphones, speakers, analog-to-digital converters, reverb units, equalizers and filters and even the acoustics of a room.

As a simple example, an *amplifier* is a system that increases the amplitude of its input signal by a factor A known as the *gain* of the amp. Given an input signal $x(t)$, such an amplification system would produce an output signal $y(t) = Ax(t)$.

A *time-invariant* system is one for which a time shift in the input signal causes an equal time shift in the output signal. In other words, the behavior of the system does not change over time.

A *linear* system is one that possesses the property of *superposition*. This means that if an input signal consists of a *weighted sum* of other signals, then the output is a weighted sum of the individual outputs that would have been produced, had each of the other signals been fed through the system independently.

Linear time-invariant (LTI) systems are extremely useful for two reasons.

Figure 14.7. The value of a discrete-time signal $x[n]$ is defined only for integer values of n.

图 14.8. 信号独立变量的简单操作。

首先，它们的行为很容易理解，而且数学上也相对容易处理。其次，音频传播理论、电子学、力学、流体流动等领域的许多实际物理系统都可以用 LTI 系统精确建模。因此，为了理解音频技术，我们将只讨论 LTI 系统。

我们可以将任何系统视为一个具有输入信号和输出信号的黑匣子，如图 14.9 所示。

图 14.9。系统就像一个黑匣子。

使用这种黑盒符号，可以方便地将简单的系统互连，构建更复杂的系统。例如：

- 系统 A 的输出可以连接到系统 B 的输入，从而产生一个先执行操作 A 再执行操作 B 的复合系统。这称为串联。
- 两个系统的输出可以加在一起。
- 系统的输出可以反馈到早期的输入中，产生所谓的反馈回路。

有关所有这些类型的连接的示例，请参见图 14.10。

所有 LTI 系统的一个非常重要的特性是它们的互连与顺序无关。因此，如果我们有一个系统 A 和系统 B 的串行连接，我们可以反转这两个系统的顺序，而输出将保持不变。

Figure 14.8. Simple manipulations of a signal's independent variable.

First, their behaviors are well understood and relatively easy to work with mathematically. Second, many real physical systems in the fields of audio propagation theory, electronics, mechanics, fluid flow, etc. can be modeled accurately using LTI systems. As such, we will restrict ourselves to a discussion of LTI systems for our purposes of understanding audio technology.

We can visualize any system as a black box with an input signal and an output signal, as shown in Figure 14.9.

Figure 14.9. A system as a black box.

Using this black-box notation, simple systems can be conveniently interconnected to construct more complex systems. For example:

- The output of system A could be connected to the input of system B, yielding a composite system that performs operation A followed by operation B. This is called a *serial connection*.
- The outputs of two systems could be added together.
- The output of a system could be *fed back* into an earlier input, yielding what is known as a *feedback loop*.

See Figure 14.10 for examples of all of these kinds of connections.

One very important property of all LTI systems is that their interconnections are *order-independent*. So if we have a serial connection of system A followed by system B, we can reverse the order of the two systems and the output will remain unchanged.

14.2.4 LTI 系统的脉冲响应

谈论将输入信号转换为输出信号的系统固然很好，绘制系统互连图也相当直观。但是，我们如何用数学方法描述系统的运行呢？

回想一下 14.2.3 节，对于线性系统，如果输入由输入信号的线性组合（加权和）组成，则输出将是各个输出的线性组合（加权和）（假设每个输入信号都独立输入系统）。因此，如果我们能找到一种方法，将任意输入信号表示为非常简单的信号的加权和，那么我们应该能够仅通过描述系统对这些非常简单信号的响应来描述系统的行为。

14.2.4.1 单位冲量

如果我们将输入信号描述为简单信号的线性组合，那么问题就来了：我们应该使用哪个简单信号？出于稍后会解释的原因，我们选择的信号是单位脉冲。该信号属于一类被称为奇异函数的相关函数，因为它们都包含至少一个不连续点或“奇点”。

在离散时间中，单位脉冲 δ [n] 非常简单：它是一个信号，除了在 n = 0 处，它的值在任何地方都为零，在 n = 0 处它的值为 1：

$$\delta[n] = \begin{cases} 1 & \text{if } n = 0, \\ 0 & \text{otherwise.} \end{cases}$$

离散时间单位脉冲如图 14.11 所示。

图 14.10。系统互连的各种方法。在串联连接中，y(t) = B(A(x(t)))。在并联连接中，y(t) = αA(x(t)) + bB(x(t))。在反馈回路中，y(t) = x(t) − α · y(t)。

14.2.4 Impulse Response of an LTI System

It's all fine and dandy to talk about systems that convert an input signal into an output signal, and it's even pretty intuitive to draw diagrams of system interconnections. But how can we describe the operation of a system mathematically?

Recall from Section 14.2.3 that for a *linear* system, if the input consists of a linear combination (weighted sum) of input signals, the output will be a linear combination (weighted sum) of the individual outputs (had each of the input signals been fed into the system independently). So, if we can figure out a way to represent an arbitrary input signal as a weighted sum of very simple signals, we should be able to describe the behavior of the system by describing *only* its response to those very simple signals.

14.2.4.1 The Unit Impulse

If we are going to describe an input signal as a linear combination of simple signals, the question arises: Which simple signal shall we use? For reasons that will become clear in a moment, our signal of choice is going to be the *unit impulse*. This signal is one of a family of related functions known as *singularity functions* because they all contain at least one discontinuity or "singularity."

In discrete time, the unit impulse $\delta[n]$ is as simple as it gets: It is a signal whose value is zero everywhere except at $n = 0$, where its value is one:

$$\delta[n] = \begin{cases} 1 & \text{if } n = 0, \\ 0 & \text{otherwise.} \end{cases}$$

The discrete-time unit impulse is illustrated in Figure 14.11.

Figure 14.10. Various ways to interconnect systems. In the serial connection, $y(t) = B(A(x(t)))$. In the parallel connection, $y(t) = aA(x(t)) + bB(x(t))$. In the feedback loop, $y(t) = x(t) - a\dot{y}(t)$.

在连续时间中，单位冲量 δ (t) 的定义稍微复杂一些。它是一个函数，除了 t = 0 时，其值在任何地方都为零，此时其值为无穷大——但曲线下的面积等于 1。

为了了解如何正式定义这样一个怪异的函数，想象一个“盒子”函数 b(t)，它的值在区间 [0,T) 之外的任何地方都为零，在该区间它的值为 1/T。该曲线下的面积就是盒子的面积，即宽度乘以高度，即 T × 1 T = 1。现在想象极限为 T → 0。此时，盒子的宽度趋近于零，高度趋近于无穷大，但其面积仍然等于 1 。如图 14.12 所示。

单位脉冲函数通常用符号δ(t)表示。它的正式定义如下：

$$\delta(t) = \lim_{T \to 0} b(t),$$

在哪里

$$b(t) = \begin{cases} 1/T & \text{if } t \geq 0 \text{ and } t < T, \\ 0 & \text{otherwise.} \end{cases}$$

如图 14.13 所示，我们通常通过绘制一个箭头来绘制单位冲量，该箭头的高度代表曲线下的面积（因为函数在 t = 0 时的实际“高度”是无限的）。

14.2.4.2 使用脉冲序列表示信号

现在我们知道了单位脉冲信号是什么，让我们看看是否可以将任意信号 x [n] 描述为单位脉冲的线性组合。（剧透警告：
事实证明我们可以。）

函数 δ [n − k] 是一个时移离散单位脉冲，除了在时刻 n = k 外，其余处的值均为零，此时其值为 1。换句话说，单位脉冲 δ [n − k] 被“定位”在时刻 k 。考虑在 k 的某个特定值（比如 k = 3 ）处的一个脉冲。让我们将脉冲“缩放” x [3] ，得到 x [3] δ [n − 3] ，以确保该脉冲的“高度”与原始函数在 k = 3 处的值匹配。如果我们对所有可能的

图 14.11. 离散时间中的单位脉冲。

In continuous time, the unit impulse $\delta(t)$ is a bit trickier to define. It is a function whose value is zero everywhere except at $t = 0$, where its value is infinite—but the *area* under the curve is equal to one.

To see how such a strange beast of a function might be formally defined, imagine a “box” function $b(t)$, whose value is zero everwhere except in the interval $[0, T)$, where its value is $1/T$. The area under this curve is just the area of the box, width times height, or $T \times \frac{1}{T} = 1$. Now imagine the limit as $T \to 0$. As this happens, the width of the box approaches zero and its height approaches infinity, but its area remains equal to 1. This is shown in Figure 14.12.

The unit impulse function is typically denoted by the symbol $\delta(t)$. It can be formally defined as follows:

$$\delta(t) = \lim_{T \to 0} b(t),$$

where

$$b(t) = \begin{cases} 1/T & \text{if } t \geq 0 \text{ and } t < T, \\ 0 & \text{otherwise.} \end{cases}$$

As shown in Figure 14.13, we typically plot the unit impulse by drawing an arrow whose height represents the area under the curve (since the actual “height” of the function at $t = 0$ is infinite).

14.2.4.2 Using an Impulse Train to Represent a Signal

Now that we know what the unit impulse signal is, let’s see if we can describe an arbitrary signal $x[n]$ as a linear combination of unit impulses. (Spoiler alert: It turns out we can.)

The function $\delta[n-k]$ is a time-shifted discrete unit impulse, whose value is zero everywhere except at time $n = k$, where it is equal to one. In other words, the unit impulse $\delta[n-k]$ is “positioned” at time k. Consider an impulse at one particular value of k (say, $k = 3$). Let’s make sure that the “height” of that impulse matches the value of the original function at $k = 3$ by “scaling” the impulse by $x[3]$, yielding $x[3]\delta[n-3]$. If we rinse and repeat for all possible

Figure 14.11. The unit impulse in discrete time.

图 14.12 单位冲量可以定义为宽度趋近于零的盒函数 b(t) 的极限。

图 14.13。单位脉冲函数 δ(t) 的值在 t = 0 时为无穷大，其余处均为零。它被绘制为一个单位高度的箭头，表示曲线下的面积为 1。

k 的值，我们得到一系列形式为 x [k] δ [n − k] 的脉冲函数。将所有这些经过缩放和时间平移的脉冲函数相加，就是原始信号 x [n] 的另一种写法：

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k]\delta[n-k]. \tag{14.4}$$

我们不会在这里给出严格的证明，但相信在连续时间中也能以几乎相同的方式进行同样的操作，这或许并不难。唯一的困难在于，对于连续时间，公式 (14.4) 中的和变成了积分。假设有一个无限序列，其中包含时间平移的单位脉冲 δ（t − τ），每个脉冲位于不同的时间 τ 。我们可以按照类似于离散时间的情况构建任意信号 x（t），如下所示：

$$x(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\delta(t-\tau)\,d\tau. \tag{14.5}$$

14.2.4.3 卷积

公式 (14.4) 告诉我们如何将信号 x [n] 表示为简单的时移单位脉冲信号 δ [n − k] 的线性组合。假设只将其中一个加权脉冲输入（x [k] δ [n − k]）放入系统。选择哪一个并不重要，所以我们选择 k = 0 处的那个。这样就得到了输入信号 x [0] δ [n] 。

我们使用符号 x [n] = □ y [n] 来表示输入信号 x [n] 被 LTI 系统转换为输出信号 y [n] 。因此我们可以写成：

$$x[0]\delta[n] \Longrightarrow y[n].$$

Figure 14.12. The unit impulse can be defined as the limit of a box function $b(t)$ whose width approaches zero.

Figure 14.13. The value of the unit impulse function $\delta(t)$ is zero everywhere except at $t = 0$, where it is infinite. It is drawn as an arrow of unit height to indicate that the area under the curve is 1.

values of k, we get a train of impulses of the form $x[k]\delta[n-k]$. Adding all these scaled, time-shifted impulse functions together is just another way of writing the original signal $x[n]$:

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k]\delta[n-k]. \tag{14.4}$$

We won't give a rigorous proof here, but it's probably not too hard to believe that doing this in continuous time works in pretty much the same way. The only difficulty is that for continuous time, the sum in Equation (14.4) becomes an integral. Let's imagine an infinite sequence of time-shifted unit impulses $\delta(t-\tau)$, each one located at a different time τ. We can build up an arbitrary signal $x(t)$ in an analogous fashion to the discrete-time case, as follows:

$$x(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\delta(t-\tau)\,d\tau. \tag{14.5}$$

14.2.4.3 Convolution

Equation (14.4) tells us how to represent a signal $x[n]$ as a *linear combination* of simple, time-shifted unit impulse signals $\delta[n-k]$. Let's imagine putting just *one* of these weighted impulse inputs ($x[k]\delta[n-k]$) through the system. It doesn't matter which one we choose, so let's select the one at $k = 0$. This gives us the input signal $x[0]\delta[n]$.

We'll use the notation $x[n] \Longrightarrow y[n]$ to indicate that an input signal $x[n]$ is being transformed by an LTI system into an output signal $y[n]$. So we can write:

$$x[0]\delta[n] \Longrightarrow y[n].$$

图 14.14.离散和连续时间系统脉冲响应的示例。

x [0] 的值只是一个常数，因此，由于我们处理的是一个线性系统，输出 y [n] 就是该常数乘以系统对单位脉冲 δ [n] 的响应。我们用信号 h [n] 来表示系统对“纯”单位脉冲的响应：δ [n] = □ h [n]。信号 h [n] 称为系统的脉冲响应。因此，我们可以将系统对简单输入信号的响应写成如下形式：

$$x[0]\delta[n] \Longrightarrow x[0]h[n].$$

图 14.14 说明了脉冲响应的概念。

LTI 系统对时移单位脉冲的响应，其实就是一个时移脉冲响应（δ [n − k] = □ h [n − k]）。因此，当 k 不为零时，一切操作完全相同，只是输入和输出信号都经过了 k 的时移：

$$x[k]\delta[n-k] \Longrightarrow x[k]h[n-k].$$

为了找到系统对整个输入信号 x [n] 的响应，我们只需将对每个单独的时间移位组件的响应相加，如下所示：

$$\sum_{k=-\infty}^{+\infty} x[k]\,\delta[n-k] \Longrightarrow \sum_{k=-\infty}^{+\infty} x[k]\,h[n-k].$$

换句话说，我们的系统的输出可以写成如下形式：

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]\,h[n-k]. \tag{14.6}$$

这个非常重要的方程被称为卷积和。为了方便起见，可以引入一个新的数学运算符 □ 来表示该运算

Figure 14.14. Examples of the impulse response of a system in discrete and continuous time.

The value of $x[0]$ is just a constant, so because we're dealing with a linear system, the output $y[n]$ will just be that same constant times the system's response to the unit impulse $\delta[n]$. Let's use the signal $h[n]$ to represent the system's response to a "bare" unit impulse: $\delta[n] \Longrightarrow h[n]$. The signal $h[n]$ is called the *impulse response* of the system. So we can write the system's response to our simple input signal as follows:

$$x[0]\delta[n] \Longrightarrow x[0]h[n].$$

The concept of impulse response is illustrated in Figure 14.14.

The response of an LTI system to a time-shifted unit impulse is just a time-shifted impulse response ($\delta[n-k] \Longrightarrow h[n-k]$). So for values of k other than zero, everything works out exactly the same except that now the input and output signals are both *time-shifted* by k:

$$x[k]\delta[n-k] \Longrightarrow x[k]h[n-k].$$

To find the system's response to the entire input signal $x[n]$, we just sum up the responses to each individual time-shifted component, like this:

$$\sum_{k=-\infty}^{+\infty} x[k]\,\delta[n-k] \Longrightarrow \sum_{k=-\infty}^{+\infty} x[k]\,h[n-k].$$

In other words, the output of our system can be written as follows:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]\,h[n-k]. \tag{14.6}$$

This very important equation is known as the *convolution sum*. It's convenient to introduce a new mathematical operator $*$ to represent the operation

卷积：

$$x[n] * h[n] = \sum_{k=-\infty}^{+\infty} x[k]\, h[n-k]. \tag{14.7}$$

公式 (14.6) 和 (14.7) 提供了一种计算 LTI 系统对任意输入信号 x [n] 的响应 y [n] 的方法，只需给定系统的脉冲响应 h [n] 即可。换句话说，对于 LTI 系统，脉冲响应信号 h [n] 完全描述了系统。这真是太酷了。

连续时间卷积

在上述讨论中，为了简化问题，我们讨论的是离散时间。在连续时间中，一切计算方式基本相同。唯一的区别在于求和变成了积分，我们需要记住在方程中包含微分 dτ。

当我们将任意信号 x (t) 施加到连续时间 LTI 系统的输入时，输出信号可以写成如下形式：

$$y(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{14.8}$$

和以前一样，我们将使用运算符 □ 作为卷积的简写：

$$x(t) * h(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{14.9}$$

与卷积和类似，方程 (14.8) 和 (14.9) 中的积分称为卷积积分。

图 14.15.连续时间卷积运算的可视化。

14.2.4.4 卷积可视化

让我们尝试在连续时间情况下可视化卷积运算。为了计算 y(t)=x(t)□h(t) 中某个特定 t 值（例如 t = 4）的解，我们执行以下步骤，如图 14.15 所示：

1. 绘制 x (τ)，使用 τ 作为时间变量，因为 t 是固定的（在此示例中为 t = 4 ）。

2. 绘制 h (t − τ) 的图。我们可以将其重写为 h (− τ + t)。由于 τ 取负，我们知道脉冲响应在 τ = 0 处发生了翻转。并且由于我们在自变量中添加了 t，我们知道信号向左移动了 t = 4 个单位。

3. 将两个信号在整个τ轴上相乘。

of convolution:

$$x[n] * h[n] = \sum_{k=-\infty}^{+\infty} x[k]\, h[n-k]. \tag{14.7}$$

Equations (14.6) and (14.7) give us a way to calculate an LTI system's response $y[n]$ to any arbitrary input signal $x[n]$, given only the *impulse response* of the system, $h[n]$. In other words, for LTI systems, the impulse response signal $h[n]$ *completely describes the system*. Pretty cool stuff.

Convolution in Continuous Time

In our discussions above, we worked in discrete time to keep things simple. In continuous time, everything works out in pretty much the same way. The only difference is that summations become integrals, and we need to remember to include the differential $d\tau$ in our equations.

When we apply an arbitrary signal $x(t)$ to the input of a continuous-time LTI system, the output signal can be written as follows:

$$y(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{14.8}$$

As before, we'll use the operator $*$ as a shorthand for convolution:

$$x(t) * h(t) = \int_{\tau=-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{14.9}$$

Analogous to the convolution sum, the integral in Equations (14.8) and (14.9) is known as the *convolution integral*.

Figure 14.15. Visualization of the convolution operation in continuous time.

14.2.4.4 Visualizing Convolution

Let's try to visualize the convolution operation in the continuous time case. To evaluate $y(t) = x(t) * h(t)$ for one specific value of t (say, $t = 4$), we perform the following steps as illustrated in Figure 14.15:

1. Plot $x(\tau)$, using τ as the time variable because t is fixed (at $t = 4$ in this example).

2. Plot $h(t-\tau)$. We can rewrite this as $h(-\tau + t)$. Because τ is negated, we know that the impulse response has been flipped about $\tau = 0$. And because we've added t to the independent variable, we know the signal has been shifted to the *left* by $t = 4$ units.

3. Multiply the two signals together across the entire τ axis.

4. 沿τ轴从−∞到+∞求积分，求出所得曲线下的面积。这就是y(t)在特定t值（本例中为t = 4）下的值。

请记住，我们必须对 t 的每个可能值重复此过程，以确定完整的输出信号 y(t)。

14.2.4.5 卷积的一些性质

卷积运算的性质与普通乘法惊人地相似。卷积是：

- 结合律：x(t)□(h1(t)□h2(t))=x(t)□h1(t))□h2(t)；以及
- 分配律：x(t)□(h1(t)+h2(t))=(x(t)□h1(t))+(x(t)□h2(t))。

14.2.5 频域和傅里叶变换

为了理解脉冲响应和卷积的概念，我们将信号描述为单位脉冲的加权和。我们也可以将信号表示为正弦波的加权和。以这种方式表示信号本质上是将其分解为频率分量。这将使我们能够推导出另一个极其强大的数学工具——傅里叶变换。

14.2.5.1 正弦信号

当二维圆周运动投射到单轴上时，就会产生正弦信号。正弦形式的音频信号会在特定频率下产生“纯”音。

最基本的正弦信号是正弦（或余弦）函数。信号 x（t）= sin t 在 t = 0 、π 和 2π 时取值为 0，在 t = π 2 时取值为 1，在 t = 3 π 2 时取值为 −1。

实值正弦信号的最一般形式是

$$x(t) = A\cos(\omega_0 t + \phi). \tag{14.10}$$

这里，A 表示正弦波的振幅（即余弦波的波峰和波谷分别达到 A 的最大值和最小值 − A ）。角频率为 ω 0 ，以弧度/秒为单位（有关频率和角频率的讨论，请参见第 14.1.1 节）。φ 表示相位偏移（也以弧度为单位），它将余弦波沿时间轴向左或向右移动。

4. Integrate from $-\infty$ to $+\infty$ along the τ axis to find the area under the resulting curve. This is the value of $y(t)$ at this one specific value of t (in this example, $t = 4$).

Remember that we must repeat this procedure for every possible value of t in order to determine the complete output signal $y(t)$.

14.2.4.5 Some Properties of Convolution

The properties of the convolution operation are surprisingly analogous to those of ordinary multiplication. Convolution is:

- *commutative:* $x(t) * h(t) = h(t) * x(t)$;
- *associative:* $x(t) * \big(h_1(t) * h_2(t)\big) = \big(x(t) * h_1(t)\big) * h_2(t)$; and
- *distributive:* $x(t) * \big(h_1(t) + h_2(t)\big) = \big(x(t) * h_1(t)\big) + \big(x(t) * h_2(t)\big)$.

14.2.5 The Frequency Domain and the Fourier Transform

In order to arrive at the concepts of impulse response and convolution, we described a signal as a weighted sum of unit impulses. We can also represent a signal as a weighted sum of sinusoids. Representing a signal in this manner essentially breaks it up into its *frequency components*. This will allow us to derive another incredibly powerful mathematical tool—the *Fourier transform*.

14.2.5.1 The Sinusoidal Signal

A sinusoidal signal is produced when a circular motion in two dimensions is projected onto a single axis. An audio signal in the form of a sinusoid produces a “pure” tone at one specific frequency.

The most basic sinusoidal signal is the sine (or cosine) function. The signal $x(t) = \sin t$ takes on the value 0 at $t = 0$, π and 2π, has a value of 1 at $t = \frac{\pi}{2}$ and has a value of -1 at $t = \frac{3\pi}{2}$.

The most general form of a real-valued sinusoidal signal is

$$x(t) = A\cos(\omega_0 t + \phi). \tag{14.10}$$

Here, A represents the *amplitude* of the sine wave (i.e., the peaks and troughs of the cosine wave hit maximum and minimum values of A and $-A$, respectively). The *angular frequency* is ω_0, measured in radians/second (see Section 14.1.1 for a discussion of frequency and angular frequency). ϕ represents a *phase offset* (also measured in radians) that shifts the cosine wave to the left or right along the time axis.

当 A = 1 、ω 0 = 1 和 φ = 0 时，公式 (14.10) 简化为 x (t) = cost t 。当φ = π²时，表达式变为x(t) = sint。cos函数表示圆周运动在横轴上的投影，而sin表示圆周运动在纵轴上的投影。

14.2.5.2 复指数信号

余弦函数实际上并非将信号表示为正弦函数之和的最佳工具。如果我们使用复数，数学运算会更加简单优雅。为了理解其工作原理，我们需要回顾一下复数数学，并了解复数乘法的工作原理。所以，请耐心阅读——读完之后，一切都会变得清晰明了。

复数简要回顾

你可能还记得高中数学课上讲过，复数是一种由实部和虚部组成的二维量。任何复数都可以写成：c = a + jb，其中 a 和 b 为实数，j = √ − 1 为虚数单位。c 的实部为 a = Re(c)，虚部为 b = Im(c)。

你可以将复数想象成二维空间（称为阿尔冈平面）中的一种“向量”[a , b]。然而，重要的是要记住，复数和向量是不可互换的——它们的数学行为截然不同。

我们将复数的幅值定义为其二维“向量”在复平面上的长度：| c | = √ a 2 + b 2 。向量与实轴的夹角称为幅角：arg c = tan − 1 (b / a)。（复数的幅角有时也称为相位。我们将看到，“相位”一词与公式 (14.10) 中的相位偏移 φ 密切相关。）图 14.16 描绘了复数的幅角和幅角。

图 14.16 复数的幅值 | c | = a 2 + b 2 是其在复平面中的长度，其辐角 arg c = tan − 1 (b / a) 是其与 Re 轴的夹角。

When $A = 1$, $\omega_0 = 1$ and $\phi = 0$, Equation (14.10) reduces to $x(t) = \cos t$. When $\phi = \frac{\pi}{2}$, the expression becomes $x(t) = \sin t$. The cos function represents the projection of a circular motion onto the horizontal axis, while sin represents its projection onto the vertical axis.

14.2.5.2 The Complex Exponential Signal

The cosine function isn't actually the best tool for representing a signal as a sum of sinusoids. The math is much simpler and more elegant if we make use of *complex numbers* instead. In order to understand how this works, we need to review complex math, and take a look at how multiplication of complex numbers works. So bear with me here—all will become clear by the time we're done.

A Brief Review of Complex Numbers

You'll probably remember from high-school math class that a complex number is a kind of two-dimensional quantity consisting of a real part and an imaginary part. Any complex number can be written as follows: $c = a + jb$, where a and b are real numbers and $j = \sqrt{-1}$ is the *imaginary unit*. The real part of c is $a = \mathfrak{Re}(c)$, and its imaginary part is $b = \mathfrak{Im}(c)$.

You can visualize a complex number as a kind of "vector" $[a, b]$ in a two-dimensional space known as the *Argand plane*. It's important to remember, however, that complex numbers and vectors are *not* interchangeable—their mathematical behaviors are quite different.

We define the *magnitude* of a complex number as the length of its 2D "vector" representation in the complex plane: $|c| = \sqrt{a^2 + b^2}$. The angle the vector makes with the real axis is known as its *argument:* $\arg c = \tan^{-1}(b/a)$. (The argument of a complex number is sometimes called its *phase*. As we'll see, the term "phase" is closely related to the *phase offset* ϕ in Equation (14.10).) The magnitude and argument of a complex number are depicted in Figure 14.16.

Figure 14.16. The magnitude $|c| = \sqrt{a^2 + b^2}$ of a complex number is its length in the complex plane, and its argument $\arg c = \tan^{-1}(b/a)$ is the angle it makes with the $\mathfrak{Re}$ axis.

复数乘法和旋转

我们不会在这里深入探讨复数的所有性质。有关复数理论的深入讨论，请参阅 http://www.math.wisc.edu/□angenent/Free-Lecture-Notes/freecomplexnumbers.pdf 。不过，这里有一个数学运算值得我们关注：复数乘法。

复数进行代数相乘（这里没有点积或叉积）：

$$\begin{aligned} c_1c_2 &= (a_1 + jb_1)(a_2 + jb_2) \\ &= (a_1a_2) + j(a_1b_2 + a_2b_1) + j^2b_1b_2 \\ &= (a_1a_2 - b_1b_2) + j(a_1b_2 + a_2b_1). \end{aligned} \tag{14.11}$$

如果计算出乘积 c 1 c 2 的幅度和幅角（角度），您会发现幅度等于两个输入幅度的乘积，而幅角是输入幅角之和：

$$\begin{aligned} |c_1c_2| &= |c_1||c_2|; \\ \arg(c_1c_2) &= \arg c_1 + \arg c_2. \end{aligned} \tag{14.12}$$

复数乘法会导致其角度（自变量）相加，这意味着复数乘法会在复平面上产生旋转。如果 c 1 的幅值为 1（| c 1 | = 1），则乘积的幅值将等于 c 2 的幅值（| c 1 c 2 | = | c 2 |）。在这种情况下，乘积表示 c 2 纯粹旋转了等于 arg c 1 的角度（参见图 14.17）。如果 | c 1 | ⁄ = 1，则乘积的幅值将按 | c 1 | 缩放，结果是 c 2 在复平面上进行螺旋运动。

这解释了为什么单位长度四元数在 3D 空间中起到旋转的作用！四元数本质上是一个四维复数，由一个实部和三个虚部组成。因此，四元数在三维空间中遵循的基本规则与常规复数在二维空间中遵循的基本规则相同。

当我们考虑将 j 乘以自身多次时会发生什么时，复数乘法产生旋转这一事实是有道理的：

$$\begin{aligned} 1 \times j &= j, \\ j \times j &= \sqrt{-1}\sqrt{-1} = -1, \\ -1 \times j &= -j, \\ -j \times j &= 1, \\ &\cdots \end{aligned}$$

2 哎呀——这听起来很像是给读者做的练习……

Complex Multiplication and Rotation

We won't get into all of the properties of complex numbers here. Check out http://www.math.wisc.edu/∼angenent/Free-Lecture-Notes/freecomplexnumbers.pdf for an in-depth discussion of complex number theory. However, there is one mathematical operation that does concern us here: the operation of complex *multiplication*.

Complex numbers are multiplied algebraically (no dot or cross products here):

$$\begin{aligned} c_1c_2 &= (a_1 + jb_1)(a_2 + jb_2) \\ &= (a_1a_2) + j(a_1b_2 + a_2b_1) + j^2b_1b_2 \\ &= (a_1a_2 - b_1b_2) + j(a_1b_2 + a_2b_1). \end{aligned} \tag{14.11}$$

If you work out[2] the magnitude and argument (angle) of the product c_1c_2, you'll find that the magnitude is equal to the *product* of the two input magnitudes, and the argument is the *sum* of the input arguments:

$$\begin{aligned} |c_1c_2| &= |c_1||c_2|; \\ \arg(c_1c_2) &= \arg c_1 + \arg c_2. \end{aligned} \tag{14.12}$$

The fact that multiplication of complex numbers causes their angles (arguments) to *add* means that complex multiplication produces a *rotation* in the complex plane. If the magnitude of c_1 is unity ($|c_1| = 1$), then the magnitude of the product will be equal to the magnitude of c_2 ($|c_1c_2| = |c_2|$). In this case, the product represents a *pure rotation* of c_2 by an angle equal to $\arg c_1$ (see Figure 14.17). If $|c_1| \neq 1$, then the product's magnitude will be *scaled* by $|c_1|$, and the result is that c_2 undergoes a *spiral motion* in the complex plane.

This explains why *unit-length* quaternions operate as rotations in 3D space! A quaternion is essentially a four-dimensional complex number, with one real part and three imaginary parts. So a quaternion follows the same basic rules in three dimensions that a regular complex number follows in two dimensions.

The fact that complex multiplication produces a rotation makes sense when we consider what happens when we multiply j by itself many times:

$$\begin{aligned} 1 \times j &= j, \\ j \times j &= \sqrt{-1}\sqrt{-1} = -1, \\ -1 \times j &= -j, \\ -j \times j &= 1, \\ &\cdots \end{aligned}$$

[2] Yikes—this sounds an awful lot like an exercise for the reader…

图 14.17 将两个幅值均为 1 的复数相乘，在复平面上产生纯旋转。

图 14.18。将虚数 j = √ − 1 乘以其自身，就像在复平面上将单位向量旋转 90 度一样。

因此，将 j 乘以自身就相当于在复平面上将一个单位向量旋转 90 度。实际上，将任何复数乘以 j 都会产生将其旋转 90 度的效果。如图 14.18 所示。

复指数和欧拉公式

对于任何复数 c 且 | c | = 1 ，函数 f (n) = cn ，其中 n 取一系列增加的正实值，将在

Figure 14.17. Multiplying two complex numbers together that both have magnitudes of 1 produces a *pure rotation* in the complex plane.

Figure 14.18. Multiplying the imaginary number $j = \sqrt{-1}$ by itself acts like rotating a unit vector by 90 degrees in the complex plane.

So multiplying j by itself is like *rotating* a unit vector by 90 degrees in the complex plane. In fact, multiplying *any* complex number by j has the effect of rotating it by 90 degrees. This is illustrated in Figure 14.18.

The Complex Exponential and Euler's Formula

For any complex number c with $|c| = 1$, the function $f(n) = c^n$, with n taking on a sequence of increasing positive real values, will trace out a *circular path* in

图 14.19。将复数反复乘以自身会在阿尔冈平面中描绘出一条圆形路径，当投影到通过原点的任意轴上时，都会产生正弦曲线。

复平面。二维空间中的任何圆形路径都会沿纵轴描绘出一条正弦曲线，并沿横轴描绘出一条相应的余弦曲线。如图 14.19 所示。

将复数进行实数幂运算（cn）会在复平面上产生旋转，因此当投影到平面上的任意轴上时，都会产生正弦曲线。事实证明，我们也可以通过将实数进行复数幂运算（nc）来获得这种旋转效果。这意味着我们可以将公式 (14.10) 写成复数形式，如下所示：

$$
\begin{aligned}
e^{j\omega_0 t} &= \cos\omega_0 t + j\sin\omega_0 t,\ t \in \mathbb{R}; \qquad (14.13)\\
\mathfrak{Re}\left[e^{j\omega_0 t}\right] &= \cos\omega_0 t;\\
\mathfrak{Im}\left[e^{j\omega_0 t}\right] &= \sin\omega_0 t,
\end{aligned}
$$

其中 e ≈ 2.71828 是定义自然对数函数底数的实超越数。

方程 (14.13) 是整个数学中最重要的方程之一，它被称为欧拉公式。它的成立原因至今仍是个谜（即使对一些经验丰富的数学家来说也是如此）。该定理可以通过研究 e jt 的泰勒级数展开式来解释，或者通过考虑 ex 的导数，然后让 x 变为复数来解释。但就我们的目的而言，依靠我们从复数乘法如何导致复平面旋转中获得的直觉就足够了。

Figure 14.19. Multiplying a complex number by itself repeatedly traces out a circular path in the Argand plane, producing a sinusoid when projected onto any axis through the origin.

the complex plane. Any circular path in two dimensions traces out a sine curve along the vertical axis, and a corresponding cosine curve along the horizontal axis. This is illustrated in Figure 14.19.

Raising a *complex* number to a *real* power (c^n) produces rotation in the complex plane, and therefore yields a sinusoid when projected onto any axis in the plane. As it turns out, we can also get this rotational effect by raising a *real* number to a *complex* power (n^c). This means that we can write Equation (14.10) in terms of complex numbers as follows:

$$
\begin{aligned}
e^{j\omega_0 t} &= \cos\omega_0 t + j\sin\omega_0 t,\ t \in \mathbb{R}; \qquad (14.13)\\
\mathfrak{Re}\left[e^{j\omega_0 t}\right] &= \cos\omega_0 t;\\
\mathfrak{Im}\left[e^{j\omega_0 t}\right] &= \sin\omega_0 t,
\end{aligned}
$$

where $e \approx 2.71828$ is the real transcendental number that defines the base of the natural logarithm function.

Equation (14.13) is one of the most important equations in all of mathematics. It is known as *Euler's formula*. Why it works is a bit of a mystery (even to some seasoned mathematicians). The theorem can be explained by looking at the Taylor series expansion of e^{jt}, or by considering the derivative of e^x and then allowing x to become a complex number. But for our purposes, it should suffice to rely on the intuitions we gained from looking at how complex multiplication results in rotation in the complex plane.

14.2.5.3 傅里叶级数

现在我们已经有了将正弦波表示为复数所需的数学工具，让我们再次将注意力转向将信号表示为正弦波之和的任务。

当信号具有周期性时，这样做最容易。在这种情况下，我们可以将信号写成一系列谐波相关的正弦波之和：

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{j(k\omega_0)t}. \tag{14.14}$$

我们称之为信号的傅里叶级数表示。这里，复指数函数 ej（k ω 0）t 是我们用来构建信号的正弦分量。这些分量是谐波相关的，因为每个分量的频率都是所谓基频 ω 0 的整数倍 k。系数 αk 表示信号 x（t）中每个谐波的“量”。

14.2.5.4 傅里叶变换

本书无法完整解释这一主题，但就我们的目的而言，只需说明（无需任何证明！）任何行为合理的信号，即使是非周期信号，都可以表示为正弦波的线性组合。一般而言，任意信号可能包含任意频率的分量，而不仅仅是谐波相关的频率。因此，公式 (14.14) 中谐波系数 αk 的离散集合变成了一个连续的数值集，表示信号包含每个频率的“多少”。

我们可以设想一个新的函数 X(ω)，其自变量是频率ω而不是时间t，其值表示原始信号 x(t) 中每个频率的含量。我们称 x(t) 是信号的时域表示，而 X(ω) 是信号的频域表示。

从数学上讲，我们可以利用傅里叶变换从信号的时域表示中找到信号的频域表示，反之亦然：

$$X(\omega) = \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt; \tag{14.15}$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega. \tag{14.16}$$

3 所有满足所谓狄利克雷条件的信号都具有傅里叶变换，因此对于我们的目的而言是“相当良好表现的”。

14.2.5.3 The Fourier Series

Now that we have the mathematical tools we need to represent sinusoids as complex numbers, let's turn our attention again to the task of representing a signal as a sum of sinusoids.

Doing this is easiest when the signal is *periodic*. In this case, we can write the signal as a sum of *harmonically related* sinusoids:

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{j(k\omega_0)t}. \tag{14.14}$$

We call this the *Fourier series* representation of the signal. Here, the complex exponential functions $e^{j(k\omega_0)t}$ are the sinusoidal components from which we are building up the signal. These components are *harmonically related*, in that each one has a frequency that is an integer multiple k of the so-called *fundamental frequency* ω_0. The coefficients a_k represent the "amount" of each harmonic present in the signal $x(t)$.

14.2.5.4 The Fourier Transform

A full explanation of this topic is beyond the scope of this book, but for our purposes it will suffice to state (without any proof whatsoever!) that *any* reasonably well-behaved signal,[3] even signals that are *non-periodic*, can be represented as a linear combination of sinusoids. In general, an arbitrary signal may contain components at *any* frequency, not just frequencies that are harmonically related. As such, the discrete set of harmonic coefficients a_k in Equation (14.14) becomes a continuum of values representing "how much" of each frequency the signal contains.

We can envision a new function $X(\omega)$ whose independent variable is the frequency ω rather than time t, and whose value represents the amount of each frequency present in the original signal $x(t)$. We say that $x(t)$ is the *time domain* representation of the signal, while $X(\omega)$ is its *frequency domain* representation.

Mathematically, we can find the frequency domain representation of a signal from its time domain representation, and vice versa, by using the *Fourier transform*:

$$X(\omega) = \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt; \tag{14.15}$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega. \tag{14.16}$$

[3] All signals that meet the so-called *Dirichlet conditions* have Fourier transforms and are therefore "reasonably well-behaved" for our purposes.

图 14.20 傅里叶变换产生一个复值频域信号。波特图用于直观地显示该复值信号的幅度和相位（或幅角）。

如果将公式 (14.16) 与公式 (14.14) 中的傅里叶级数进行比较，就会发现它们之间的相似之处。我们不再通过一系列离散的系数 ak 来描述频率分量的“量”，而是使用连续函数 X（ω）来描述它们。但在这两种情况下，我们都将 x（t）表示为正弦曲线的“和”。

14.2.5.5 波特图

一般来说，实值信号的傅里叶变换是一个复值信号（X（ω）∈ C）。在可视化傅里叶变换时，我们通常使用两幅图来绘制。例如，我们可以绘制它的实部和虚部。或者，我们可以在两个不同的图上绘制它的幅度和幅角——这种可视化方法称为波特图（发音为“Boh-dee”）。

图 14.20 显示了一个信号及其波特图的示例。

14.2.5.6 快速傅里叶变换（FFT）

存在一系列用于计算离散时间傅里叶变换的快速算法。这类算法被称为快速傅里叶变换（FFT），恰如其分。您可以在 http://en.wikipedia.org/wiki/Fast_Fourier_transform 上了解更多关于 FFT 的信息。

Figure 14.20. The Fourier transform yields a complex-valued frequency domain signal. A Bode plot is used to visualize this complex-valued signal in terms of its magnitude and its phase (or argument).

If you compare Equation (14.16) to the Fourier series from Equation (14.14), you can see the similarity. Rather than describing the “amounts” of the frequency components via a discrete series of coefficients a_k, we're now describing them using the continuous function $X(\omega)$. But in both cases we're representing $x(t)$ as a “sum” of sinusoids.

14.2.5.5 Bode Plots

In general the Fourier transform of a real-valued signal is a *complex-valued* signal ($X(\omega) \in \mathbb{C}$). When visualizing the Fourier transform, we often draw it using two plots. For example, we might plot its real and imaginary components. Or we might plot its magnitude and its argument (angle) on two different plots—a visualization known as a *Bode plot* (pronounced “Boh-dee”). Figure 14.20 shows an example of a signal and its Bode plot.

14.2.5.6 The Fast Fourier Transform (FFT)

A collection of fast algorithms exist for calculating the Fourier transform in discrete time. This family of algorithms is called, aptly enough, the *fast Fourier transform* or FFT. You can read more about the FFT at http://en.wikipedia.org/wiki/Fast_Fourier_transform.

14.2.5.7 傅里叶变换和卷积

值得注意的是，时域中的卷积对应于频域中的乘法，反之亦然。给定一个脉冲响应为 h(t) 的系统，我们知道可以找到系统对输入 x(t) 的输出 y(t)，如下所示：

$$y(t) = x(t) * h(t).$$

在频域中，给定脉冲响应 H(ω) 和输入 X(ω) 的傅里叶变换，我们可以找到输出的傅里叶变换如下：

$$Y(\omega) = X(\omega)H(\omega).$$

这个结果非常不可思议，而且非常实用。有时，使用系统的脉冲响应 h(t) 在时间轴上进行卷积更方便，而有时，使用系统的频率响应在频域上进行乘法更方便。

$H(\omega)$.
事实证明，LTI 系统表现出一种称为对偶性的性质，这意味着你可以反转时间和频率的角色，而几乎相同的数学规则仍然适用。例如，我们可以通过观察在频率轴上对两个信号的傅里叶变换进行卷积时发生的情况来理解信号调制（一个信号与另一个信号的相乘）在时域中的工作原理。两种解决问题的方法总比一种好！

14.2.5.8 过滤

傅里叶变换使我们能够直观地看到构成几乎任何音频信号的频率集。滤波器是一种 LTI 系统，它可以衰减选定范围的输入频率，同时保持所有其他频率不变。低通滤波器会保留低频，同时衰减高频。高通滤波器则相反，它会保留高频，衰减低频。带通滤波器会同时衰减低频和高频，但会保留有限通带内的频率。陷波滤波器则相反，它会保留低频和高频，但会衰减有限阻带内的频率。

滤波器用于立体声系统的均衡器，根据用户输入衰减或增强特定频率。如果噪声信号和所需信号的频谱占据频率轴的不同区域，滤波器也可以用来衰减噪声。例如，如果高频噪声信号

14.2.5.7 Fourier Transforms and Convolution

It is interesting to note that convolution in the time domain corresponds to multiplication in the frequency domain and vice versa. Given a system whose impulse response is $h(t)$, we know that we can find the output of the system $y(t)$ in response to an input $x(t)$ as follows:

$$y(t) = x(t) * h(t).$$

In the frequency domain, given the Fourier transforms of the impulse response $H(\omega)$ and the input $X(\omega)$, we can find the Fourier transform of the output as follows:

$$Y(\omega) = X(\omega)H(\omega).$$

This result is pretty incredible, and it's also very handy. Sometimes it is more convenient to perform a convolution on the time axis using a system's impulse response $h(t)$, while at other times it's more convenient to perform a multiplication in the frequency domain using the system's *frequency response* $H(\omega)$.

As it turns out, LTI systems exhibit a property called *duality*, which says that you can reverse the roles of time and frequency and virtually the same mathematical rules continue to apply. So, for example, we can understand how signal *modulation* (the multiplication of one signal by another) works in the time domain by looking at what happens when we convolve the Fourier transforms of the two signals on the frequency axis. Having two ways of tackling a problem is always better than one!

14.2.5.8 Filtering

The Fourier transform allows us to visualize the set of frequencies that make up virtually any audio signal. A *filter* is an LTI system that attenuates a selected range of input frequencies while leaving all other frequencies unaltered. A *low-pass* filter retains low frequencies while attenuating high frequencies. A *high-pass* filter does the opposite, retaining high frequencies and attenuating lower frequencies. A *band-pass* filter attenuates both low and high frequencies but retains frequencies within a limited *passband*. A *notch* filter does the opposite, retaining low and high frequencies but attenuating frequencies within a limited *stopband*.

Filters are used in a stereo system's *equalizer*, by attenuating or boosting specific frequencies based on user inputs. Filters can also be used to attenuate noise, if the spectra of the noise signal and the desirable signal occupy different regions of the frequency axis. For example, if a high-frequency noise signal is

图 14.21 理想滤波器的频率响应 H（ω）在通带中的值为 1，在阻带中的值为 0。

如果噪声对低频语音或音乐信号产生不利影响，则可以使用低通滤波器来消除噪声。

理想滤波器的频率响应 H（ω）看起来像一个矩形框，通带值为 1，阻带值为 0。当我们将其乘以输入信号 X（ω）的傅里叶变换时，输出 Y（ω）= X（ω）H（ω）将完全保留其通带频率，而阻带频率则全部设置为零。理想滤波器的频率响应如图 14.21 所示。

当然，一个完全通过某些频率而完全抑制其他频率的理想滤波器可能并非理想之选。大多数实际滤波器的频率响应在通带和阻带之间具有逐渐下降的趋势。这有助于在所需频率和不需要频率之间没有单一、明确界限的情况下进行滤波。图 14.22 显示了具有逐渐下降趋势的低通滤波器的频率响应。

大多数高保真音频设备上都配有均衡器 (EQ)，允许用户调整输出的低音、中音和高音音量。EQ 实际上只是一组针对不同频率范围进行调谐的滤波器，并串联应用于音频信号。

过滤理论是一个庞大的研究领域，我们无法在此一一详述。更多信息，请参阅 [41，第 6 章]。

14.3 声音技术

在我们完全理解组成游戏音频引擎的软件之前，我们需要牢牢掌握音频硬件和技术以及行业专业人士用来描述它的术语。

Figure 14.21. The frequency response $H(\omega)$ for an ideal filter has a value of one in the passband and zero in the stopband.

adversely affecting a lower-frequency voice or music signal, a low-pass filter could be used to eliminate the noise.

The frequency response $H(\omega)$ of an ideal filter looks like a rectangular box, with a value of one in the passband and zero in the stopband. When we multiply this by the Fourier transform of our input signal $X(\omega)$, the output $Y(\omega) = X(\omega)H(\omega)$ will have its passband frequencies preserved exactly, and its stopband frequencies all set to zero. The frequency response for an ideal filter is shown in Figure 14.21.

Of course, an ideal filter that completely passes certain frequencies and completely suppresses others may not be desirable. The frequency responses of most real-world filters have a gradual fall-off between the passband and stopband. This aids filtering in situations where there is no single, clear-cut line between the desirable frequencies and the unwanted frequencies. The frequency response of a low-pass filter with a gradual fall-off is shown in Figure 14.22.

The *equalizer* (EQ) found on most high-fidelity audio equipment permits the user to adjust the amount of bass, mid-range and treble that is output. An EQ is really just a collection of filters tuned to different frequency ranges and applied in series to an audio signal.

Filtering theory is an immense field of study, so we can't possibly do it justice here. For a great deal more information, see [41, Chapter 6].

14.3 The Technology of Sound

Before we can fully understand the software that comprises a game's audio engine, we need a firm grasp of audio hardware and technology and of the terminology used by industry professionals to describe it.

14.3.1 模拟音频技术

最早的音频硬件基于模拟电子技术。这是录制、处理和播放音频压缩波最简单的方法，因为声音本身就是一种模拟物理现象。在本节中，我们将简要探讨一些关键的模拟音频技术。

14.3.1.1 麦克风

麦克风（也称为“mic”或“麦克”）是一种将音频压缩波转换为电信号的换能器。麦克风利用各种技术将声波的机械压力变化转换为基于电压变化的等效信号。动圈麦克风利用电磁感应，而电容麦克风利用电容变化。其他类型的麦克风则利用压电发电或光调制来产生电压信号。

不同的麦克风具有不同的灵敏度模式，称为极性模式。这些模式描述了麦克风对围绕其中心轴不同角度的声音的灵敏度。全向麦克风对所有方向的灵敏度相同。双向麦克风有两个灵敏度“叶”，呈8字形。心形麦克风本质上具有单向灵敏度曲线，因其略呈心形的极性模式而得名。一些常见的

图 14.22。RC（电阻-电容）低通滤波器的频率响应 H（ω），具有逐渐下降的曲线。两幅图的横轴和纵轴均采用对数刻度绘制。

14.3.1 Analog Audio Technology

The earliest audio hardware was based on analog electronics. This was the easiest way to record, manipulate and play back audio compression waves, because sound is itself an analog physical phenomenon. In this section, we'll briefly explore some key analog audio technologies.

14.3.1.1 Microphones

A microphone (also known as a "mic" or "mike") is a transducer that converts an audio compression wave into an electronic signal. Microphones make use of various technologies in order to convert the mechanical pressure variations of a sound wave into an equivalent signal based on variations in electric voltage. A *dynamic microphone* uses electromagnetic induction, while a *condenser microphone* utilizes changes in capacitance. Other types of mics use piezoelectric generation or light modulation to produce a voltage signal.

Different microphones have different sensitivity patterns, known as *polar patterns*. These patterns describe how sensitive the mic is to sound at various angles about its central axis. An omnidirectional mic is equally sensitive in all directions. A bidirectional mic has two sensitivity "lobes" in the shape of a figure eight. A cardioid mic has essentially a unidirectional sensitivity profile, so named because of its somewhat heart-shaped polar pattern. Some common

Figure 14.22. The frequency response $H(\omega)$ for an RC (resistor-capacitor) low-pass filter with a gradual fall-off. Both the horizontal and vertical axes of both plots are drawn using a logarithmic scale.

图 14.23. 三种典型的麦克风极性模式，从左上角顺时针方向依次为：全向性、心形和双向性。

麦克风极性图如图 14.23 所示。

14.3.1.2 扬声器

扬声器本质上是一个反向操作的麦克风——它是一种换能器，将变化的输入电压信号转换为膜片中的振动，进而引起气压变化，从而产生声压波。

Figure 14.23. Three typical microphone polar patterns, clockwise from upper left: omnidirectional, cardioid and bidirectional.

microphone polar patterns are illustrated in Figure 14.23.

14.3.1.2 Speakers

A *speaker* is basically a microphone operated in reverse—it is a transducer that converts a varying input voltage signal into vibrations in a membrane, which in turn gives rise to air pressure variations that result in a sound pressure wave.

14.3.1.3 扬声器布局：立体声

音响系统通常支持多个扬声器输出声道。诸如 iPod、车载音响系统或您祖父的便携式“音箱”之类的立体声设备至少支持左右立体声声道各两个扬声器。一些高保真立体声系统还拥有两个额外的“高音扬声器”——这些微型扬声器能够重现左右声道中的最高频率声音。这使得两个主扬声器可以更大，从而更好地覆盖低音。一些立体声系统还支持低音炮或 LFE（低频效果）扬声器。这类系统有时被称为 2.1 系统——左右声道各两个，LFE 扬声器“点一”。

耳机与扬声器

区分开放式房间中的立体声扬声器和立体声耳机非常重要。房间中的立体声扬声器通常位于聆听者前方，并向两侧偏移。这意味着来自左扬声器的声波实际上也会被右耳接收到，反之亦然。来自较远扬声器的声波到达耳朵时会略有时间延迟（相移）和衰减。来自较远扬声器的相移声波往往会与来自较近扬声器的声波发生干扰。音响系统应将这种干扰考虑在内，以产生最高品质的声音。

另一方面，耳机直接与耳朵接触，因此左右声道完全隔离，不会互相干扰。此外，由于耳机几乎直接将声音传送到耳道，耳朵形状本身的头部相关传输效应 (HRTF) 不会发挥作用（参见第 14.1.4 节），这意味着听众接收到的空间信息会略少。

14.3.1.4 扬声器布局：环绕声

家庭影院环绕声系统通常有两种类型：5.1 和 7.1。正如您毫无疑问猜到的那样，这些数字指的是五个或七个“主”扬声器，加上一个低音炮。环绕声系统的目标是通过提供位置信息以及高保真声音再现，让听众沉浸在逼真的声景中（参见第 14.1.4 节）。5.1 系统中的主扬声器声道包括：中置、左前、右前、左后和右后。7.1 系统增加了两个额外的扬声器，即环绕左和环绕右，它们分别放置在听众的两侧。杜比数字 AC-3 和 DTS 是两种流行的环绕声

14.3.1.3 Speaker Layouts: Stereo

Sound systems usually support multiple speaker output channels. A stereo device such as an iPod, the sound system in your car or your grandpa's portable "boom box" supports at least two speakers for the left and right stereo channels. Some high-fidelity stereo systems also boast two additional "tweeters"—tiny speakers that are capable of reproducing the highest-frequency sounds within the left and right channels. This allows the two main speakers to be larger, and therefore better at covering the bass. Some stereo systems also support a subwoofer or LFE (low-frequency effects) speaker. Such systems are sometimes called 2.1 systems—two for the left and right, and "dot one" for the LFE speaker.

Headphones versus Speakers

It's important to distinguish between stereo speakers in an open room and stereo headphones. Stereo speakers in a room will typically be positioned in front of the listener and offset to either side. This means that the sound waves coming from the *left* speaker are actually received by the *right* ear as well, and vice versa. The waves from the more-distant speaker will arrive at the ear with a slight time delay (phase shift) and a slight attenuation. The phase-shifted sound waves from the more-distant speaker will tend to *interfere* with those coming from the closer speaker. The sound system should take this interference into account in order to produce the highest quality sound.

Headphones, on the other hand, come in direct contact with the ears, so the left and right channels are perfectly isolated and do not interfere with one another. Also, because headphones deliver sound almost directly to the ear canal, the head-related transfer effects (HRTF) of the shape of the ears themselves do not come into play (see Section 14.1.4), meaning that somewhat less spatial information is received by the listener.

14.3.1.4 Speaker Layouts: Surround Sound

Home theater *surround sound* systems typically come in two flavors: 5.1 and 7.1. As you undoubtedly guessed, these numbers refer to the five or seven "main" speakers, plus the one subwoofer. The goal of a surround sound system is to immerse the listener in a realistic soundscape, by providing *positional* information as well as high-fidelity sound reproduction (see Section 14.1.4). The main speaker channels in a 5.1 system are: center, front left, front right, rear left and rear right. A 7.1 system adds two additional speakers, surround left and surround right, which are intended to be placed directly to either side of the listener. Dolby Digital AC-3 and DTS are two popular surround sound

图 14.24.7.1 环绕声家庭影院系统的扬声器布置。

技术。典型的7.1家庭影院的扬声器布局如图14.24所示。

杜比环绕声 (Dolby Surround)、杜比定向逻辑 (Dolby Pro Logic) 和杜比定向逻辑 II (Dolby Pro Logic II) 是将立体声源信号扩展为 5.1 环绕声的技术。立体声信号缺乏直接驱动 5.1 扬声器配置所需的位置信息。但使用这些杜比技术，可以利用原始立体声源信号中的各种线索，启发式地生成缺失位置信息的近似值。

14.3.1.5 模拟信号电平

音频电压信号可以以各种电压电平传输。麦克风通常会产生低振幅电压信号，这些信号称为麦克风电平信号。组件之间的连接则使用电压更高的线路电平信号。专业音频设备和消费类电子产品在线路电平电压方面存在很大差异。专业设备通常设计为可处理的线路电平范围从标称信号峰峰值 2.191 V（伏特）到最高峰峰值 3.472 V。消费类设备上“线路电平”信号的峰峰值电压变化很大，但大多数消费类设备的输出峰峰值高达 1.0 V，并且输入可以处理高达 2.0 V 的信号。重要的是

Figure 14.24. Speaker arrangement for a 7.1 surround sound home theater system.

technologies. The speaker layout of a typical 7.1 home theater is shown in Figure 14.24.

Dolby Surround, Dolby Pro Logic and Dolby Pro Logic II are technologies for expanding a stereo source signal into 5.1 surround sound. A stereo signal lacks the positional information necessary to drive the 5.1 speaker configuration directly. But using these Dolby technologies, an approximation of the missing positional information can be generated heuristically using various cues found within the original stereo source signal.

14.3.1.5 Analog Signal Levels

Audio voltage signals may be transmitted at various voltage levels. A microphone usually produces a low-amplitude voltage signal—these are called *mic-level* signals. For connections between components, higher-voltage *line-level* signals are used. There's a big difference between professional audio equipment and consumer electronics when it comes to line-level voltages. Professional gear is usually designed to work with line levels ranging from 2.191 V (volts) peak-to-peak for a nominal signal up to a maximum voltage of 3.472 V peak-to-peak. The peak-to-peak voltage of a "line level" signal on consumer equipment varies quite a bit, but most consumer devices output up to 1.0 V peak-to-peak, and have inputs that can handle up to 2.0 V signals. It's impor-

连接音频设备时，务必确保输入和输出信号的电平匹配。如果输入的电压过高，设备无法处理，会导致信号削波。如果输入的电压过低，则会导致音频听起来比应有的音量小。

14.3.1.6 放大器

麦克风产生的微弱电压必须经过放大，才能以足够的力度驱动扬声器，产生可听见的声波。放大器是一种模拟电子电路，其输出几乎与输入信号完全相同，但信号幅度显著增加。放大器的本质是增加信号的功率。它通过从某种电源获取能量，并以模拟输入信号随时间变化的方式驱动该电源产生的升高电压来实现这一点。换句话说，放大器会调制其电源的输出，以匹配其电压低得多的输入信号。

放大器的核心技术是晶体管——这个众所周知且极具创造力的器件，是许多现代电子设备的核心，包括其巅峰之作——计算机。晶体管利用半导体材料将两个原本隔离的独立电路之间的电压连接起来。这样，低压信号就可以用来驱动高压电路。这正是放大器所需要的。我们不会在这里深入探讨晶体管和放大器的底层工作原理。但如果您对此感兴趣，可以观看这个精彩的YouTube视频，了解第一个晶体管的工作原理：https://www.youtube.com/watch?v=RdYHljZi7ys。您也可以在这里阅读更多关于放大器电路的信息：http://en.wikipedia.org/wiki/Amplifier。

放大系统的增益 A 定义为输出功率 P out 与输入功率 P in 之比。与声压级一样，增益通常以分贝为单位：

$$A = 10\log_{10}\left(\frac{P_{\text{out}}}{P_{\text{in}}}\right)\ \text{dB}.$$

14.3.1.7 音量/增益控制

音量控制器本质上是一个逆放大器，也称为衰减器。它不是增加电信号的幅度，而是降低幅度，同时保持波形的所有其他方面不变。在家庭影院系统中，D/A 转换器会产生一个幅度非常小的电压信号。功率放大器会将此信号放大到最大“安全”输出功率，超过此功率，扬声器发出的声音就会开始削波和失真（甚至损坏硬件）。然后，音量控制器

tant to match the levels of input and output signals when connecting audio equipment. Passing a voltage that is too high for the device to handle will cause clipping of the signal. And passing a voltage that is too low will result in audio that sounds quieter than it should.

14.3.1.6 Amplifiers

The small voltages produced by a microphone must be *amplified* in order to drive speakers with enough force to produce audible sound waves. An *amplifier* is an analog electronic circuit that produces at its output a nearly exact replica of its input signal, but with the *amplitude* of the signal increased significantly. An amp essentially increases the *power* content of a signal. It does this by drawing from some kind of power source, and driving the increased voltage produced by this power source in such a way as to mimic the behavior of the input signal over time. In other words, an amp *modulates* the output of its power source to match its much lower-voltage input signal.

The core technology behind an amplifier is the *transistor*—that well-known and utterly ingenious device that sits at the heart of many modern electronic devices, including its crowning achievement—the computer. A transistor makes use of a semiconducting material in order to link the voltages between two otherwise isolated, independent circuits. As such, a low-voltage signal can be used to drive a higher-voltage circuit. This is exactly what is required of an amplifier. We won't get into the details of how transistors and amplifiers work under the hood here. But if you're curious, you can whet your whistle with this great YouTube video on how the very first transistor worked: https://www.youtube.com/watch?v=RdYHljZi7ys. And you can read more about amplifier circuits here: http://en.wikipedia.org/wiki/Amplifier.

The *gain* A of an amplification system is defined as the ratio of output power P_{out} to input power P_{in}. Like sound pressure level, gain is typically measured in decibels:

$$A = 10\log_{10}\left(\frac{P_{\text{out}}}{P_{\text{in}}}\right)\ \text{dB}.$$

14.3.1.7 Volume/Gain Controls

A *volume control* is basically an inverse amplifier, also known as an *attenuator*. Rather than increasing the amplitude of an electrical signal, it *decreases* the amplitude, while keeping all other aspects of the waveform intact. In a home theater system, the D/A converter produces a voltage signal with a very small amplitude. The power amp boosts this signal up to the maximum "safe" output power, beyond which the sound produced by your speakers would begin to clip and distort (or even damage your hardware). The volume control then

衰减该最大输出功率以产生所需收听音量的声音。

音量控制器的制作比放大器简单得多。只需在放大器输出和扬声器之间的电路中引入一个可变电阻（也称为电位器）即可。当电阻最小（等于或非常接近于零）时，输入信号的幅度保持不变，从而产生最大音量的声音。当电阻达到最大值时，输入信号的幅度会最大程度地衰减，从而产生最小音量的声音。

如果您家里的立体声音响系统以分贝为单位输出音量，您可能已经注意到，这些值始终为负值。这是因为音量控制会衰减功率放大器的输出。音量表仍然像增益一样测量，但“输入”功率是放大器的最大功率，“输出”功率是用户选择的音量：

$$A = 10\log_{10}\left(\frac{P_{\text{体积}}}{P_{\max}}\right)\ \text{dB},$$

which will be negative as long as $P_{\text{volume}} < P_{\max}$.

14.3.1.8 模拟接线和连接器

模拟单声道音频电压信号可以用一对线传输；立体声信号则需要三根线（两个声道加一个公共地线）。线路可以是设备内部的，这种情况下通常称为总线。线路也可以是外部的，用于连接不同的设备。

外部线路通常通过高端扬声器上常见的“夹式”或螺旋式连接器，或各种标准化连接器连接到音频硬件。例如，RCA 插孔、大型 TRS（尖/环/套）插孔（20 世纪初电话接线员使用的那种）、TRS 迷你插孔（iPod、手机和大多数 PC 声卡上都有）、键控插孔（最常见于高品质麦克风和功率放大器）等等。

音频线材的质量等级多种多样。粗线材的电阻更小，因此可以将信号传输到更远的距离，而不会出现不可接受的衰减。可选的屏蔽层有助于降低噪音。当然，电线和连接器的金属材质选择也会影响线材的质量。

attenuates this maximum output power to produce sound at the desired listening volume.

A volume control is much simpler to make than an amplifier. One can be constructed by introducing a variable resistor, also known as a *potentiometer*, into the circuit somewhere between the amplifier's output and the speakers. When the resistance is at its minimum (at or very close to zero), the amplitude of the input signal isn't changed, and a sound of maximum volume is produced. When the resistance is at its maximum setting, the input signal's amplitude is maximally attenuated, and a sound of minimum volume is produced.

If your stereo system at home reports the volume in decibels, you've probably noticed that the values are always negative. This is because the volume control is attenuating the output of the power amp. The volume meter is still measured like a gain, but the "input" power is the maximum power of the amp, and the "output" power is the volume selected by the user:

$$A = 10\log_{10}\left(\frac{P_{\text{volume}}}{P_{\max}}\right)\ \text{dB},$$

which will be negative as long as $P_{\text{volume}} < P_{\max}$.

14.3.1.8 Analog Wiring and Connectors

An analog monophonic audio voltage signal can be carried by a pair of wires; a stereo signal requires three wires (two channels plus a common ground). The wiring can be internal to a device, in which case it is usually called a *bus*. Wiring can also be external, for use in connecting different devices to one another.

External wiring is typically connected to audio hardware either via a direct "clip" or screw-post connector, of the kind found on high-end speakers, or via various standardized connectors. Examples include RCA jacks, large TRS (tip/ring/sleeve) jacks (the kind used by telephone operators in the early 1900s), TRS mini-jacks (found on your iPod, mobile phone and most PC sound cards), keyed jacks (found most often on high-quality microphones and power amps), and the list goes on.

Audio wiring is available in a wide range of quality levels. Thicker-gauge wiring offers less resistance and therefore can transmit signals over farther distances without unacceptable levels of attenuation. Optional shielding can help reduce noise. And of course the choice of which metal to use in the construction of the wires and connectors can make a difference in the quality of the wiring as well.

14.3.2 数字音频技术

光盘 (CD) 的问世标志着音频行业迈向数字音频存储和处理的转折点。数字技术开辟了许多新的可能性，从缩小存储介质的尺寸、增加存储介质的容量，到利用强大的计算机硬件和软件以前所未有的方式合成和处理音频。如今，模拟音频存储设备已成为过去，模拟音频信号通常只在必要时使用——例如麦克风和扬声器。

正如我们在第 14.2.1.1 节中看到的，模拟音频技术和数字音频技术之间的区别恰好对应于信号处理理论研究中连续时间和离散时间信号之间的区别。

14.3.2.1 模数转换：脉冲编码调制

要录制用于数字系统（例如计算机或游戏机）的音频，必须先将模拟音频信号的时变电压转换为数字形式。脉冲编码调制 (PCM) 是对采样模拟声音信号进行编码的标准方法，以便将其存储在计算机内存中、通过数字电话网络传输或刻录到光盘上。

在脉冲编码调制中，电压测量以固定的时间间隔进行。电压测量值可以以浮点格式存储，也可以量化，以便每个测量值可以存储在具有固定位数（通常为 8、16、24 或 32）的整数中。然后将测得的电压值序列存储到内存中的数组中，或写入长期存储介质中。测量单个模拟电压并将其转换为量化数字形式的过程称为模数转换或 A/D 转换。通常使用专用硬件来执行 A/D 转换。当我们以固定的时间间隔重复此过程时，它称为采样。执行 A/D 转换和/或采样的硬件或软件组件称为 A/D 转换器或 ADC。

用数学术语来说，给定连续时间音频信号 p(t)，我们构造采样版本 p[n]，使得对于每个样本，p[n]=p(nTs)，其中 n 是用于索引样本的非负整数，Ts 是每个样本之间的时间量，称为采样周期。采样的基本原理如图 14.25 所示。

PCM采样产生的数字信号有两个重要特性：

- 采样率。这是电压测量的频率

14.3.2 Digital Audio Technology

The introduction of the compact disc (CD) marked a turning point in the audio industry toward digital audio storage and processing. Digital technology opens up a great many new possibilities, from reducing the size and increasing the capacity of storage media, to using powerful computer hardware and software to synthesize and manipulate audio in previously unimagined ways. Today, analog audio storage devices are a thing of the past, and analog audio signals are typically employed only where necessary—at the microphone and the speaker.

As we saw in Section 14.2.1.1, the distinction between analog and digital audio technologies corresponds exactly to the distinction between *continuous-time* and *discrete-time* signals in the study of signal processing theory.

14.3.2.1 Analog-to-Digital Conversion: Pulse-Code Modulation

To record audio for use in a digitial system, such as a computer or game console, the time-varying voltage of an analog audio signal must first be converted into digital form. Pulse-code modulation (PCM) is the standard method for encoding a sampled analog sound signal so that it can be stored in a computer's memory, transmitted over a digital telephony network or burned onto a compact disc.

In pulse-code modulation, voltage measurements are taken at regular time intervals. The voltage measurements may be stored in floating-point format, or they may be *quantized* so that each measurement can be stored in an integer with a fixed number of bits (typically 8, 16, 24 or 32). The sequence of measured voltage values is then stored into an array in memory, or written out to a long-term storage medium. The process of measuring a single analog voltage and converting it to quantized numeric form is called *analog-to-digital conversion* or A/D conversion. Specialized hardware is typically used to perform A/D conversions. When we repeat this process at regular time intervals, it is called *sampling*. A hardware or software component that performs A/D conversion and/or sampling is referred to as an A/D converter or ADC.

In math terms, given the continuous-time audio signal $p(t)$, we construct the *sampled* version $p[n]$ such that for each sample, $p[n] = p(nT_s)$, where n is a non-negative integer used to index the samples, and T_s is the amount of time between each sample, known as the *sampling period*. The basics of sampling are illustrated in Figure 14.25.

The digital signal that results from PCM sampling has two important properties:

- *Sampling rate*. This is the frequency at which the voltage measurements

图 14.25 离散时间信号可以被认为是连续时间信号的采样版本。

（样本）。原则上，只要采样频率是原始信号中最高频率成分的两倍，模拟信号就可以以数字方式记录而不会有任何保真度损失。这个令人震惊且极其有用的事实被称为香农-奈奎斯特采样定理。正如我们在 14.1.2.2 节中看到的，人类只能听到有限频带内的声音（从 20 Hz 到 20 kHz）。因此，所有人类感兴趣的音频信号都是带限的，可以使用略高于 40 kHz 的采样率忠实地记录下来。（语音信号占用的频带更窄，从 300 Hz 到 3.4 kHz，因此数字电话系统仅使用 8 kHz 的采样频率即可。）

• 位深度。这描述了用于表示每个量化电压测量值的位数。量化误差是指将测量电压值四舍五入到最接近的量化值而产生的误差。在其他条件相同的情况下，位深度越大，量化误差越小，因此音频录制质量越高。未压缩音频数据格式的典型位深度为 16。位深度有时也称为分辨率。

香农-奈奎斯特采样定理

香农-奈奎斯特采样定理指出，如果对一个带限连续时间信号（即，在有限频带之外，傅里叶变换处处为零的信号）进行采样，得到其离散时间对应信号，则只要采样率足够高，就可以从离散信号中精确地恢复出原始连续时间信号。满足该关系的最小采样频率称为奈奎斯特频率。

Figure 14.25. A discrete-time signal can be thought of as a sampled version of a continuous-time signal.

(samples) are taken. In principle an analog signal can be recorded digitally without *any* loss of fidelity, provided that it is sampled at a frequency *twice* that of the highest-frequency component present in the original signal. This somewhat astounding and incredibly useful fact is known as the *Shannon-Nyquist sampling theorem*. As we saw in Section 14.1.2.2, humans can only hear sounds within a limited band of frequencies (from 20 Hz to 20 kHz). So all audio signals of interest to human beings are *band-limited*, and can be faithfully recorded using a sampling rate of a little over 40 kHz. (Voice signals occupy a narrower band of frequencies, from 300 Hz to 3.4 kHz, so digital telephony systems can get away with a sampling frequency of only 8 kHz.)

- *Bit depth*. This describes the number of bits used to represent each quantized voltage measurement. *Quantization error* is the error introduced by rounding the measured voltage values to the nearest quantized value. All other things being equal, a greater bit depth results in lower quantization error, and therefore yields a higher-quality audio recording. A bit depth of 16 is typical among uncompressed audio data formats. Bit depth is sometimes referred to as *resolution*.

The Shannon-Nyquist Sampling Theorem

The *Shannon-Nyquist sampling theorem* states that if a *band-limited* continuous-time signal (i.e., a signal whose Fourier transform is zero everywhere outside a limited band of frequencies) is sampled to produce its discrete-time counterpart, the original continuous-time signal can be recovered *exactly* from the discrete signal, provided that the sampling rate is high enough. The minimum sampling frequency for which this relation holds is called the *Nyquist*

频率 。

$$\omega_s > 2\,\omega_{\max},$$

在哪里

$$\omega_s = \frac{2\pi}{T_s}.$$

显然，正是这条定理的存在，才使得数字技术得以应用于音频处理。如果没有它，数字音频注定永远无法达到模拟音频的音质，计算机也无法像今天这样在高保真音频制作中发挥重要作用。

我们不会深入探讨采样定理的原理。但我们可以通过理解以下事实来获得一些启发：以规则的时间间隔对信号进行采样会导致其频谱（傅里叶变换）沿频率轴反复复制。采样频率越高，这些信号频谱副本的“间隔”就越大。因此，如果原始信号是带限的，

图 14.26. 带限信号的频谱在除有限频带外的其他所有地方均为零（上）。如果采样频率超过奈奎斯特频率，频谱副本不会重叠，原始信号可以精确恢复（中）。如果采样频率过低，频谱副本会重叠，从而产生混叠（下）。

frequency.

$$\omega_s > 2\,\omega_{\max},$$

where

$$\omega_s = \frac{2\pi}{T_s}.$$

Clearly, it is the existence of this theorem that allows digital technology to be used in audio processing. Without it, digital audio would be doomed never to sound as good as analog audio, and computers would not be playing the significant role in the production of high-fidelity audio that they do today.

We won't get into all of the gory details of why the sampling theorem works. But we can gain some insight by realizing that the act of sampling a signal at regularly spaced intervals in time causes its frequency spectrum (Fourier transform) to be duplicated over and over along the frequency axis. The higher the sampling frequency, the more "spaced out" these copies of the signal's frequency spectrum will be. So if the original signal is band-limited,

Figure 14.26. The frequency spectrum of a band-limited signal is zero everywhere except within a limited frequency band (top). If the sampling frequency exceeds the Nyquist frequency, the spectrum copies do not overlap and the original signal can be recovered exactly (middle). If the sampling frequency is too low, the spectrum copies overlap and aliasing results (bottom).

如果采样频率足够高，我们可以保证频谱副本之间的距离足够远，从而不会相互重叠。在这种情况下，我们可以通过低通滤波器滤除原始频谱以外的所有频谱副本，从而精确地恢复原始频谱。但是，如果采样频率太低，频谱副本就会相互重叠。这称为混叠，它会阻止我们精确地恢复原始信号的频谱。有关混叠和非混叠采样的说明，请参见图 14.26。

14.3.2.2 数模转换：解调

当要播放数字声音信号时，需要一个与模数转换相反的过程。我们称之为数模转换，简称D/A转换。它也被称为解调，因为它消除了脉冲编码调制的影响。数模转换电路被称为DAC。

D/A 转换硬件会生成与数字信号中每个采样电压值相对应的模拟电压，这些电压值由内存中量化的 PCM 值数组表示。如果我们定期以 PCM 期间测量样本的速率驱动该硬件，并假设采样率根据香农-奈奎斯特采样定理足够高，则产生的模拟电压信号应该与原始电压信号完全匹配。

实际上，当我们用一系列离散电压电平驱动模拟电压电路时，由于硬件试图快速地从一个电压电平切换到另一个电压电平，经常会引入不必要的高频振荡。D/A 硬件通常包含一个低通或带通滤波器来消除这些不必要的振荡，从而确保精确地再现原始模拟信号。有关滤波的更多信息，请参见第 14.2.5.8 节。

14.3.2.3 数字音频格式和编解码器

存在多种数据格式，用于将 PCM 音频数据存储在光盘上或通过互联网传输。每种格式都有其历史和优缺点。某些格式（例如 AVI）实际上是“容器”格式，可以将数字音频信号封装成多种数据格式。

一些音频格式以未压缩的形式存储 PCM 数据。另一些音频格式则利用各种形式的数据压缩来减少所需的文件大小或传输带宽。有些压缩方案是有损的，这意味着在压缩/解压缩过程中会损失原始信号的部分保真度。另一些压缩方案是无损的，这意味着

and if the sampling frequency is high enough, we can guarantee that the copies of the frequency spectrum will be spaced far enough apart so as not to overlap with one another. When this happens, we can recover the original frequency spectrum exactly via a low-pass filter that filters out all of the copies of the spectrum except the original. However, if the sampling frequency is too low, the spectrum copies will overlap with one another. This is called *aliasing*, and it prevents us from exactly recovering the original signal's spectrum. See Figure 14.26 for an illustration of aliased and unaliased sampling.

14.3.2.2 Digital-to-Analog Conversion: Demodulation

When a digital sound signal is to be played back, a process opposite to that of analog-to-digital conversion is required. We call this, sensibly enough, *digital-to-analog conversion* or D/A conversion for short. It is also termed *demodulation* because it undoes the effects of pulse-code modulation. A digital-to-analog conversion circuit is called a DAC.

D/A conversion hardware generates an analog voltage corresponding to each sampled voltage value in a digital signal, as represented by an array of quantized PCM values in memory. If we drive this hardware with new values periodically, at the rate at which the samples were measured during PCM, and presuming that the sample rate was high enough as per the Shannon-Nyquist sampling theorem, the analog voltage signal produced should exactly match the original voltage signal.

Practically speaking, when we drive an analog voltage circuit with a sequence of discrete voltage levels, unwanted high-frequency oscillations are often introduced as the hardware tries to rapidly change from one voltage level to another. D/A hardware typically includes a low-pass or band-pass filter to remove these unwanted oscillations, thereby ensuring an accurate reproduction of the original analog signal. For more information on filtering, see Section 14.2.5.8.

14.3.2.3 Digital Audio Formats and Codecs

Various data formats exist for storing PCM audio data on disc or transmitting it over the Internet. Each format has its history, and its pros and cons. Some formats such as AVI are actually "container" formats, which can encapsulate digital audio signals in more than one data format.

Some audio formats store the PCM data in an uncompressed form. Others utilize various forms of data compression to reduce the required file size or transmission bandwidth. Some compression schemes are *lossy*, meaning that some of the fidelity of the original signal is lost in the compression/decompression process. Other compression schemes are *lossless*, meaning that

经过一次往返压缩/解压缩循环后，可以准确恢复原始 PCM 数据。

让我们来看看几种最常见的音频数据格式。

• 在预先知道信号的元信息（例如采样率和位深度）的情况下，有时会使用原始无头 PCM 数据。

• 线性PCM (LPCM) 是一种未压缩的音频格式，最多可支持八个音频通道，采样频率为 48 kHz 或 96 kHz，每个样本的比特率为 16、20 或 24。LPCM 中的“线性”是指幅度测量采用线性尺度（而不是对数尺度）。

• WAV 是由 Microsoft 和 IBM 创建的一种未压缩的文件格式。它在 Windows 操作系统上很常见。它的正确名称是“波形音频文件格式”，尽管它也很少被称为“Windows 音频”。WAV 文件格式实际上是资源交换文件格式 (RIFF) 系列格式之一。RIFF 文件的内容以块的形式排列，每个块都有一个四字符代码 (FOURCC)，用于定义块的内容和块大小字段。WAV 文件中的比特流符合线性脉冲编码调制 (LPCM) 格式。WAV 文件也可以包含压缩音频，但它们最常用于存储未压缩的音频数据。

• WMA（Windows Media Audio）是微软专有的音频压缩技术，旨在替代 MP3。详情请访问 http://en。
详情请参阅 wikipedia.org/wiki/Windows_Media_Audio。

• AIFF（音频交换文件格式）是由苹果电脑公司开发的一种格式，广泛应用于 Macintosh 电脑。与 WAV/RIFF 文件类似，AIFF 文件通常包含未压缩的 PCM 数据，由多个数据块组成，每个数据块前面都有一个四字符代码和一个大小字段。AIFF-C 是 AIFF 格式的压缩版本。

• MP3 是一种有损压缩音频文件格式，已成为大多数数字音频播放器的实际标准，并广泛应用于游戏、多媒体系统和服务。该格式的全名实际上是 MPEG-1 或 MPEG-2 音频第三层。MP3 压缩可以使文件大小缩小十分之一，但与原始未压缩音频的感知差异却非常小。这种效果是通过利用感知编码实现的——这种技术可以消除音频信号中大多数人无法感知的部分。

the original PCM data can be recovered exactly after a round-trip compression/decompression cycle.

Let's take a look at a few of the most common audio data formats.

- *Raw header-less PCM* data is sometimes used in situations where the meta-information about the signal, such as the sample rate and bit depth, is known a priori.
- *Linear PCM* (LPCM) is an uncompressed audio format that can support up to eight channels of audio at a 48 kHz or 96 kHz sampling frequency, and 16, 20 or 24 bits per sample. The "linear" in LPCM refers to the fact that the amplitude measurements are taken on a linear scale (as opposed to, say, a logarithmic scale).
- *WAV* is an uncompressed file format created by Microsoft and IBM. Its use is commonplace on the Windows operating system. Its correct name is "waveform audio file format" although it is also rarely referred to as "audio for windows." The WAV file format is actually one of a family of formats known as *resource interchange file format* (RIFF). The contents of a RIFF file are arranged in chunks, each with a four-character code (FOURCC) that defines the contents of the chunk and a chunk size field. The bitstream in a WAV file conforms to the linear pulse-code modulation (LPCM) format. WAV files can also contain compressed audio, but they are most commonly used for storing uncompressed audio data.
- *WMA* (Windows Media Audio) is proprietary audio compression technology designed by Microsoft as an alternative to MP3. See http://en.wikipedia.org/wiki/Windows_Media_Audio for details.
- *AIFF* (audio interchange file format) is a format developed by Apple Computer, Inc. and used widely on Macintosh computers. Like a WAV/RIFF file, an AIFF file typically contains uncompressed PCM data, and is comprised of chunks, each prefaced by a four-character code and a size field. AIFF-C is a compressed variant of the AIFF format.
- *MP3* is a lossy compressed audio file format that has become the de facto standard on most digital audio players, and is also widely used by games and multimedia systems and services. The full name of this format is actually MPEG-1 or MPEG-2 audio layer III. MP3 compression can result in files that are one-tenth the size, but with very little perceptual difference from the original uncompressed audio. These results are achieved by making use of *perceptual coding*—a technique that eliminates portions of the audio signal that are beyond the perception of most people anyway.

• ATRAC 代表自适应变换声学编码 (Adaptive Transform Acoustic Coding)，这是索尼开发的一系列专有音频压缩技术。该格式最初开发的目的是为了让索尼的 MiniDisc 介质能够存储与 CD 播放时间相同的音频，同时占用更少的空间，并且音质几乎不会下降。更多详情，请参阅 http://en.wikipedia.org/wiki/Adaptive_Transform_Acoustic_Coding。

• Ogg Vorbis 是一种提供有损压缩的开源文件格式。
Ogg 是一种“容器”格式，通常与 Vorbis 数据格式结合使用。

• 杜比数字 (AC-3) 是一种有损压缩格式，支持从单声道到 5.1 环绕声的声道格式。

• DTS 是由 DTS, Inc. 开发的剧院音频技术集合。DTS Coherent Acoustics 是一种可通过 S/PDIF 接口传输的数字音频格式（参见第 14.3.2.5 节），可用于 DVD 和激光影碟。

• VAG 是一种专有音频文件格式，可供所有 PlayStation 3 开发者使用。它采用自适应差分 PCM (ADPCM)，这是一种基于 PCM 的模数转换方案。差分 PCM (DPCM) 存储的是样本之间的差异，而不是样本本身的绝对值，以便更有效地压缩信号。自适应 DPCM 会动态改变采样率，以进一步提高可实现的压缩比。

• MPEG-4 SLS、MPEG-4 ALS 和 MPEG-4 DST 是提供无损压缩的格式。

这份清单并非面面俱到。事实上，音频文件格式种类繁多，压缩/解压缩算法更是数不胜数。想要了解音频数据格式的奇妙世界，请访问我们的老朋友维基百科：http://en.wikipedia.org/wiki/Digital_audio_format。“PlayStation 3 Secrets”网站也提供了一些关于音频格式的精彩信息：https://bit.ly/2HOVtvR。

14.3.2.4 并行和交错音频数据

组织多声道音频数据的一种方法是将每个单声道的样本存储在单独的缓冲区中。在这种情况下，您需要六个并行的缓冲区来描述 5.1 音频信号。此结构如图 14.27 所示。

多通道音频数据也可以在单个缓冲区内进行交错。在这种情况下，每个时间索引的所有样本都会被分组在一个预

- *ATRAC* stands for Adaptive Transform Acoustic Coding—a family of proprietary audio compression techniques developed by Sony. The format was originally developed to allow Sony's MiniDisc media to contain audio with the same running time as a CD while occupying significantly less space and undergoing an imperceptable degradation in quality. See http://en.wikipedia.org/wiki/Adaptive_Transform_Acoustic_Coding for more details.
- *Ogg Vorbis* is an open source file format that offers lossy compression. Ogg refers to a "container" format that is commonly used in conjunction with the Vorbis data format.
- *Dolby Digital (AC-3)* is a lossy compression format supporting channel formats from mono to 5.1 surround sound.
- *DTS* is a collection of theater audio technologies developed by DTS, Inc. DTS Coherent Acoustics is a digital audio format transportable through S/PDIF interfaces (see Section 14.3.2.5) and used on DVDs and Laserdiscs.
- *VAG* is a proprietary audio file format available for use by all PlayStation 3 developers. It makes use of *adaptive differential PCM* (ADPCM), an analog-to-digital conversion scheme based on PCM. Differential PCM (DPCM) stores the deltas between samples rather than the absolute values of the samples themselves, in order to allow the signal to be compressed more effectively. Adaptive DPCM varies the sample rate dynamically in order to further improve the achievable compression ratio.
- *MPEG-4 SLS, MPEG-4 ALS and MPEG-4 DST* are formats that offer lossless compression.

This list is by no means comprehensive. In fact, there are a dizzying number of audio file formats, and an even longer list of compression/decompression algorithms. For an introduction to the fascinating world of audio data formats, check out our old friend Wikipedia: http://en.wikipedia.org/wiki/Digital_audio_format. The "PlayStation 3 Secrets" website also provides some excellent information on audio formats: https://bit.ly/2HOVtvR.

14.3.2.4 Parallel and Interleaved Audio Data

One way to organize multi-channel audio data is to store the samples for each monophonic channel into a separate buffer. In this case, you'd need six parallel buffers to describe a 5.1 audio signal. This arrangement is shown in Figure 14.27.

Multi-channel audio data can also be *interleaved* within a single buffer. In this case, all of the samples for each time index are grouped together in a pre-

图 14.28. 交错格式的六通道 (5.1) PCM 总线数据。

定义的顺序。图 14.28 描绘了一个包含六通道（5.1）音频信号的交错 PCM 缓冲区。

14.3.2.5 数字接线和连接器

S/PDIF（索尼/飞利浦数字互连格式）是一种以数字方式传输音频信号的互连技术，从而消除了模拟线路引入噪音的可能性。S/PDIF 标准可以通过同轴电缆连接（也称为 S/PDIF）或光纤连接（称为 TOSLINK）在物理上实现。

无论物理接口是什么（S/PDIF 同轴或 TOSLINK 光纤），S/PDIF 传输协议都仅限于传输 2 通道 24 位 LPCM 无压缩音频，标准采样率范围为 32 kHz 至 192 kHz。然而，并非所有设备都能在所有采样率下工作。相同的物理接口也可用于传输比特流编码音频（例如，杜比数字或 DTS 有损压缩数据），杜比数字的比特率范围为 32 kbps 至 640 kbps，DTS 的比特率范围为 768 kbps 至 1536 kbps。

未压缩的多声道 LPCM（即超过两个立体声声道）只能通过消费级音频设备上的 HDMI（高清多媒体接口）连接发送。HDMI 接口用于传输未压缩的数字视频以及压缩或未压缩的数字音频信号。HDMI 支持高达 36.86 Mbps 的多声道或比特流音频比特率。但是，HDMI 的音频比特率会因视频模式而异——只有 720p/50 Hz 或更高的模式才能充分利用音频带宽。有关这方面的更多信息，请参阅“视频依赖性”部分下的 HDMI 规范。Apple 的 DisplayPort 和 Thunderbolt 接口是其他高带宽替代方案，在许多方面与 HDMI 类似。

USB 连接有时用于发送音频信号。在大多数游戏机上，USB 输出仅用于驱动耳机。

无线音频连接也是可行的。蓝牙标准是

图 14.27. 并行格式的六通道 (5.1) PCM 总线数据。

Figure 14.28. Six-channel (5.1) PCM bus data in interleaved format.

defined order. Figure 14.28 depicts an interleaved PCM buffer containing a six-channel (5.1) audio signal.

14.3.2.5 Digital Wiring and Connectors

S/PDIF (Sony/Philips Digital Interconnect Format) is an interconnect technology that transmits audio signals *digitally*, thereby eliminating the possibility of noise being introduced by analog wiring. The S/PDIF standard is physically realized either via a coaxial cable connection (also called S/PDIF) or a fiber-optic connection (known as TOSLINK).

Regardless of the physical interface (S/PDIF coaxial or TOSLINK optical), the S/PDIF transport protocol is limited to 2-channel 24-bit LPCM uncompressed audio at standard sampling rates ranging from 32 kHz to 192 kHz. However, not all equipment works at all sample rates. The same physical interfaces can be also be used to transport bitstream-encoded audio (e.g., Dolby Digital or DTS lossy compressed data) at bitrates ranging from 32 kpbs to 640 kbps for Dolby Digital and 768 kbps to 1536 kbps for DTS, respectively.

Uncompressed multi-channel LPCM (i.e., greater than two stereo channels) can only be sent over an HDMI (high-definition multimedia interface) connection on consumer audio equipment. HDMI connectors are used for transmission of both uncompressed digital video and either compressed or uncompressed digital audio signals. HDMI supports up to a 36.86 Mbps bitrate for multi-channel or bitstream audio. However, HDMI bitrates for audio vary depending on the video mode—only 720 p/50 Hz modes or higher are capable of utilizing the full audio bandwidth. See the HDMI specification under the section “video dependency” for more information on this. Apple’s DisplayPort and Thunderbolt connectors are other high-bandwidth alternatives similar in many respects to HDMI.

USB connections are sometimes used to send audio signals. On most game consoles, the USB output is intended only to drive headphones.

Wireless audio connections are also possible. The Bluetooth standard is the

Figure 14.27. Six-channel (5.1) PCM bus data in parallel format.

最常用的无线传输音频信号的方法。

14.4 以 3D 形式渲染音频

到目前为止，我们已经学习了声音的物理原理、信号处理的数学原理以及用于录制和播放声音的各种技术。在本节中，我们将探索如何将这些理论和技术应用于游戏引擎，从而为我们的游戏制作逼真、沉浸式的音景。

任何在虚拟 3D 世界中发生的游戏都需要某种 3D 音频渲染引擎。高质量的 3D 音频系统应该为玩家提供丰富、身临其境且逼真的音效，与 3D 世界中发生的事情相符，同时支持故事情节并忠实于游戏的音效设计。

- 该系统的输入是来自游戏世界的无数 3D 声音：脚步声、讲话声、物体相互碰撞的声音、枪声、风声或雨声等环境声音等等。

- 它的输出是少量的声道，当在扬声器中播放时，尽可能真实地再现玩家在虚拟游戏世界中实际听到的声音。

理想情况下，我们希望音频引擎能够输出完整的 7.1 或 5.1 环绕声，因为这能为耳朵提供尽可能丰富的位置提示。然而，音频引擎也必须支持立体声输出，以满足那些没有高端家庭影院系统的玩家的需求，或者那些只想戴着耳机玩游戏以免吵醒邻居的玩家的需求。

游戏的音频引擎还负责播放虚拟世界之外的声音。例如，音乐曲目、游戏菜单系统的声音、旁白的画外音、玩家角色的声音（尤其是在第一人称射击游戏中），以及某些环境声音。我们称之为 2D 声音。这类声音旨在与 3D 空间化引擎的输出混合后，直接在扬声器中播放。

14.4.1 3D声音渲染概述

3D音频引擎执行的主要任务如下：

most commonly used method of transmitting audio signals wirelessly.

14.4 Rendering Audio in 3D

Thus far, we've learned about the physics of sound, the mathematics of signal processing and the various technologies that are used to record and play back sounds. In this section, we'll explore how all of this theory and technology can be put to use in a game engine, in order to produce realistic, immersive soundscapes for our games.

Any game that takes place in a virtual 3D world requires some sort of *3D audio rendering engine*. A high-quality 3D audio system should provide the player with a rich, immersive and believable soundscape that matches what's going on in this 3D world, while supporting the story and remaining true to the tonal design of the game.

- The inputs to this system are the myriad *3D sounds* that emanate from all over the game world: footsteps, speech, the sound of objects bumping into one another, gunfire, ambient sounds like wind or rainfall and so on.

- Its output is a handful of sound channels that, when played in the speakers, reproduce as believably as possible what the player would actually hear if he or she were really there in the virtual game world.

Ideally we'd like our audio engine to produce its output in full 7.1 or 5.1 surround sound, because this gives the ears the richest possible set of positional cues. However, audio engines must also support stereo output for players who don't have fancy home theater systems—or who just want to play their game using headphones so they don't wake their neighbors.

A game's audio engine is also responsible for playing sounds that do not originate in the virtual world. Examples include the music track, sounds made by the in-game menu system, a narrator's voice-over, the voice of the player character (especially in first-person shooters) and possibly certain ambient sounds. We call these *2D sounds*. Such sounds are designed to be played "directly" in the speakers, after having been mixed with the outputs of the 3D spatialization engine.

14.4.1 Overview of 3D Sound Rendering

The primary tasks performed by the 3D audio engine are as follows:

• 声音合成是生成与游戏世界中发生的事件相对应的声音信号的过程。这些声音信号可能是通过播放预先录制的声音片段产生的，也可能是在运行时由程序生成的。

• 空间化营造出一种错觉，让每个 3D 声音从聆听者的视角，仿佛来自游戏世界中的正确位置。空间化是通过控制每个声波的振幅（即增益或音量）来实现的，具体方式有两种：

◦ 基于距离的衰减控制声音的整体音量，以指示其与听众的径向距离。

◦ 声像控制每个可用扬声器中声音的相对音量，以指示声音到达的方向。

• 声学建模通过模拟聆听空间的早期反射和后期混响，并考虑声源与聆听者之间路径的部分或全部阻挡，增强了渲染音景的真实感。一些声音引擎还会模拟大气吸收（第 14.1.3.2 节）和/或 HRTF 效应（第 14.1.4 节）的频率相关效应。

• 还可以应用多普勒频移来解释声源和听众之间的任何相对运动。

• 混音是控制游戏中所有 2D 和 3D 声音相对音量的过程。混音过程部分由物理原理驱动，部分由游戏音效设计师的审美选择决定。

14.4.2 音频世界建模

为了渲染虚拟世界的音景，我们必须首先向引擎描述这个世界。“音频世界模型”由以下元素组成：

• 3D 声源。游戏世界中的每个 3D 声音都由单声道音频信号组成，从特定位置发出。我们还必须为引擎提供其速度、辐射模式（全向、锥形、平面）和范围（超出该范围声音将无法听到）。

• 监听器。监听器是位于游戏世界中的“虚拟麦克风”。它由其位置、速度和方向定义。

- *Sound synthesis* is the process of generating the sound signals that correspond to the events occurring in the game world. These might be produced by playing back pre-recorded sound *clips*, or they might be *procedurally* generated at runtime.
- *Spatialization* produces the illusion that each 3D sound is coming from the proper location in the game world, from the point of view of the listener. Spatialization is accomplished by controlling the *amplitude* of each sound wave (i.e., its gain or volume) in two ways:
  - *Distance-based attenuation* controls the *overall volume* of a sound in order to provide an indication of its *radial distance* from the listener.
  - *Pan* controls a sound's relative volume in each of the available speakers in order to provide an indication of *direction* from which the sound is arriving.
- *Acoustical modeling* heightens the realism of the rendered soundscape by mimicking the early reflections and late reverberations that characterize the listening space, and by accounting for the presence of obstacles that partially or completely block the path between the sound source and the listener. Some sound engines also model the frequency-dependent effects of *atmospheric absorption* (Section 14.1.3.2) and/or HRTF effects (Section 14.1.4).
- *Doppler shifting* may also be applied to account for any relative movement between a sound source and the listener.
- *Mixing* is the process of controlling the relative volumes of all the 2D and 3D sounds in our game. The mix is driven in part by physics and in part by aesthetic choices made by the game's sound designers.

14.4.2 Modeling the Audio World

In order to render the soundscape of a virtual world, we must first describe that world to the engine. The "audio world model" consists of the following elements:

- *3D sound sources*. Each 3D sound in the game world consists of a monophonic audio signal, emanating from a specific *position*. We must also provide the engine with its *velocity*, *radiation pattern* (omnidirectional, conical, planar) and *range* (beyond which the sound is inaudible).
- *Listener*. The listener is a "virtual microphone" located in the game world. It is defined by its *position*, *velocity* and *orientation*.

• 环境模型。该模型要么描述虚拟世界中存在的表面和物体的几何形状和属性，要么描述游戏发生的聆听空间的声学特性。

基于距离的衰减会用到声源和听者的位置；声源的辐射模式也会影响基于距离的衰减计算。听者的方位定义了一个参考系，声音的角位置就是在这个参考系中计算的。这个角度反过来又决定了声像——5.1 或 7.1 环绕声的五个或七个主扬声器中声音的相对音量。应用多普勒频移时会用到声源和听者的相对速度。最后但同样重要的是，环境模型用于对聆听空间的声学效果进行建模，并解释声路的部分或完全阻塞情况。

14.4.3 基于距离的衰减

基于距离的衰减会随着 3D 声音与听众之间的径向距离的增加而降低 3D 声音的音量。

14.4.3.1 衰减最小值和最大值

典型的游戏世界中，声源数量非常庞大。由于硬件和 CPU 带宽的限制，我们不可能渲染所有声源。而且我们也不想这么做，因为基于距离的衰减会导致距离听者一定距离以外的所有声音都无法被听到。因此，每个声源通常都标注有衰减 (FO) 参数。

衰减最小值（简称“FO 最小值”）是一个最小半径，我们将其表示为 r 最小值，在此半径内，声音完全不会衰减，并且音量可以完全听到。衰减最大值或“FO 最大值”是一个最大半径，表示为 r 最大值，超过此半径，声源将被视为静音，因此可以忽略。在 FO 最小值和 FO 最大值之间，我们需要将音量从完全衰减平滑地混合到零。

14.4.3.2 混合至零

将音量从最大音量逐渐降至零的一种方法是在 FO min 和 FO max 之间使用线性渐变。根据声音的类型，线性衰减可能听起来不错。

在 14.1.3.1 节中，我们了解到声音强度与我们对“响度”的感知密切相关，并且会随着径向距离的增加而下降，具体规律如下：

- *Environmental model*. This model either describes the *geometry* and *properties* of the surfaces and objects present in the virtual world, and/or it describes the *acoustic properties* of the listening spaces in which gameplay takes place.

The *positions* of the source and listener are used for *distance-based attenuation*; the *radiation pattern* of the sound source also factors into the distance-based attenuation calculation. The *orientation* of the listener defines a reference frame in which the *angular position* of the sound is calculated. This angle in turn determines the *pan*—the relative volumes of the sound in the five or seven main speakers of 5.1 or 7.1 surround sound, respectively. The *relative velocity* of source and listener is used when applying a *Doppler shift*. And last but not least, the *environmental model* is used for modeling the acoustics of the listening space and to account for partial or complete blockage of the sound path.

14.4.3 Distance-Based Attenuation

Distance-based attenuation reduces the volume of a 3D sound as the radial distance between it and the listener increases.

14.4.3.1 Fall-Off Min and Max

The number of sound sources in a typical game world is very large. Due to hardware and CPU bandwidth limitations, we couldn't possibly render them all. And we wouldn't want to, because thanks to distance-based fall-off all sounds beyond a certain distance from the listener can't be heard anyway. For this reason, each sound source is usually annoted with fall-off (FO) parameters.

The *fall-off min* ("FO min" for short) is a minimum radius, which we'll denote r_{min}, within which the sound doesn't fall off at all and is heard at full volume. The *fall-off max* or "FO max" is a maximum radius, denoted r_{max}, beyond which the sound source is considered to be silent and can therefore be ignored. Between the FO min and FO max, we need to blend smoothly from full volume down to zero.

14.4.3.2 Blending to Zero

One way to blend from maximum volume down to zero is to use a linear ramp between FO min and FO max. Depending on the type of sound, a linear fall-off might sound just fine.

In Section 14.1.3.1, we learned that sound intensity, which is closely related to our perception of "loudness," falls off with radial distance according to a

1/ r² 规则。增益与声压振幅成正比，其衰减速度为 1/ r。因此，正确的做法是使用 1/ r 曲线将声音的增益从最大音量逐渐衰减至零。

函数 1/r 的一个问题是它是渐近函数——无论 r 有多大，它都永远不会完全趋近于零。我们可以通过将曲线稍微向下移动，使其与 r 轴相交于 r max 来解决这个问题。或者，我们可以简单地将声音强度限制为零，使所有 r > r max 的 r 都为零。

14.4.3.3 打破规则

在制作《最后生还者》时，顽皮狗的音效部门发现，使用 1/r2 规则衰减角色对话会导致距离较近的角色的对话过快变得难以理解。这是一个严重的问题，尤其是在潜行环节，因为在潜行环节中，听清敌人周围的对话既是重要的战术手段，也是推进剧情的手段。

为了解决这个问题，顽皮狗的音效部门采用了一种复杂的衰减曲线，使对话在靠近听者时衰减得更慢，在中间区域衰减得更快，然后随着与听者的距离越来越远而再次衰减。这使得对话在更远的距离也能被听到，同时仍然保持自然的衰减效果。

对话衰减曲线也会根据游戏当前的“紧张程度”（例如，敌人是否察觉不到玩家、正在寻找玩家，还是正在与玩家直接交战）在运行时进行动态调整。这使得《最后生还者》中的声音在潜行游戏中能够传递到更远的距离，同时又不会在战斗爆发时变得过于强烈。

最后，可以选择启用“甜化”混响，让角色的声音在拐角处也能清晰地传递，即使直接路径完全被遮挡。当模拟真实的衰减效果比确保玩家能够清晰听到对话更重要时，这个工具非常有用。

在设计 3D 音频模型时，有各种各样的“作弊”方法。
但无论你做什么，都要记住这个简单的道理：永远不要害怕做任何事来满足游戏的需求。别担心——物理定律不会被触犯！

14.4.3.4 大气衰减

正如我们在 14.1.3.2 节中看到的，低音被大气衰减的程度小于高音。一些游戏，包括顽皮狗的《最后生还者》，通过对每个声道应用低通滤波器来模拟这种现象。

$1/r^2$ rule. Gain, which is proportional to the amplitude of the sound pressure, falls of as $1/r$. So really the right thing to do is to use a $1/r$ curve to blend the gain of a sound from full volume down to zero.

One problem with the function $1/r$ is that it is asymptotic—it never quite reaches zero, no matter how large r gets. We can fix this by shifting the curve slightly downward so that it crosses the r axis at $r_{\max}$. Or we can simply clamp the sound intensity to zero for all $r > r_{\max}$.

14.4.3.3 Bending the Rules

When making *The Last of Us*, Naughty Dog's sound department discovered that attenuating character dialog using the $1/r^2$ rule caused speech to become unintelligible too quickly for characters that were only a modest distance away. This was a serious problem, especially during the stealth sections of gameplay, where hearing the enemies' ambient conversations was important both as a tactical tool and as a means of advancing the storyline.

To solve this problem, the sound department at Naughty Dog utilized a sophisticated fall-off curve that causes dialog to roll off more slowly near the listener, more quickly in the mid-range, and then more slowly again as the distance to the listener grows very large. This allows speech to be audible over longer distances, while still retaining a natural-sounding fall-off.

The dialog fall-off curves were also adjusted dynamically at runtime, based on the current "tension level" of the game (i.e., whether the enemies are unaware of the player, are searching for him or are engaged in direct combat with him). This is what allows the voices in *The Last of Us* to project over longer distances during stealth gameplay, while not rising to overpowering levels when combat breaks out.

Finally, a "sweetener" reverb could be optionally enabled to allow character voices to bleed around corners, even when the direct path is 100% obstructed. This tool is incredibly helpful in situations where modeling realistic fall-off is less important than ensuring that the player can hear a conversation clearly.

There are all sorts of ways to "cheat" when designing your 3D audio model. But no matter what you do, always remember this simple lesson: Never be afraid to do whatever it takes to satisfy the *needs of your game*. Don't worry—the laws of physics won't be offended!

14.4.3.4 Atmospheric Attenuation

As we saw in Section 14.1.3.2, low-pitched sounds are attenuated by the atmosphere less than high-pitched sounds. Some games, including Naughty Dog's *The Last of Us*, model this phenomenon by applying a low-pass filter to each

3D 声音，其通带随着声源和听众之间的距离的增加而向越来越低的频率滑动。

14.4.4 平移

声像平移是一种技术，用于营造3D声音来自特定方向的错觉。通过控制每个可用扬声器的音量（即增益），我们可以在三维空间中产生声音的幻像。这种声像平移方法被称为振幅平移，因为我们通过仅调整每个扬声器产生的声波振幅来向听众提供角度信息（而不是使用相位偏移、混响或滤波来提供位置提示）。它有时也被称为独立同分布声像平移，因为它依赖于双耳强度差 (IID) 的感知效应来产生声音的幻像。

“声像”（pan）一词源于早期技术，当时人们使用“全景电位器”（可变电阻）或“声像电位器”来控制立体声系统左右扬声器的相对音量。将声像电位器拨到一端，只会在左扬声器中产生声音；拨到另一端，则会完全驱动右扬声器；而将声像电位器拨盘置于中间位置，则会将声音均匀地分配到两个扬声器中。

为了理解声像的工作原理，我们设想听众位于一个圆心。扬声器位于圆周的不同位置，因此本书将其称为扬声器圆。圆的半径近似于听众与任意一个扬声器之间的平均距离。

对于立体声系统，前置和右置扬声器的位置大致位于中心左右两侧±45度。对于立体声耳机，它们的位置大致位于±90度（半径要小得多）。对于7.1环绕声，我们只考虑七个“主”扬声器，因为LFE声道不提供位置提示。这些扬声器的位置大致如图14.29所示。当声像平移到5.1系统时，我们只需省略环绕左和环绕右扬声器即可。

暂时，我们将每个 3D 声音视为一个点源。要对声音进行声像平移，我们首先确定其方位角（水平角）。方位角必须在听者的局部空间中进行测量，因此零度角对应于听者正前方的位置。接下来，我们找出圆周上与该方位角相邻的两个扬声器。我们将该角度转换为两个扬声器之间弧度的百分比。最后，我们使用此百分比来确定

3D sound whose passband slides toward lower and lower frequencies as the distance between the sound source and the listener increases.

14.4.4 Pan

Panning is a technique used to provide the illusion that a 3D sound is coming from a particular direction. By controlling the volume (i.e., gain) of the sound in each of the available speakers, we can induce the perception of a *phantom image* of the sound in three-dimensional space. This method of panning is called *amplitude panning* because we are providing angular information to the listener by adjusting only the amplitudes of the sound waves produced at each speaker (as opposed to using phase offsets, reverb or filtering to provide positional cues). It is sometimes referred to as *IID panning* because it relies on the perceptual effects of interaural intensity difference (IID) to produce a sound's phantom image.

The term "pan" comes from early technology that used a "panoramic potentiometer" (variable resistor) or "pan pot" to control the relative volumes of the left and right speakers of a stereo system. Dialing the pan pot to one extreme would produce sound only in the left speaker; dialing it to the other extreme would drive the right speaker exclusively; and centering the pan pot dial would distribute the sound equally to both speakers.

To understand how pan works, we envision our listener located at the center of a circle. The speakers are positioned at various points on the circumference of this circle, so we'll call it the *speaker circle* in this book. The radius of the circle approximates the average distance between the listener and any one speaker.

For a stereo sound system, the front and right speakers are located roughly at ±45 degrees to the left and right of center. For stereo headphones, they are positioned at ±90 degrees (and the radius is much smaller). For 7.1 surround sound, we consider only the seven "main" speakers, as the LFE channel provides no positional cues. These speakers are located roughly as shown in Figure 14.29. When panning to a 5.1 system, we simply omit the surround left and surround right speakers.

For the time being, let's treat each 3D sound as a point source. To pan a sound, we first determine its *azimuthal* (horizontal) angle. The azimuthal angle must be measured in the local space of the listener, so that an angle of zero corresponds to the position directly in front of the listener. Next, we figure out which two speakers around the circumference of our circle are *adjacent* to this azimuthal angle. We convert the angle into a percentage of the arc between the two speakers. Finally, we use this percentage to determine the gains of the

图 14.29.7.1 声道扬声器布局。

图 14.30. 将声音视为点源，计算与声源紧邻的两个扬声器之间的声像混合百分比 β。

每个扬声器都有声音。

为了用数学公式来表达这一点，我们用符号 θ s 表示声音的方位角。我们将两个相邻扬声器的角度分别称为 θ 1 和 θ 2 。百分比 β 的计算方法如下：

$$\beta = \frac{\theta_s - \theta_1}{\theta_2 - \theta_1}.$$

图 14.30 说明了这一计算。

Figure 14.29. Speaker layout for 7.1 pan.

Figure 14.30. Treating the sound as a point source, the pan blend percentage β is calculated between the two speakers immediately adjacent to the source.

sound in each speaker.

To formulate this mathematically, let's use the symbol θ_s for the azimuthal angle of the sound. We'll call the angles of the two adjacent speakers θ_1 and θ_2. The percentage β is then calculated as follows:

$$\beta = \frac{\theta_s - \theta_1}{\theta_2 - \theta_1}.$$

This calculation is illustrated in Figure 14.30.

14.4.4.1 恒定增益平移

我们的第一反应可能是使用百分比 β 在两个扬声器的增益之间进行简单的线性插值。给定未声像化声音的增益 A，则该声音在每个扬声器中播放时的增益计算如下：

$$A_1 = (1-\beta)A;$$
$$A_2 = \beta A.$$

这被称为恒定增益平移，因为净增益 A = A 1 + A 2 是恒定的，与 θ s 和 β 的值无关。
恒定增益声像的主要问题是，当声音在声场中移动时，它无法产生恒定的响度感知。增益控制声压波的幅度，从而控制声压级 (SPL)。然而，正如我们在 14.1.2 节中了解到的，人类对响度的感知实际上与声波的强度或功率成正比，而强度或功率都随 SPL 的平方而变化。

为了说明这个问题，假设我们的声音被声像移动到两个扬声器的中间位置。恒定增益声像会将 A 1 和 A 2 的增益分别设置为 1 2 A。但这会产生总功率 A 2 1 + A 2 2 = (1 2 A) 2 + (1 2 A) 2 = 1 2 A 2 。换句话说，如果声音仅被声像移动到左扬声器或右扬声器，那么声音的响度将只有原来的一半。

14.4.4.2 恒功率泛定律

显然，为了在声像在听者周围移动时保持响度感知的恒定，我们需要保持功率恒定。这条规则被称为恒定功率声像定律，或简称为声像定律。

有一个非常简单的方法可以实现恒定功率平移定律。我们不使用线性插值来计算增益，而是使用混合百分比 β 的正弦和余弦来计算它们：

$$A_1 = \sin(\tfrac{\pi}{2}\beta)A;$$
$$A_2 = \cos(\tfrac{\pi}{2}\beta)A.$$

再次考虑声像，其声像平移到两个扬声器的中间位置（β = 1 2）。在恒定功率声像平移下，两个扬声器的增益将设置为 A 1 = A 2 = 1 √ 2 A 。这得出总功率为 A 2 1 + A 2 2 = (1 √ 2 A) 2 + (1 √ 2 A) 2 = A 2 。这适用于任何 β 值，因此无论我们的声像位于圆周上的哪个位置，功率 A 2 都是恒定的。

14.4.4.1 Constant Gain Panning

Our first instinct might be to use the percentage β to perform a simple linear interpolation between the gains of the two speakers. Given the gain A of the unpanned sound, the gains of that sound as played in each speaker would be calculated as follows:

$$A_1 = (1-\beta)A;$$
$$A_2 = \beta A.$$

This is known as *constant gain* panning, because the net gain $A = A_1 + A_2$ is constant, independent of the values of θ_s and β.

The main problem with constant gain panning is that it does *not* produce the perception of constant *loudness* as the sound moves around the acoustic field. Gain controls the amplitude of the sound pressure wave, and therefore controls the *sound pressure level* (SPL). However, as we learned in Section 14.1.2, human perception of loudness is actually proportional to the *intensity* or *power* of a sound wave, both of which vary as the *square* of the SPL.

As an illustration of the problem, imagine that our sound is panned to the halfway point between our two speakers. Constant gain panning would have us set the gains A_1 and A_2 to $\frac{1}{2}A$ each. But this yields a total power of $A_1^2 + A_2^2 = (\frac{1}{2}A)^2 + (\frac{1}{2}A)^2 = \frac{1}{2}A^2$. In other words, the loudness of the sound will be *one-half* of what it would have been, had the sound been panned to only the left or the right speaker.

14.4.4.2 The Constant Power Pan Law

Clearly in order to keep the perception of loudness constant as a sound's image moves about the listener, we need to keep the *power* constant. This rule is known as the *constant power pan law*, or just the *pan law* for short.

There's a very easy way to implement the constant power pan law. Instead of linearly interpolating the gains, we use the sine and cosine of the blend percentage β to calculate them:

$$A_1 = \sin(\tfrac{\pi}{2}\beta)A;$$
$$A_2 = \cos(\tfrac{\pi}{2}\beta)A.$$

Consider again a sound image that is panned to halfway between the two speakers ($\beta = \frac{1}{2}$). With constant power panning, the two speakers' gains will be set to $A_1 = A_2 = \frac{1}{\sqrt{2}}A$. This yields a total power of $A_1^2 + A_2^2 = (\frac{1}{\sqrt{2}}A)^2 + (\frac{1}{\sqrt{2}}A)^2 = A^2$. This works for any value of β, so the power A^2 is constant no matter where our sound image is placed around the circle.

音响设计师经常应用“3 dB 规则”来解释声像定律：如果要将声音均匀地混合到两个扬声器，则每个扬声器的增益应相对于声音混合时使用的增益降低 3 dB 。
只能在一个扬声器中播放。值 − 3 dB 的出现是因为 log 10 1 √ 2 ≈ − 0.15 。电压增益（或幅度增益）定义为 20 log 10 (A 输出 / A 输入)，且 20 × − 0.15 = − 3 dB。（对数前面的 20 是因为分贝是贝尔的十分之一，乘以 2 是为了说明我们处理的是 A 2 而不是 A 。）

14.4.4.3 净空

在某些情况下，声像平移会导致声音完全由一个扬声器呈现，而在其他情况下，则需要两个（或更多，正如我们稍后会看到的）扬声器来呈现。假设一个声音由两个相邻的扬声器等量播放，并且音量非常大，以至于每个扬声器都输出了最大功率。当该声音仅平移到一个扬声器时会发生什么？答案是，我们可能会损坏这个扬声器，因为恒定功率声像平移定律要求我们对一个扬声器使用的增益要高于两个扬声器。

为了避免这个问题，我们需要人为地降低所有声音的最大增益，这样在最坏的情况下，一个扬声器播放声音时就不会过载。人为地降低最大音量范围的做法被称为“留出一些动态余量”。

动态余量的概念也适用于混音。当两个或多个声音混合时，它们的振幅会叠加。通过在混音中留出一些动态余量，我们可以应对同时播放大量高音量声音的最坏情况。

14.4.4.4 居中还是不居中？

在电影院中，中央声道历来用于播放对话；只有音效才会被平移到房间周围的其他扬声器。这种做法背后的理念是，电影中的角色说话时通常出现在屏幕上，因此观众希望听到他们的声音位于屏幕正中央。这种方法还有一个好处，就是将对话与影片中的其他声音分离，这意味着响亮的音效不会占用所有可用的动态余量并淹没对话。

在 3D 游戏中，情况则大不相同。玩家通常希望听到来自自己周围“正确”位置的对话。如果玩家将镜头旋转 180 度，对话也应该围绕

Sound designers often apply a “3 dB rule” to account for the pan law: If a sound is to be mixed equally to two speakers, the gain in each speaker should be reduced by 3 dB relative to the gain that would be used if the sound were to be played in only one speaker. The value -3 dB arises because $\log_{10}\left(\frac{1}{\sqrt{2}}\right) \approx -0.15$. Voltage gain (or amplitude gain) is defined as $20\log_{10}(A_{\text{out}}/A_{\text{in}})$, and $20 \times -0.15 = -3$ dB. (The 20 in front of the logarithm arises because a decibel is one-tenth of a bel, multiplied by two to account for the fact that we're dealing with A^2 and not A.)

14.4.4.3 Headroom

Panning causes sounds to be rendered entirely by one speaker in some situations, and by two (or more, as we'll see) speakers in others. Let's say a sound is being played equally by two adjacent speakers, and its volume is so loud that each speaker is outputting its maximum power. What happens when that sound pans around to only one speaker? The answer is that we'd probably blow out the speaker, because our constant power pan law requires us to use more gain for one speaker than for two.

To prevent this problem, we need to artificially lower the maximum gains of our sounds across the board, such that the worst-case scenario of playing the sound in one speaker won't overdrive that speaker. The practice of artificially reducing the maximum range of volume is known as “leaving oneself some *headroom*.”

The concept of headroom also applies to *mixing*. When two or more sounds are mixed, their amplitudes add up. By leaving some headroom in our mix, we can accomodate worst-case scenarios where a large number of high-volume sounds play simultaneously.

14.4.4.4 To Center or Not to Center?

In cinema, the center channel was historically used for speech; only the sound effects would be panned to the other speakers around the room. The idea behind this practice was that the characters in the movie are usually on-screen when they speak, so the audience expects to hear their voices front-and-center. This approach has the nice side-effect of separating out the speech from the rest of the sounds in the film, meaning that loud sound effects won't use up all the available headroom and drown out the dialog.

In 3D games, the situation is quite different. The player generally wants to hear dialog coming from the “correct” location around him or her. If the player swings the camera by 180 degrees, the dialog should likewise swing about the

声场角度为 180 度。因此，游戏通常不会将所有对话都分配给中置扬声器；而是将其包含在声效和对话的声像中。

当然，这又回到了动态余量的问题——现在响亮的枪声会完全淹没对话。顽皮狗通过“折衷”解决了这个问题，我们总是在中置声道播放部分对话，同时将部分对话与音效一起平移到其他扬声器上。

14.4.4.5 重点

当声源距离听者较远时，我们可以将其视为点声源。我们只需计算一个方位角，并将其输入到我们的恒功率声像平移系统中即可。然而，当声源接近或实际进入定义扬声器与听者径向距离的圆周时，它就无法再被准确地建模为一个由单一角度表示的点声源。

设想一下朝向并经过声源的情况。起初，声源完全出现在前置扬声器中。当它经过听众时，我们需要以某种方式将声音传输到后置扬声器。如果我们将声音建模为点声源，我们唯一的选择就是将声音从前置扬声器“弹出”到后置扬声器。

理想情况下，我们希望声像在靠近听众时，逐渐在扬声器周围“扩散”。这样，当声源靠近听众时，我们可以开始在侧置扬声器中播放更多声音。当声源与听众重合时，声音可以在所有七个（或五个）扬声器中播放。当声源通过后，我们可以平滑地将声音过渡到后置扬声器，并在其远离听众后方时将前置扬声器的增益降至零。

如果我们把声源建模为一条圆弧而不是扬声器圆上的一个点，就能实现类似甚至更多的效果。换个角度来看，我们可以将每个声源想象成三维空间中的任意形状，它在扬声器圆上的投影与一定的角度成正比，在圆内形成一个“楔形饼”。这类似于三维图形中环境光遮蔽计算中常用的立体角概念——详情请参阅http://en.wikipedia.org/wiki/Solid_angle。

我们将扩展声源所夹的角度称为聚焦角，并将其表示为α。点声源可以被认为是$\alpha = 0$的“边缘情况”。聚焦角如图14.31所示。

要渲染具有非零聚焦角的声音，我们必须首先确定与其在扬声器圆上的投影圆弧相交或与圆弧紧邻的扬声器子集。然后，我们必须将声音的

sound field by 180 degrees. As such, games usually do not assign all dialog to the center speaker; instead, it is included in the pan for both sound effects and dialog.

Of course, this brings us back to the headroom problem—loud gunfire can now completely drown out the speech. At Naughty Dog, we overcame this problem by “splitting the difference” and always playing *some* of the dialog in the center channel, as well as panning some of it to the rest of the speakers along with the sound effects.

14.4.4.5 Focus

When the source of a sound is far away from the listener, we can treat it as a point source. We simply calculate a single azimuthal angle and feed it into our constant power panning system. However, when a sound source approaches or actually enters into the circle that defines the radial distance of the speakers from the listener, it can no longer be accurately modeled as a point source represented by a single angle.

Consider the case of moving toward and past a sound source. At first, the sound source appears entirely in the front speakers. As it passes the listener, we somehow need to transfer the sound to the rear speakers. If we model the sound as a point source, our only option is to “pop” the sound from the fronts to the rears.

Ideally we'd like the sound's image to gradually “spread out” around the speaker circle as it approaches. That way, as it nears the listener, we can start playing more of it in the side speakers. When the sound source is coincident with the listener, it can be played in all seven (or five) speakers. And once it passes, we can smoothly transition the sound to the rears, dropping the front gains to zero as it recedes behind the listener.

We can do this kind of thing and more if we model a sound source not as a point on the speaker circle but as an *arc*. Looking at it another way, we can think of each sound source as having an arbitrary shape in 3D space, and its *projection* onto the speaker circle *subtends* a certain angle, defining a “pie wedge” shape within the circle. This is analogous to the concept of *solid angle* often used in the calculation of ambient occlusion in 3D graphics—see http://en.wikipedia.org/wiki/Solid_angle for details.

We'll call the angle subtended by an extended sound source the *focus angle*, and we'll denote it α. A point source can be thought of as an “edge case” in which $\alpha = 0$. The focus angle is depicted in Figure 14.31.

To render a sound with a nonzero focus angle, we must first determine the subset of speakers that either intersect its projected arc on the speaker circle, or are immediately adjacent to the arc. Then we must divide the sound's

图 14.31 聚焦角 α 定义了扩展声源在扬声器圆上的投影。

这些扬声器之间的强度/功率，以引起对延伸到投影弧的幻影图像的感知。

我们可以通过多种方式将声音分配给相关的扬声器。例如，我们可以安排所有位于焦点“饼状”区域内的扬声器接收相同的最大功率，然后将较少的声音分配给圆弧附近紧邻的两个扬声器，以产生衰减。但无论如何操作，我们必须始终遵循恒定功率声像定律。因此，我们必须这样设置增益，使它们的平方和（即功率之和）等于原始未声像声源的平方增益。

14.4.4.6 处理垂直度

无论是立体声还是环绕声，扬声器都大致位于水平面上。这种布局使得声音很难定位在听众耳朵平面的上方或下方。

当然，理想的方案是使用球形扬声器阵列来模拟真正的“环绕声”声场。一种名为 Ambisonics (http://en.wikipedia.org/wiki/Ambisonics) 的技术能够同时支持平面和球形扬声器阵列。然而，目前还没有任何游戏机支持该技术。索尼现在在其 PS4 Platinum 无线耳机中提供了 3D 音频技术，并且一些游戏也开始支持该技术。但即使有了 3D 音频技术，游戏仍然需要支持 5.1 和 7.1 声道系统的平面扬声器阵列。

事实证明，焦点的概念可以用来模拟一些

Figure 14.31. The focus angle α defines the projection of an extended sound source on the speaker circle.

intensity/power among these speakers in order to induce the perception of a phantom image that extends across the projected arc.

We can divide the sound amongst the relevant speakers in various ways. For example, we could arrange for all the speakers that lie *within* the focus “pie slice” to receive equal maximum power, and then apportion less of the sound to the two speakers immediately adjacent the arc to create a fall-off. But no matter how we do it, we must remember to always obey the constant power pan law. So, we must set the gains in such a way that the sum of their squares (i.e., the sum of their powers) equals the squared gain of the original unpanned sound source.

14.4.4.6 Dealing with Verticality

In both stereo and surround sound set-ups, the speakers all lie roughly in a horizontal plane. This arrangement makes it tricky to position sounds above or below the plane of the listener’s ears.

The ideal of course would be to model a true “periphonic” sound field by using a spherical speaker arrangement. A technology known as *Ambisonics* (http://en.wikipedia.org/wiki/Ambisonics) is capable of accommodating both planar and spherical speaker arrangements. However, it is not supported by any game console—at least not yet. Sony now offers a 3D audio technology in their Platinum Wireless Headset for PS4, and games are beginning to support it. But even in the presence of 3D audio technology, games still need to support a planar speaker arrangement for 5.1 and 7.1 sound systems.

It turns out that the concept of *focus* can be leveraged to simulate some de-

我们的声音意象中垂直度的程度。我们只需将所有声音投射到水平面上，然后对投射距离过近或过远的声音使用非零聚焦角。远处的抬高声音的渲染方式与非抬高声音几乎相同。但当抬高声音从头顶上方经过时，我们会将其在多个扬声器之间混合，从而在扬声器圈内产生幻影图像。如果我们将其与基于距离的衰减和与频率相关的大气吸收相结合，我们就可以为听众提供足够的提示，使声音听起来像是位于听众上方或下方。

14.4.4.7 关于 Pan 的进一步阅读

恒定功率平移定律的基础知识可以在这里找到：http://www.rs-met.com/documents/tutorials/PanRules.pdf。以下网站也是关于该主题的丰富资源：http://www.music.miami.edu/programs/mue/Research/jwest/Chap_3/Chap_3_IID_Based_Panning_Methods.html。

赫尔辛基理工大学的 Ville Pukki 发表了题为“通过振幅平移技术生成和感知空间声音”的论文，可在 https://aaltodoc.aalto.fi/bitstream/handle/123456789/2345/isbn9512255324.pdf?sequence=1 上查阅，该论文清晰地描述了空间化问题，概述了基于矢量的振幅平移 (VBAP) 方法，同时提供了广泛的参考书目以供进一步阅读。

David Griesinger 的论文《立体声和环绕声声像的实践》也值得一读；可在 http://www.davidgriesinger.com/pan_laws.pdf 获取。David 的网站上充满了关于声音感知和音频再现技术的研究成果。

14.4.5 传播、混响和声学

即使我们实现了基于距离的衰减、声像和多普勒，我们的3D声音引擎仍然无法生成逼真的音景。这是因为人类用来辨别自身所处空间类型的许多听觉线索，都来自于早期反射、晚期混响以及由声波通过多条路径到达耳朵而产生的头部相关传递函数 (HRTF) 效应。“声音传播建模”这一术语可以应用于任何旨在考虑声波在空间中传播方式的技术。

在研究、互动媒体和游戏中，人们使用了许多不同的方法。这些技术可以分为三大类：

- 几何分析试图模拟声波的实际传播路径，

gree of verticality in our sound imagery. We simply *project* all sounds onto the horizontal plane, and then use a nonzero focus angle for any sounds whose projections fall too close to or within the speaker circle. An elevated sound that is far away will be rendered in virtually the same way as one that is not elevated. But as the elevated sound passes overhead, we blend it across multiple speakers, thereby producing a phantom image within the speaker circle. If we combine this with distance-based attenuation and frequency-dependent atmospheric absorptions, we can provide the listener with enough cues to make the sound seem to be located above or below the listener.

14.4.4.7 Further Reading on Pan

The basics of the constant power pan law can be found here: http://www.rs-met.com/documents/tutorials/PanRules.pdf. The following site is also a great resource on the topic: http://www.music.miami.edu/programs/mue/Research/jwest/Chap_3/Chap_3_IID_Based_Panning_Methods.html.

The paper entitled “Spatial Sound Generation and Perception by Amplitude Panning Techniques” by Ville Pukki of the Helsinki University of Technology, available at https://aaltodoc.aalto.fi/bitstream/handle/123456789/2345/isbn9512255324.pdf?sequence=1, provides a clear description of the spatialization problem and outlines the vector based amplitude panning (VBAP) method, as well as providing an extensive bibliography for further reading.

David Griesinger’s paper, “Stereo and Surround Panning in Practice,” also makes for a very interesting read; it is available at http://www.davidgriesinger.com/pan_laws.pdf. David’s website is chock full of research on sound perception and audio reproduction technologies.

14.4.5 Propagation, Reverb and Acoustics

Even if we were to implement distance-based attenuation, pan and Doppler, our 3D sound engine still wouldn’t be able to generate a realistic soundscape. This is because a lot of the auditory cues we humans use to sort out what kind of space we’re in come from the early reflections, late reverberations and head-related transfer function (HRTF) effects caused by sound waves taking multiple paths to reach our ears. The term “*sound propagation modeling*” can be applied to any technique that is designed to take into account the ways in which sound waves propagate through a space.

Many different approaches are used, both in research and in interactive media and games. These technologies fall into three basic categories:

- *geometric analysis* attempts to model the actual pathways taken by sound waves,

- 基于感知的模型专注于使用聆听空间声学的 LTI 系统模型来重现耳朵的感知，并且
- 临时方法采用各种近似值，以最少的数据和/或处理带宽产生相当准确的声学效果。

以下论文很好地概述了前两类的许多技术：http://www-sop.inria.fr/reves/Nicolas. Tsingos/publis/presence03.pdf。在本节中，我们将简要讨论 LTI 系统建模，然后将注意力转向一些临时方法，因为它们在实际游戏中更实用。

14.4.5.1 使用 LTI 系统建模传播效应

想象一下，我站在一个房间里，房间里摆满了各种材质的物品。房间里发出一种声音。声音在房间里反射、衍射、回荡，最终到达我的耳朵。仔细想想，这些声波的具体路径其实并不重要。唯一影响我感知的是干直达声波与各种时移的、可能被消音或其他改变的湿间接声波的特定叠加。

事实证明，所有这些效应都可以用线性时不变 (LTI) 系统建模。理论上，如果我们能够测量房间中一对代表声源和听者的点的脉冲响应，我们就能精确地确定在该声源位置播放的任何声音在听者位置听到时应该是什么效果。我们只需要将干声与脉冲响应进行卷积即可！

$$p_{\text{wet}}(t) = p_{\text{dry}}(t) * h(t).$$

乍一看，这项技术似乎是灵丹妙药。然而，它实际上比乍一看更困难，也更不实用。在现实生活中，确定一个空间的脉冲响应相当容易——你可以录制一个短暂的“咔哒”声，它近似于单位脉冲δ(t)，录制的信号将近似于h(t)。但在虚拟空间中，我们需要对每个游戏空间进行复杂而昂贵的模拟才能确定h(t)。此外，为了准确地模拟房间的声学效果，我们需要对整个游戏世界中的大量源-听众点对执行此计算，一旦计算出来，这些数据的大小将是巨大的。最后，卷积运算本身并不便宜，而且过去的游戏机和声卡缺乏对游戏中的每个声音实时执行此操作的强大功能。

- *perceptually based models* focus on reproducing what the ear perceives using an LTI system model of the acoustics of a listening space, and
- *ad hoc methods* employ various kinds of approximations to produce reasonably accurate acoustics with minimal data and/or processing bandwidth.

The following paper does a good job of surveying many of the techniques that fall into the first two categories: http://www-sop.inria.fr/reves/Nicolas. Tsingos/publis/presence03.pdf. In this section, we'll briefly discuss LTI systems modeling, and then turn our attention to a few ad hoc methods, because they tend to be more practical for use in real games.

14.4.5.1 Modeling Propagation Effects with an LTI System

Imagine that I am standing in a room containing various objects made of various materials. A sound is made in the room. It reflects and diffracts and bounces around the room, and eventually reaches my ears. If you think about it, it doesn't really matter which specific paths those sound waves took. The only thing that affects my perception is the specific superposition of the dry direct sound waves and the various time-shifted and possibly muffled or otherwise altered wet indirect waves.

It turns out that all of these effects can be modeled with a linear time-invariant (LTI) system. Theoretically, if we could measure the *impulse response* of the room for a given pair of points that represent the source of the sound and the listener, we can determine exactly how *any* sound we might play at that source location should sound if heard at the listener position. All we need to do is convolve the dry sound with the impulse response!

$$p_{\text{wet}}(t) = p_{\text{dry}}(t) * h(t).$$

This technique seems like a silver bullet at first blush. However, it is actually more difficult and less practical than it may at first seem. It's pretty easy to determine the impulse response of a space in real life—you can record the sound of a short "click" that approximates the unit impulse $\delta(t)$, and the recorded signal will approximate $h(t)$. But in a virtual space, we'd need to perform a complex and expensive simulation of each play space in order to determine $h(t)$. Also, to model the room's acoustics accurately, we'd need to perform this calculation for a large number of source-listener point pairs throughout the game world, and once calculated the size of this data would be immense. Finally, the operation of convolution is itself not inexpensive, and game consoles and sound cards have in the past lacked the horsepower to do this for every sound in the game in real time.

图 14.32.根据听众的位置交叉混合混响设置是一个好主意。

现代游戏硬件的性能日益强大，基于卷积的传播建模方法也变得越来越可行。例如，Micah Taylor 等人创建了一个卷积混响的实时演示，并取得了令人满意的效果——参见 https://intel.ly/2J8Gpsu。即便如此，大多数游戏仍然没有使用这种方法，而是依赖于各种临时方法和近似值来模拟环境混响。

14.4.5.2 混响区域

模拟游戏空间湿润特性的一种常用方法是手动放置区域来标注游戏世界，每个区域都标记有适当的混响设置，例如预延迟、衰减、密度和扩散。有关这些参数的讨论，请参阅第 14.1.3.4 节。当虚拟听众穿过这些区域时，我们可以点亮相应的混响模式：如果玩家进入一个铺有瓷砖的大房间，我们可以增强回声；当玩家进入一个小衣柜时，我们可以虚拟地消除混响，产生非常干的声音。

当听众穿过游戏空间时，在混响设置之间平滑地交叉混合是一个好主意。我们可以使用简单的线性插值来对每个参数执行这种交叉混合。混合百分比最好通过测量听众进入区域的程度来计算。例如，想象一下通过门口在室外空间和室内空间之间移动。我们可以在门口周围定义一个发生混合的区域。如果听众完全位于混合区域之外，则混合百分比应产生 100% 的室外混响设置和 0% 的室内设置。如果听众站在混合区域的中间点，我们希望混响设置混合比例为 50/50。一旦听众走出建筑物内的混合区域，我们将达到 0% 室外/ 100% 室内混合。图 14.32 说明了这个想法。

Figure 14.32. It's a good idea to cross-blend between reverb settings based on the position of the listener.

Modern gaming hardware is getting more powerful all the time, and a convolution-based approach to propagation modeling is becoming more feasible. For example, Micah Taylor et al. created a real-time demo of convolution reverb that produced promising results—see https://intel.ly/2J8Gpsu. That said, most games still don't use this approach, but instead they rely on various ad hoc methods and approximations to model environmental reverb.

14.4.5.2 Reverb Regions

One common approach to modeling the wet characteristics of a play space is to annotate the game world with manually placed regions, each of which is tagged with appropriate reverb settings such as pre-delay, decay, density and diffusion. See Section 14.1.3.4 for a discussion of these parameters. As the virtual listener moves through these regions, we can light up the appropriate reverb mode: If the player enters a large tiled room, we can bump up the echos; when the player enters a small closet, we can virtually eliminate the reverb to produce a very dry sound.

It's a good idea to smoothly cross-blend between reverb settings as the listener moves through the play space. We can use simple linear interpolation to perform this cross-blend for each parameter. The blend percentage is best calculated using a measure of how far "into" the region the listener is. For example, imagine moving between an outdoor space and an indoor space through a doorway. We could define a region around the doorway within which the blend occurs. If the listener is entirely outside the blend region, the blend percentage should yield 100% of the outdoor reverb settings and 0% of the indoor settings. If the listener is standing at the halfway point within the blend region, we'd want a 50/50 mix of the reverb settings. Once the listener passes out of the blend region inside the building, we'll have reached a 0% outdoor / 100% indoor blend. This idea is illustrated in Figure 14.32.

14.4.5.3 阻碍、闭塞和排除

当我们使用区域来定义游戏空间的声学效果时，我们通常会为每个区域分配一个脉冲响应函数或一组混响设置。这可以捕捉每个游戏空间的精髓（例如，铺着瓷砖的宽敞大厅、堆满外套的小衣柜、平坦的户外原野等等）。但由于障碍物的存在，这会导致声学效果的再现不够完美。例如，想象一个方形房间，中间有一根大柱子。如果声源位于房间的角落，听众在房间内移动时会感受到截然不同的音色，这取决于声源的直达路径是否被柱子遮挡。如果我们对这个房间使用一组混响参数，就无法捕捉到这些细微的差别。

为了解决这个问题，我们可以尝试以某种方式模拟环境的几何和材料特性，确定声波如何受到其路径中障碍物的影响，然后使用此分析的结果来改变与房间相关的“基本”混响设置。

图 14.33 展示了游戏世界中的物体和表面影响声波传播的三种方式：

- 遮挡。这指的是从声源到听者之间不存在畅通路径的情况。即使听者与声源之间只有一堵薄墙或一扇门，即使完全被遮挡，听者仍然可能听到。被遮挡的声音中的干湿成分要么同时衰减和/或被抑制，要么从听者的角度来看，声音完全静音。

- 阻塞。这指的是声音与听者之间的直接路径被阻挡，但存在间接路径的情况。例如，当声源经过汽车、柱子或其他障碍物后方时，就会发生阻塞。阻塞声音的干声成分要么完全消失，要么被严重抑制，而湿声成分也可能发生变化，因为声波需要经过更长、反射次数更多的路径才能到达听者。

- 隔离。这种情况指的是声源和听者之间存在一条畅通的直接路径，但间接路径在某种程度上受到了影响。如果声音在一个房间内产生，并穿过门窗等狭窄的开口到达听者，就会发生这种情况。在隔离情况下，声音的干声成分保持不变，但湿声成分会被衰减、抑制，或者在非常狭窄的开口处完全消失。

14.4.5.3 Obstruction, Occlusion and Exclusion

When using regions to define the acoustics of our play spaces, we typically assign a *single* impulse response function or a *single* collection of reverb settings to each region. This captures the essence of each play space (e.g., large tiled hall, small closet lined with coats, flat outdoor plain, etc.). But it results in a less-than-perfect reproduction of the acoustics that arise due to obstacles. For example, imagine a square room with a large pillar in the center. If a sound source is located in the corner of the room, a listener will perceive a very different timbre as he or she moves about the room, depending on whether the direct path is obstructed by the pillar or not. If we use a single set of reverb parameters for this room, we cannot capture these subtleties.

To address this problem, we can attempt to model the geometry and material properties of the environment in some way, determine how sound waves are affected by the obstacles in their path, and then use the results of this analysis to alter the “base” reverb settings associated with the room.

Figure 14.33 shows the three ways in which the objects and surfaces in the game world can affect the transmission of sound waves:

- *Occlusion*. This describes a situation in which there exists no unfettered path from the sound source to the listener. A listener might still be able to hear a fully occluded sound, if for example there is only a thin wall or door between it and the source of the sound. Either the dry and wet components of an occluded sound are both attenuated and/or muffled, or the sound is entirely silent from the point of view of the listener.
- *Obstruction*. This describes a case in which the direct path between the sound and the listener is blocked, but an indirect path is available. Obstruction can occur for example when a sound source passes behind a car, pillar or other obstacle. The dry component of an obstructed sound is either entirely absent or greatly muffled, and the wet component may be altered as well to account for the sound waves having to take a longer, more reflected path to the listener.
- *Exclusion*. This describes a case in which there is a free direct path between source and listener, but the indirect path is compromised in some way. This can happen if a sound is produced in one room and passes through a narrow opening such as a door or window to reach the listener. In an exclusion situation, the dry component of the sound remains unaltered but the wet component is attenuated, muffled or, for very narrow openings, entirely absent.

图 14.33. 从上到下：闭塞、阻碍和排除。

分析直接路径

判断直达路径是否被阻挡并不困难。我们只需从听者向每个声源投射一条射线（参见第 13.3.7.1 节）。如果被阻挡，则直达路径被遮挡。如果没有被阻挡，则直达路径畅通无阻。

如果我们想要模拟声音穿过墙壁或其他障碍物的传播，仍然可以使用射线投射。我们从声源向听者投射一条射线，每次接触时，我们都会查询受影响表面的材料属性，以确定它吸收了多少声音的能量。如果它允许一些能量通过，我们可以从障碍物的另一侧开始投射另一条射线，继续追踪到达听者的路径。一旦所有声音的能量都被吸收，我们就可以得出结论，声音无法被听到。但是，如果射线一路到达听者，没有损失任何声能，我们可以将干声成分的增益衰减相应的量，以模拟声音的传播。

Figure 14.33. From top to bottom: occlusion, obstruction and exclusion.

Analyzing the Direct Path

Determining whether the direct path is blocked or not is not difficult. We simply cast a ray (see Section 13.3.7.1) from the listener to each sound source. If it is blocked, the direct path is occluded. If not, it is free.

If we wish to model sound transmission *through* walls and other obstacles, ray casting can still be used. We cast a ray from source to listener, and for each contact we query the material properties of the impacted surface to determine how much of the sound's energy it absorbs. If it allows some energy to pass through, we can cast another ray starting on the other side of the obstacle and continue tracing the path to the listener. Once all of the sound's energy has been absorbed, we can conclude that the sound cannot be heard. But if the ray makes it all the way to the listener without losing all sound energy, we can attenuate the gain of the dry sound component by the corresponding amount to simulate transmission of the sound.

分析间接路径

确定间接路径是否被遮挡是一个更加困难的问题。理想情况下，我们会执行某种搜索（例如 A* 算法）来确定从声源到听者是否存在路径，以及每条可行路径会引入多少衰减和反射。实际上，这种路径追踪方法很少使用，因为它占用大量处理器和内存资源。而且，归根结底，我们游戏程序员对创建物理上精确的模拟效果（比如能让我们获得诺贝尔物理学奖）并不真正感兴趣。我们只想创造一个身临其境、令人信服的音景。

别担心，一切还好。我们可以通过各种方式获得声音间接路径的近似模型。例如，如果我们使用混响区域来模拟游戏中各个空间的整体声学效果（参见第 14.4.5.2 节），我们就可以利用这些区域来确定是否存在间接路径。例如，我们可以运用一些简单的经验法则：

1. 如果源和侦听器位于同一区域，则假设存在间接路径。

2. 如果声源和听众位于不同的区域，则假设间接路径被遮挡。

利用这些假设并结合我们的直接路径射线投射的结果，我们可以区分四种情况：自由、遮挡、阻碍或排除。

考虑衍射

当任何波穿过狭窄的开口或与角落相互作用时，它都会像图 14.34 所示那样扩散。我们将这种现象称为衍射。由于衍射，只要直达路径和弯曲路径之间的角度差不太大，声音在拐角处也能被听到，就像存在直达路径一样。

判断声音是否能够通过衍射到达听者的一种方法是，在中心“直达”射线周围投射几条“弯曲”射线。大多数碰撞引擎不支持曲线路径追踪，但我们可以通过使用多条直线射线投射来模拟曲线路径。图 14.35 展示了一个简单的示例，其中从声源到听者投射了五条射线——一条直达射线，加上两条“弯曲”轨迹，每条轨迹都由两条直线射线投射组成。从技术上讲，我们对想要追踪的每条曲线路径都采用了分段线性近似。

Analyzing the Indirect Path

Determining whether the indirect path is occluded is a much more difficult problem. Ideally, we'd perform some kind of search (A* perhaps) to determine whether or not a path exists from the source to the listener, and also how much attenuation and reflection is introduced by each viable path. In practice, this *path tracing* method is rarely used because it is processor- and memory-intensive. And at the end of the day, we game programmers aren't really interested in creating physically accurate simulations that will win us Nobel prizes in physics. We merely want to produce a soundscape that is *immersive* and *believable*.

Never fear, all is not lost. There are all sorts of ways in which we can obtain an *approximate* model of the indirect path of a sound. For example, if we are using *reverb regions* to model the overall acoustics of the various spaces in our game (see Section 14.4.5.2), we could leverage these regions to determine whether an indirect path exists. For example, we could use some simple rules of thumb:

1. If the source and listener are in the same region, assume an indirect path exists.

2. If the source and listener are in different regions, assume the indirect path is occluded.

Using these assumptions combined with the results of our direct path ray cast, we can differentiate between the four cases: free, occluded, obstructed or excluded.

Accounting for Diffraction

When any wave passes through a narrow opening or interacts with a corner, it spreads out as shown in Figure 14.34. We call this phenomenon *diffraction*. Because of diffraction, sounds can be heard around corners as if a direct path existed, as long as the angular difference between the direct path and curved path is not too great.

One way to determine whether sound can diffract in order to reach the listener is to cast a few "curved" rays around the central "direct" ray. Most collision engines don't support curved path tracing, but we can emulate a curved path by using multiple straight-line ray casts. Figure 14.35 shows a simple example, in which five rays are cast from the sound source to the listener—one direct ray, plus two "curved" traces comprised of two straight-line ray casts each. Technically speaking we're employing a *piecewise-linear approximation* to each curved path we wish to trace.

如果直射光线被遮挡，但弯曲的轨迹可以“看到”听众，这告诉我们听众位于附近角落的“衍射区域”内，并且应该听到声音，就好像它没有被遮挡一样。

应用混响和增益模型

到目前为止，我们已经讨论了如何确定直接路径和间接路径是否被阻塞。这种分析还可以告诉我们一些关于遮挡或障碍物对声学影响的信息。（例如，穿过墙壁的声音可能会变得低沉；声音沿着较长的“弹性”路径传播可能会产生大量的混响。）现在的问题是：我们如何在渲染声音时运用这些知识？

一种简单的方法是，根据直接路径或间接路径是否完全或部分阻塞，分别衰减声音的干湿成分。为了优化效果，我们还可以根据在确定声音路径时收集到的启发式信息，对声音的每个成分施加或多或少的混响。每个游戏的需求各不相同，所以在这种情况下，反复试验是你最好也是唯一的选择！

混合阻塞的声音

如果你真的去实现我们上面讨论的所有内容，就会发现一个明显的问题。当声源在上述四种状态之间移动时——例如，从自由状态到受阻状态——声音的音色和响度会显得“突突突”。有很多方法可以平滑这种过渡。你可以应用一点滞后，即延迟音响系统对每个声音受阻状态变化的响应，然后利用这个短暂的延迟窗口，在两组混响设置之间平滑地交叉混合。但是延迟

图 14.34。衍射使声音的干声部分清晰可闻，即使直接路径被阻塞。

If the direct ray is occluded but the curved traces can “see” the listener, this tells us that the listener is within the “diffraction region” around a nearby corner, and should hear the sound as if it is not occluded.

Applying the Model Using Reverb and Gain

Thus far, we've discussed how to determine whether the direct and indirect paths are blocked or not. This analysis can also tell us something about the acoustic impact of an occlusion or obstruction. (For example, sound passing through a wall can be muffled; sound taking a long “bouncy” path might introduce a lot of reverb.) The question now is: How do we apply this knowledge when rendering the sound?

One simple approach is to simply attenuate the dry and wet components of the sound individually, based on whether the direct or indirect paths are totally or partially blocked, respectively. To finesse the results, we can also apply more or less reverb to each component of the sound, based on whatever heuristic information we gathered when determining the path(s) taken by the sound. The needs of every game are different, so this is one of those times when trial and error is your best and only option!

Blending Obstructed Sounds

If you were to go off and implement everything we talked about in the sections above, you'd notice a glaring problem. As a sound source moves between the four states described above—for example, from being free to being obstructed—the timbre and loudness of the sound will seem to “pop.” There are a number of ways to smooth out such transitions. You could apply a little *hysteresis*, meaning that you delay the response of the sound system to changes in the obstructed state of each sound, and then use this short delay window to smoothly cross-blend between the two sets of reverb settings. But the delay

Figure 14.34. Diffraction causes the dry component of a sound to be clearly audible even when the direct path is blocked.

图 14.35 曲线射线投射可以用多条直线射线来近似。

可能会很明显，所以这不是一个理想的解决方案。

为了《神秘海域》和《最后生还者》系列，顽皮狗的高级音效程序员乔纳森·拉尼尔发明了一套专有系统，他称之为“随机传播建模”。我可以告诉你，在不泄露任何商业机密的情况下，这个系统会向每个声源投射一束射线，有些是直接的，有些是间接的，并在多帧中累积这些命中/未命中的结果。根据这些数据，我们生成了一个概率模型，用于计算每个声源的干声和湿声所经历的遮挡程度。这使我们能够平滑地将声音从完全遮挡过渡到完全自由，而不会出现明显的“砰砰”声。

14.4.5.4 《最后生还者》中的声音传送门

对于《最后生还者》，顽皮狗需要一种方法来模拟声音在环境中的实际传播路径。如果敌方 NPC 站在一条通往玩家所在房间的长走廊里说话，我们希望玩家能够听到他的声音从门口传来，而不是沿着直线路径“穿过墙壁”。

为此，我们使用了一个互连区域网络。区域分为两种：房间和入口。对于每个声源，我们利用声音设计师在布置区域时提供的连接信息，找到从听众到声音的路径。如果声源和听众都在同一个房间，我们会使用在《神秘海域》系列游戏中使用过的、行之有效的阻塞/遮挡/排除分析方法。但如果声源位于通过入口直接连接到听众房间的房间中，我们会像声音位于入口区域一样播放声音。我们发现，只需要在房间连接图中“跳一跳”就可以使其适用于游戏中出现的所有真实情况。显然，我在这里遗漏了很多重要的细节，但图 14.36 说明了这个系统的基本工作原理。

Figure 14.35. Curved ray casts can be approximated using multiple straight-line rays.

might be noticeable, so this isn't an ideal solution.

For the *Uncharted* and *The Last of Us* series, Naughty Dog's senior sound programmer Jonathan Lanier invented a proprietary system that he called *stocastic propagation modeling*. Without giving away any trade secrets, I can tell you that this system involves casting a bunch of rays to each sound source, some direct and some indirect, and accumulating these hit/miss results over many frames. From this data, we generate a probabalistic model of the degree of occlusion experienced by both the dry and wet components of each sound source. This allows us to smoothly transition a sound from being fully obstructed to fully free without noticeable "pops."

14.4.5.4 Sound Portals in *The Last of Us*

For *The Last of Us*, Naughty Dog needed a way to model the actual pathways that sounds take through the environment. If an enemy NPC is speaking while standing in a long hallway that connects to the room the player is in, we wanted to be able to hear the sound of his voice coming from the *doorway*, not "through the wall" along a straight-line path.

To do this, we used a network of interconnected regions. There were two kinds of regions: *rooms* and *portals*. For each sound source, we found a path from the listener to the sound by using connectivity information provided by the sound designer when laying out the regions. If both the sound source and listener were in the same room, we'd use the tried and true method of performing obstruction/occlusion/exclusion analysis that we used on the *Uncharted* series. But if the sound source was in a room directly connected to the listener's room via a portal, we would play the sound *as if* it were located in the portal region. We found that we only needed to go "one hop" in the room connectivity graph to make this work for all real situations that arose in the game. Obviously I'm leaving out a lot of important details here, but Figure 14.36 illustrates the basics of how this system worked.

图 14.36.顽皮狗公司在《最后生还者》中使用的基于门户的音频传播模型。

14.4.5.5 环境声学的进一步阅读

音频传播建模和声学分析是当前研究的热点领域，随着硬件性能的不断提升，越来越多的先进技术被应用于游戏行业。以下列出了一些链接，希望能激发您的兴趣，但如果您使用 Google 搜索“声音传播”或“声学建模”，就能找到更多精彩内容！

- 育碧蒙特利尔工作室的 Jean-François Guay 撰写的《视频游戏中的实时声音传播》（https://bit.ly/2HdBiLc）；
- A 在 2003 年 GDC 上发表的“游戏中的现代音频技术”
  缅什科夫（https://bit.ly/2J7FYyD）；
- Jake Simpson 的“游戏中的 3D 声音”（https://bit.ly/2HfVFTU）。

14.4.6 多普勒频移

正如我们在 14.1.3.5 节中看到的，多普勒效应是一种频率变化，它取决于声源和听者之间的相对速度：v rel = v source − v listener 。这种频率变化可以通过简单地对声音信号进行时间缩放来近似。这就产生了艾尔文和花栗鼠让我们如此熟悉的“花栗鼠效应”——通过加快声音速度，音调也会上升。由于我们的声音信号是数字的（即采样的离散时间信号），这种时间缩放可以通过采样率转换来实现（参见 14.5.4.4 节）。然而，这并不严格地

Figure 14.36. The portal-based audio propagation model used in *The Last of Us* by Naughty Dog, Inc.

14.4.5.5 Further Reading on Environmental Acoustics

Audio propagation modeling and acoustics analysis are areas of active research, and more and more advanced techniques are being applied in the game industry as hardware capabilities continue to improve. A few links are listed below to whet your appetite, but a Google search for “sound propagation” or “acoustics modeling” will provide many more hours of enjoyment!

- “Real-Time Sound Propagation in Video Games” by Jean-François Guay of Ubisoft Montreal (https://bit.ly/2HdBiLc);
- “Modern Audio Technologies in Games” presented at GDC 2003 by A. Menshikov (https://bit.ly/2J7FYyD);
- “3D Sound in Games” by Jake Simpson (https://bit.ly/2HfVFTU).

14.4.6 Doppler Shift

As we saw in Section 14.1.3.5, the Doppler effect is a change in frequency that’s dependent upon the *relative velocity* between source and listener: $\mathbf{v}_{\text{rel}} = \mathbf{v}_{\text{source}} - \mathbf{v}_{\text{listener}}$. This frequency change can be approximated by simply time-scaling the sound signal. This results in the “chipmunk effect” with which Alvin and the Chipmunks have made us all so familiar—by speeding up a sound, the pitch also rises. Because our sound signals are digital (i.e., sampled discrete-time signals), this kind of time scaling can be accomplished via sample rate conversion (see Section 14.5.4.4). However, this is not strictly the

正确的做法，因为声音的加快或减慢会变得明显。

理想的解决方案是在不影响时间轴的情况下应用音高移位。这可以通过多种方式实现，包括相位声码器和时域谐波缩放方法。这些技术的完整描述超出了我们的讨论范围，但您可以在 http://www.dspdimension.com/admin/time-pitch-overview 上阅读更多相关信息。

时间无关的音高变换技术在音频引擎中非常强大，部分原因在于它还允许您执行频率无关的时间缩放。因此，您不仅可以在不改变多普勒时间的情况下改变声音的音高，还可以在不改变音高的情况下加快或减慢声音的速度，从而实现各种其他炫酷的效果。

14.5 音频引擎架构

到目前为止，我们已经讨论了 3D 声音渲染背后的概念和方法，以及其背后的理论和技术。在本节中，我们将关注用于实现 3D 音频渲染引擎的软件和硬件组件的架构。

与大多数计算机系统一样，游戏引擎的音频渲染软件通常被安排成分层硬件和软件组件的“堆栈”（见图 14.37）。

- 硬件不可避免地充当着这一结构的基础，至少提供必要的电路来驱动数字或模拟扬声器输出，从而将我们的电脑或游戏机连接到耳机、电视或环绕声家庭影院系统。音频硬件还可以通过提供编解码器、混音器、混响器、效果器、波形合成器和/或 DSP 芯片，为堆栈中位于其上方的软件提供“加速”。这些硬件通常被称为声卡

图 14.37 音频硬件/软件“堆栈”。

correct thing to do, because the speeding up or slowing down of the sound can become noticable.

The ideal solution is to apply a pitch shift without affecting the time axis. This can be done in a number of ways, including the *phase vocoder* and *time domain harmonic scaling* approaches. A complete description of these techniques is beyond our scope here, but you can read more about them at http://www.dspdimension.com/admin/time-pitch-overview.

Time-independent pitch shifting technology is an extremely powerful thing to have in your audio engine, in part because it also allows you to perform frequency-independent time scaling. So not only can you alter the pitch of sounds without changing timing for Doppler, you can also speed up or slow down sounds without altering their pitch for all sorts of other cool effects.

14.5 Audio Engine Architecture

To this point, we've discussed the concepts and methodologies behind 3D sound rendering, and the theory and technologies that underlie them. In this section, we'll turn our attention to the architecture of the software and hardware components used to implement a 3D audio rendering engine.

As with most computer systems, a game engine's audio rendering software is typically arranged into a "stack" of layered hardware and software components (see Figure 14.37).

- *Hardware* inevitably serves as the foundation of this structure, providing at minimum the necessary circuitry to drive the digital or analog speaker outputs that connect our PC or game console to a pair of headphones, a TV or a surround sound home theater system. Audio hardware may also provide "acceleration" to the software above it in the stack by supplying codecs, mixers, reverb tanks, effects units, waveform synthesizers and/or DSP chips in silicon. This hardware is often called the *sound card*

Figure 14.37. The audio hardware/software "stack."

因为 PC 有时通过插入式外围卡提供其音频功能。

- 在个人计算机上，硬件通常封装在驱动程序层中，从而允许操作系统支持来自广泛供应商的声卡。
- 在个人电脑和游戏机上，硬件和驱动程序通常都包含在低级应用程序编程接口 (API) 中，旨在使程序员无需处理直接控制硬件和驱动程序的繁琐细节。
- 3D 音频引擎本身就是建立在这些基础之上的。

音频硬件/软件堆栈向程序员提供的功能集通常模仿录音室和现场音乐会中使用的多通道混音台 (http://en.wikipedia.org/wiki/Mixing_console) 的功能集（参见图 14.38）。混音台可以接收来自麦克风和/或电子乐器的大量音频输入。输入的声音可以进行滤波和均衡处理，还可以添加混响和其他效果。然后，控制台用于将所有信号混合在一起，并根据声音设计师的需要设置声音的相对音量。最终的混音输出被路由到扬声器（用于现场表演）或多轨录音的各个声道。

同样，音频硬件/软件堆栈必须接受大量输入（2D 和 3D 声音），以各种方式处理它们，将它们混合在一起以便适当设置它们的相对增益，最后将这些信号平移到扬声器输出通道，为人类玩家产生三维音景的幻觉。

14.5.1 音频处理管道

正如我们在第 14.4.1 节中了解到的，渲染 3D 声音的过程涉及许多离散步骤：

- 对于每个 3D 声音，必须合成“干”数字 (PCM) 信号。
- 应用基于距离的衰减来提供与听众的距离感，并对信号施加混响，以模拟虚拟聆听空间的声学效果，并为听众提供空间感提示。这会产生一个新的“湿”信号。
- 将湿信号和干信号（独立地）平移到一个或多个扬声器，以便在三维空间中产生每个信号的最终“图像”。

because PCs sometimes provide their audio capabilities via a plug-in peripheral card.

- On a personal computer, the hardware is typically encapsulated in a *driver* layer, allowing the OS to support sound cards from a wide range of vendors.
- On both PCs and game consoles, the hardware and drivers are usually wrapped in a low-level *application programming interface* (API) designed to free the programmer from having to deal with the minutia of controlling the hardware and drivers directly.
- The *3D audio engine* itself is built on top of these foundations.

The feature set presented to the programmer by the audio hardware/software stack is usually modeled after the feature set of a *multi-channel mixer console* (http://en.wikipedia.org/wiki/Mixing_console) of the sort used in recording studios and at live concerts (see Figure 14.38). A mixer board can accept a relatively large number of audio inputs obtained from microphones and/or electronic instruments. The input sounds can be filtered and equalized, and reverb and other effects can be applied to them. The console is then used to mix all of the signals together, setting the relative volumes of the sounds as desired by the sound designer. The final mixed output is routed to the speakers (for a live performance) or to the individual channels of a multi-track recording.

In the same sense, the audio HW/SW stack must accept a large number of inputs (2D and 3D sounds), process them in various ways, mix them together so that their relative gains are set appropriately and finally pan these signals to the speaker output channels to produce the illusion of a three-dimensional soundscape for the human player.

14.5.1 The Audio Processing Pipeline

As we learned in Section 14.4.1, the process of rendering a 3D sound involves a number of discrete steps:

- For each 3D sound, a “dry” digital (PCM) signal must be *synthesized*.
- Distance-based attenuation is applied to provide a sense of distance from the listener, and *reverb* is applied to the signal to model the acoustics of the virtual listening space and to provide spatialization cues to the listener. This produces a new “wet” signal.
- The wet and dry signals are *panned* (independently) to one or more speakers in order to produce the final “image” of each signal in three-dimensional space.

图 14.38.Focusrite 的多通道混音控制台，支持 72 个输入和 48 个输出。

• 所有 3D 声音的平移多通道信号混合在一起形成单个多通道信号，该信号可以通过并行的 DAC 和放大器组发送以驱动模拟扬声器输出，或者直接发送到数字输出，例如 HDMI 或 S/PDIF。

显然，我们将渲染 3D 音频的过程视为一个管道。由于游戏世界通常包含大量声源，因此该管道的多个实例会同时运行。因此，

图 14.39.音频处理图（管道）。

Figure 14.38. A multi-channel mixer console by Focusrite with support for 72 inputs and 48 outputs.

- The panned multi-channel signals of all the 3D sounds are *mixed* together into a single multi-channel signal, which is either sent through a parallel bank of DACs and amps to drive the analog speaker outputs or sent directly to a digital output such as HDMI or S/PDIF.

Clearly, we think of the process of rendering 3D audio as a *pipeline*. And because a game world typically contains a large number of sound sources, multiple instances of this pipeline are in flight simultaneously. For this reason,

Figure 14.39. The audio processing graph (pipeline).

音频处理管道有时也称为音频处理图。它实际上是一个由相互连接的组件组成的图形，最终汇聚到构成最终混音和声像输出的几个扬声器通道。

图 14.39 展示了音频图的高级视图。

14.5.2 概念和术语

在深入探索音频处理流程之前，我们需要熟悉一些概念和用于描述它们的术语。

14.5.2.1 声音

每个通过音频渲染图的 2D 或 3D 声音都称为“声音”。这个术语源自电子音乐的早期：合成器通过一组称为“声音”的波形发生器产生音符。

合成器包含有限数量的波形发生器电路，因此电子音乐家会谈论他们的合成器能够同时产生多少个声音。同样，游戏的音频渲染引擎通常也包含有限数量的编解码器、混响单元等等。特定音频软硬件堆栈支持的最大声音数量取决于音频图中独立并行路径的数量。这个数量通常受内存资源、硬件资源和/或处理能力的限制。这个数量有时被称为系统支持的复音数。

2D 声音

游戏的音频渲染管线还必须能够处理 2D 声音，例如音乐、菜单音效、旁白配音等等。2D 语音也由音频渲染管线处理。2D 声音处理与 3D 声音处理的主要区别在于：

- 2D 声音源自多声道信号，每个可用扬声器对应一个声道信号，而 3D 声音源自干单声道信号。因此，2D 声音不会通过声像调节器。
- 2D 声音可能包含“烘焙”混响或其他效果。如果是这样，声音可能无法利用渲染引擎的混响功能。

因此，2D 声音通常在主混音器之前进入管道，在那里它们与 3D 声音相结合以产生最终的“混音”。

the audio processing pipeline is sometimes called the *audio processing graph*. It truly is a graph of interconnected components, ultimately culminating in the handful of speaker channels that comprise the final mixed, panned output. Figure 14.39 presents a high-level view of the audio graph.

14.5.2 Concepts and Terminology

Before we can explore the audio processing pipeline in any depth, we need to become familiar with a few concepts and the terminology used to describe them.

14.5.2.1 Voices

Each 2D or 3D sound passing through the audio rendering graph is called a *voice*. This term comes from the early days of electronic music: A synthesizer would produce musical notes via a set of waveform generators called “voices.”

A synthesizer contains a limited number of waveform generator circuits, so electronic musicians speak of how many simultaneous voices their synth can produce. In the same sense, a game’s audio rendering engine typically has a limited number of codecs, reverb units and so on. The maximum number of voices supported by a particular audio HW/SW stack is dictated by the number of independent parallel pathways through the audio graph. This number is generally bounded by limited memory resources, limited hardware resources and/or processing power limitations. This number is sometimes referred to as the degree of *polyphony* supported by the system.

2D Voices

A game’s audio rendering pipeline must also be capable of handling 2D sounds, such as music, menu sound effects, narrator voice-overs and so on. 2D voices are also processed by the audio rendering pipeline. The main things that differentiate 2D sound processing from 3D processing are:

- 2D sounds originate as multi-channel signals, one for each available speaker, whereas 3D sounds originate as dry monophonic signals. As such, 2D sounds do not pass through a pan pot.
- A 2D sound may contain “baked” reverb or other effects. If so, the sound may not make use of the reverb capabilities of the rendering engine.

As such, 2D sounds typically enter the pipeline just prior to the master mixer, where they are combined with the 3D sounds to produce the final “mix.”

图 14.40. 单个 3D 语音在音频图中经过的管道。

14.5.2.2 公交车

组成音频图的各个组件之间的互连称为总线。在电子学中，总线是一种主要用于连接其他电路的电路。在软件中，总线只不过是一种描述组件之间互连的逻辑结构。

14.5.3 语音总线

图 14.40 更详细地展示了单个 3D 语音在音频引擎渲染过程中所经过的组件流水线。在接下来的章节中，我们将详细探讨每个组件，并了解它们之间为何以这种方式相互连接。

14.5.3.1 声音合成：编解码器

音频信号以数字形式通过渲染图。术语“合成”描述了生成这些数字信号的过程。音频信号可以通过简单地“播放”预先录制的音频片段来合成。它们也可以通过程序生成，例如通过组合一个或多个基本波形（正弦波、方波、锯齿波等），和/或对谐波丰富的噪声信号应用各种滤波器。由于

Figure 14.40. The pipeline through which an individual 3D voice passes on its way through the audio graph.

14.5.2.2 Buses

The interconnections between the components that make up the audio graph are called *buses*. In electronics, a bus is a circuit whose primary purpose is to connect other circuits to one another. In software, a bus is nothing more than a logical construct that describes the presence of an interconnection between components.

14.5.3 The Voice Bus

Figure 14.40 presents a more-detailed view of the pipeline of components through which a single 3D voice passes as it is rendered by the audio engine. In the following sections, we'll explore each of these components in detail and learn why they are interconnected in the way that they are.

14.5.3.1 Sound Synthesis: Codecs

An audio signal passes through the rendering graph in digital form. The term *synthesis* describes the process of generating these digital signals. Audio signals may be synthesized by simply "playing back" a pre-recorded audio clip. They might also be procedurally generated, perhaps by combining one or more fundamental waveforms (sinusoid, square wave, sawtooth, etc.), and/or by applying various filters to a harmonically rich noise signal. Since

大多数游戏几乎都使用预先录制的音频片段，因此我们在此仅讨论它们。

预先录制的音频片段可以以目前使用的各种压缩和未压缩音频文件格式（参见第 14.3.2.3 节）中的任何一种提供给游戏引擎。原始 PCM 数据是音频处理图中各个组件接受的“规范”格式。因此，需要使用一种称为编解码器的设备或软件组件将每个源音频片段转换为原始 PCM 数据流。编解码器会解释源数据格式，并在必要时解压缩数据，然后将其传输到语音总线上，以便通过音频处理图进行处理。

14.5.3.2 增益控制

3D 世界中每个源声音的响度可以通过多种方式控制：录制音频片段时，我们可以设置录制音量，以产生所需响度的声音。我们可以使用离线工具处理音频片段来调整其增益。在运行时，我们还可以使用音频图中的增益控制组件动态调整音频片段的音量。有关增益控制的详细讨论，请参阅第 14.3.1.7 节。

14.5.3.3 发送

当录音室或现场音乐会的音响工程师想要为声音添加效果时，他/她可以将声音从多通道混音台输出，通过效果“踏板”，然后再送回混音台进行进一步处理。这些输出被称为辅助发送输出，简称辅助发送。

在音频处理图中，“辅助发送”一词的用法类似：它描述了管道中的一个分叉点，将信号分成两个并行信号。其中一个信号用于声音的干声部分。另一个信号通过混响/效果器组件传输，以产生声音的湿声部分。

14.5.3.4 混响

湿信号路径通常通过一个添加早期反射和晚期混响的组件进行路由。混响可以使用卷积来实现，如第 14.4.5.1 节所述。如果由于主机或 PC 缺少 DSP 硬件，或者游戏的 CPU 和/或内存预算不足，卷积无法实时实现，则可以使用混响箱来实现混响。这本质上是一个缓冲系统，它缓存声音的延时副本，然后将其与原始声音混合以模拟早期反射和/或晚期混响，并结合滤波器来模拟

most games use pre-recorded audio clips almost exclusively, we'll restrict our discussion to them here.

Pre-recorded audio clips can be provided to the game engine in any one of the myriad compressed and uncompressed audio file formats in use today (see Section 14.3.2.3). Raw PCM data is the "canonical" format accepted by the various components in the audio processing graph. Therefore, a device or software component known as a *codec* is used to convert each source audio clip into a raw PCM data stream. The codec interprets the source data format, decompresses the data if necessary, and then transmits it onto the voice bus for its journey through the audio processing graph.

14.5.3.2 Gain Control

The loudness of each source sound in the 3D world can be controlled in a number of ways: When recording the audio clip, we can set the recording levels to produce a sound at the desired loudness. We can process the clip in an offline tool to adjust its gain. At runtime, we can also dynamically adjust the volume of the clip using a *gain control* component within the audio graph. See Section 14.3.1.7 for a detailed discussion of gain control.

14.5.3.3 Aux Sends

When a sound engineer at a recording studio or live concert wants to apply effects to a sound, he or she can route the sound out of the multi-channel mixing console, through an effects "pedal," and then back into the mixing board for further processing. These outputs are known as *auxiliary send* outputs, or *aux sends* for short.

Within the audio processing graph, the term "aux send" is used in an analogous manner: It describes a bifurcation point in the pipeline, splitting the signal into two parallel signals. One of these signals is for the dry component of the sound. The other is piped through a reverb/effects component to create the wet component of the sound.

14.5.3.4 Reverb

The wet signal path is typically routed through a component that adds early reflections and late reverberations. Reverb might be implemented using a convolution, as described in Section 14.4.5.1. If convolution is not practical in real time, either because the console or PC lacks DSP hardware or because the game's CPU and/or memory budgets are insufficient, reverb can be implemented using a *reverb tank*. This is essentially a buffering system that caches time-delayed copies of a sound that are then mixed with the original to mimic early reflections and/or late reverberations, combined with a *filter* to mimic the

干扰效应和反射声波中高频成分的普遍衰减。

14.5.3.5 发送前过滤器

语音管道通常包含一个在辅助发送分支之前应用的滤波器，因此它同时作用于声音的干湿成分。这被称为预发送滤波器。它通常用于模拟声源处发生的现象。例如，我们可以用预发送滤波器来模拟佩戴防毒面具的人的声音。

14.5.3.6 发送后过滤器

辅助发送分叉后通常会提供另一个滤波器。因此，该滤波器仅适用于声音的干声部分。该滤波器可用于模拟直达声路径上障碍物/遮挡的消音效果。在顽皮狗，我们还使用发送后滤波器来实现由于大气吸收而产生的特定频率的衰减（参见第 14.1.3.2 节）。

14.5.3.7 平底锅

3D 声音的干湿分量在沿着语音总线传输的过程中都是单声道信号。在管道的末端，这两个单声道信号中的每一个都必须被声像调整到两个立体声扬声器/耳机，或者五个或七个环绕声扬声器。因此，每个 3D 语音总线都会终止于两个或多个声像调节器，一个用于干信号，一个或多个用于湿信号。这些分量的声像调整可能有所不同。干信号的声像调整会根据声源的实际位置进行调整。然而，湿信号的声像调整可以更宽，以模拟反射声波从各个方向入射到听众头部的方式。如果声音来自狭窄的门口，湿信号的声像调整可能只有几度。但如果听众站在宽敞大厅的中央，湿信号可能应该被赋予 360 度的声像调整（即，它应该在所有扬声器中均匀地呈现）。

14.5.4 主混音器

每个声像旋钮的输出都是一条多通道总线，包含每个所需输出通道（立体声或环绕声）的信号。游戏中通常会同时播放大量 3D 声音。主混音器会接收所有这些多通道输入，并将它们混合成单个多通道信号，然后输出到扬声器。

interference effects and general attenuation of high-frequency components in the reflected sound waves.

14.5.3.5 Pre-Send Filter

The voice pipeline typically includes a filter that is applied before the aux send bifurcation, and therefore applies to both the dry and wet components of the sound. This is called a *pre-send filter*. It is generally used to model phenomena that occur at the source of the sound. For example, we could mimic the sound of someone wearing a gas mask with a pre-send filter.

14.5.3.6 Post-Send Filter

Another filter is typically provided after the aux send bifurcation. As such, this filter only applies to the dry component of the sound. This filter can be useful for modeling the muffling effect of an obstruction/occlusion on the direct sound path. At Naughty Dog, we also use a post-send filter to implement the frequency-specific fall-off that occurs due to atmospheric absorption (see Section 14.1.3.2).

14.5.3.7 Pan Pots

The dry and wet components of a 3D sound are monophonic signals throughout their journey along the voice bus. At the very end of the pipeline, each one of these two mono signals must be panned to the two stereo speakers/headphones or the five or seven surround sound speakers. For this reason, every 3D voice bus terminates in two or more pan pots, one for the dry signal and one or more for the wet. The components may be panned differently. The dry signal is panned according to the actual location of the source. The wet signal, however, may be panned with a wider focus to simulate the way in which reflected sound waves tend to impinge on the listener's head from various directions. If the sound is coming from a narrow doorway, the focus of the wet signal might be only a few degrees. But if the listener is standing in the center of a cavernous hall, the wet signal should probably be given a 360-degree focus (i.e., it should be rendered in all speakers equally).

14.5.4 Master Mixer

Each pan pot's output is a multi-channel bus, containing signals for each of the desired output channels (stereo or surround sound). A game typically has a large number of 3D sounds playing simultaneously. The *master mixer* takes all of these multi-channel inputs and mixes them together into a single multi-channel signal for output to the speakers.

根据具体实现，主混音器可能由硬件实现，也可能完全由软件实现。如果主混音由硬件实现，声卡设计师可以选择进行模拟混音或数字混音。（软件只能进行数字混音，原因显而易见。）

14.5.4.1 模拟混音

模拟混频器本质上只是一个求和电路 - 将各个输入信号的幅度相加，然后将所得波的幅度衰减至所需的信号电压范围内。

14.5.4.2 数字混音

混音也可以通过专用 DSP 芯片或通用 CPU 上运行的软件以数字方式进行。数字混音器接收多个 PCM 数据流作为输入，并在输出端产生单个 PCM 数据流。

数字混音器的工作比模拟混音器略微复杂一些，因为它所组合的 PCM 通道集合可能以不同的采样率和/或不同的位深度录制。混音器的所有输入信号必须执行两个过程，即采样深度转换和采样率转换，才能转换为通用格式。完成转换后，混音工作就变得轻而易举了。在每个时间点，所有输入样本的值都会被简单地加在一起，并根据需要调整最终输出幅度，以使组合信号达到所需的音量范围。

14.5.4.3 样本深度转换

如果混频器输入信号的位深度不同，可以使用样本深度转换将它们转换为通用格式。此操作很简单。我们只需将输入样本值反量化为浮点格式，然后以所需的输出位深度重新量化每个样本值即可。有关量化的详细信息，请参阅第 12.8.2 节。

14.5.4.4 采样率转换

如果输入信号的采样率不同，则必须使用采样率转换功能，在混音之前将它们全部转换为所需的输出采样率。原则上，这需要将信号转换为模拟形式，然后以所需的采样率进行重采样（可以使用 D/A 和 A/D 硬件完成）。实际上，模拟采样率转换往往会引入不必要的噪声，因此转换几乎总是通过直接在 PCM 数据流上运行直接数模转换算法来完成。

Depending on the specifics of the implementation, the master mixer might be implemented in hardware, or it might live entirely in software. If the master mix is performed in hardware, the sound card designer has the option of performing an analog mix or a digital mix. (Software can only do digital mixing, for obvious reasons.)

14.5.4.1 Analog Mixing

An analog mixer is essentially just an summation circuit—the amplitudes of the individual input signals are added together, and the resultant wave's amplitude is then attenuated to fall back within the desired signal voltage range.

14.5.4.2 Digital Mixing

Mixing can also be performed *digitally* by software running on a dedicated DSP chip or a general-purpose CPU. A digital mixer takes multiple PCM data streams as its inputs, and produces a single PCM data stream at its output.

A digital mixer's job is a little more complicated than that of an analog mixer, because the collection of PCM channels it is combining may have been recorded at different sample rates and/or different bit depths. Two processes known as *sample depth conversion* and *sample rate conversion* must be executed on all of the mixer's input signals to bring them into a common format. Once this has been done, mixing again becomes trivial. At each time index, the values of all the input samples are simply added together, and the final output amplitude is adjusted if necessary to bring the combined signal into the desired volume range.

14.5.4.3 Sample Depth Conversion

If the *bit depths* of the mixer's input signals differ, *sample depth conversion* can be used to convert them to a common format. This operation is trivial. We simply de-quantize the input sample values into floating-point format, and then re-quantize each one at the desired output bit depth. See Section 12.8.2 for all the gory details on quantization.

14.5.4.4 Sample Rate Conversion

If the *sample rates* of the input signals differ, *sample rate conversion* must be used to convert them all into the desired output sample rate prior to mixing. In principle, this involves converting the signal into analog form, and then resampling it at the desired rate (which could be done using D/A and A/D hardware). In practice, analog sample rate conversion tends to introduce unwanted noise, so the conversion is almost always accomplished by running a direct digital-to-digital algorithm directly on the PCM data stream.

图 14.41 典型的主输出总线。

要完全理解这些算法的工作原理，需要理解信号处理理论（参见 14.2 节），本文不作全面讨论。但在某些简单情况下，这个概念很容易理解。例如，如果我们将采样率加倍，可以对相邻的样本进行插值，并将这些值作为新样本插入，从而使样本数量加倍。实际情况并非如此简单——例如，必须注意避免在最终信号中引入混叠。有关采样率转换的详细讨论，请参阅 http://en.wikipedia.org/wiki/Sample_rate_conversion。

14.5.5 主输出总线

语音混音完成后，会由主输出总线进行处理。主输出总线由一系列组件组成，用于在将输出发送到扬声器之前对其进行处理。图 14.41 描绘了一条典型的主输出总线，其组件简要介绍如下。每个音频引擎的处理方式略有不同，并非所有引擎都支持下文所述的所有组件。

一些引擎可能还会引入此处未显示的附加组件。

- **前置放大器。前置放大器允许在主信号通过输出总线的其余部分之前对其进行增益调整。**
- LFE 发生器。正如我们在 14.4.4 节中提到的，声像调节器仅驱动立体声或环绕声系统中的两个、五个或七个“主”扬声器。LFE（低音炮）声道对声音 3D 声像的定位没有影响。LFE 发生器是一个组件，它提取最终混合信号中的最低频率，并用它来驱动 LFE 声道。

Figure 14.41. A typical master output bus.

An understanding of signal processing theory (see Section 14.2) is necessary to fully understand how these algorithms function, and a full discussion is beyond our scope here. But in certain simple cases, the concept is easy enough to grasp. For example, if we are *doubling* the sample rate, we can interpolate adjacent samples and insert these values as new samples, thereby doubling the number of samples. It's not quite as simple as this—one must take care to avoid introducing aliasing into the resulting signal, for example. See http://en.wikipedia.org/wiki/Sample_rate_conversion for a detailed discussion of sample rate conversion.

14.5.5 The Master Output Bus

Once the voices have been mixed, they are processed by the *master output bus*. This is a collection of components that process the output prior to sending it to the speakers. A typical master output bus is depicted in Figure 14.41, and its components are described briefly below. Every audio engine does things a bit differently, and not all engines support all of the components described below. Some engines may also introduce additional components not shown here.

- *Pre-amp*. The pre-amp allows the master signal's gain to be trimmed prior to passing through the remainder of the output bus.
- *LFE generator*. As we mentioned in Section 14.4.4, a pan pot only drives the two, five or seven "main" speakers of a stereo or surround sound system. The LFE (subwoofer) channel does not contribute to the positioning of a sound's 3D image. An *LFE generator* is a component that extracts the very lowest frequencies of the final mixed signal and uses this to drive the LFE channel.

• 均衡器。大多数音频引擎都提供某种均衡器 (EQ)。如第 14.2.5.8 节所述，EQ 允许单独增强或衰减信号中的特定频段。典型的 EQ 会将频谱划分为 4 到 10 个可单独调节的频段。

• 压缩器。压缩器对音频信号执行动态范围压缩 (DRC)。压缩器会降低信号中最大音量部分的音量，并/或提高最小音量部分的音量。它通过分析输入信号的音量特性并动态调整压缩率来自动执行此操作。有关 DRC 的详细讨论，请参阅 http://en.wikipedia.org/wiki/Dynamic_range_compression。

• 主增益控制。此组件允许控制整个游戏的整体音量。

• 输出。主总线的输出是与扬声器通道相对应的线路电平模拟信号和/或数字 HDMI 或 S/PDIF 多通道信号的集合，适合传输到电视或家庭影院系统。

14.5.6 实现总线

14.5.6.1 模拟总线

模拟总线是通过多个并行电子连接实现的。为了传输单声道音频信号，我们需要在电路板上布置两条并行的导线或“线路”：一条用于传输电压信号本身，另一条用于接地。

模拟总线几乎是瞬时工作的。上游组件的输出信号会立即被下游组件接收，因为该信号是一种连续的物理现象。这类电路非常简单。唯一真正复杂的问题是确保输入和输出信号的电压电平和阻抗匹配。

14.5.6.2 数字总线

我们可以想象用简单的数字电路在数字组件之间建立即时连接。然而，这需要连接的组件以完美的同步运行：发送方产生一个字节数据的那一刻，接收方就必须接收它。

否则，该字节将会丢失。
为了克服连接两个数字组件固有的同步问题，通常在输入和/或输出处使用环形缓冲器

- *Equalizer*. Most audio engines provide some kind of *equalizer* (EQ). As described in Section 14.2.5.8, an EQ allows specific frequency bands in the signal to be boosted or attenuated individually. A typical EQ divides the spectrum up into anywhere from four to tens of individually tunable bands.
- *Compressor*. A compressor performs *dynamic range compression* (DRC) on the audio signal. A compressor reduces the volume of the loudest portions of the signal and/or increases the volume of the quietest moments. It does this automatically by analyzing the input signal's volume characteristics and adjusting the compression dynamically. See http://en.wikipedia.org/wiki/Dynamic_range_compression for a detailed discussion of DRC.
- *Master gain control*. This component allows the overall volume of the entire game to be controlled.
- *Outputs*. The output of the master bus is a collection of line-level analog signals corresponding to the speaker channels and/or a digital HDMI or S/PDIF multi-channel signal, suitable for transmission to a TV or a home theater system.

14.5.6 Implementing a Bus

14.5.6.1 Analog Buses

An analog bus is implemented via a number of parallel electronic connections. To carry a monophonic audio signal, we need two parallel wires or “lines” on the circuit board: one to carry the voltage signal itself, and one to serve as ground.

An analog bus operates pretty much instantaneously. The output signal from an upstream component is immediately consumed by the next downstream component, because the signal is a continuous physical phenomenon. Such circuits are quite simple. The only real complication is ensuring that the voltage levels and impedances of the input and output signals match.

14.5.6.2 Digital Buses

One could imagine using simple digital circuitry to build instantaneous connections between our digital components. However, this would require the connected components to run in perfect lock-step: At the exact moment that the sender produces a byte of data, the receiver would have to consume it. Otherwise, the byte would be lost.

To overcome the synchronization problems inherent in connecting two digital components, *ring buffers* are typically used at the input and/or output of

每个组件。环形缓冲区是可由两个客户端（一个读取器和一个写入器）共享的缓冲区。为了使其工作，我们在缓冲区内维护两个指针或索引，称为读取头和写入头。读取器在读取头处使用数据，在数据被使用时在缓冲区中向前推进数据，并在到达缓冲区末尾时包装。写入器将数据存储到写入头处的缓冲区中，同样前进和包装。任何一个头都不允许“传递”另一个，这保证了两个客户端不会相互冲突（即，读取尚未写入的数据，或在当前正在读取的数据之上写入）。

例如，将编解码器的数字输出连接到 DAC 的数字输入最简单的方法是使用共享环形缓冲区。编解码器写入的缓冲区与 DAC 读取的缓冲区完全相同。

共享缓冲区方法虽然简单，但只有当两个组件能够访问相同的物理内存时才有效。当组件在单个 CPU 上的线程中运行时，这一点很容易实现。为了使共享内存方法在两个独立的操作系统进程（每个进程都有自己的私有虚拟内存空间）之间工作，操作系统需要提供一种机制，将相同的物理内存映射到每个进程的虚拟地址空间。这通常只有当两个进程在同一核心上运行，或者在多核计算机的不同核心上运行时才有可能。

如果两个组件运行在不同的内核上，且无法共享内存（例如，一个组件运行在PC上，另一个组件运行在插入式声卡上），则每个组件都需要各自的输入或输出缓冲区。数据必须从一个组件的输出缓冲区复制到另一个组件的输入缓冲区。这可以通过直接内存访问控制器 (DMAC) 来实现，就像在 PS3 的 PPU 和 SPU 之间传输数据时一样。它也可以采用专用总线来实现，例如，在 PC 上，用于连接主 CPU 内核和插入式外设卡的 PCI Express (PCIe) 总线。

14.5.6.3 总线延迟

为了播放声音，游戏或应用程序必须定期将音频数据输入编解码器，最终驱动扬声器输出。我们称之为音频服务。游戏或应用程序的音频服务速率对于正常的声音产生至关重要：如果数据包发送频率过低，缓冲区将下溢，这意味着设备会在新数据包到达之前消耗所有数据。这会导致音频在软件追赶时短暂中断。如果数据包发送过于频繁，PCM 缓冲区可能会溢出，导致数据包丢失。这会使音频听起来像是“跳跃”的。

each component. A ring buffer is a buffer that can be shared by two clients—one *reader* and one *writer*. To make this work, we maintain two pointers or indices within the buffer, called the *read head* and the *write head*. The reader consumes data at the read head, advancing it forward in the buffer as data is consumed, and wrapping when the end of the buffer is reached. The writer stores data into the buffer at the write head, advancing and wrapping as well. Neither head is permitted to “pass” the other, which guarantees that the two clients cannot conflict with one other (i.e., reading data that hasn't been written yet, or writing over top of data that is currently being read).

The simplest way to connect, say, the digital output of a codec to the digital input of a DAC is to use a *shared* ring buffer. The codec writes to the exact same buffer read by the DAC.

While simple, the shared buffer approach only works when the two components have access to the same physical memory. This is trivial to do when the components are running in threads on a single CPU. To make a shared memory approach work between two separate operating system *processes*, each of which has its own private virtual memory space, the OS needs to provide a mechanism for mapping the same physical memory into each process's virtual address space. This is usually only possible when the two processes are running on the same core, or on different cores within a multicore computer.

If the two components are running on different cores that cannot share memory (as would be the case if one were running on the PC and the other on a plug-in sound card, for example), then each component needs its own input or output buffer. Data must be *copied* from the output buffer of one component to the input buffer of the other. This might be accomplished via a *direct memory access controller* (DMAC), as is the case when transferring data between the PPU and the SPUs on the PS3. It might also be accomplished via a specialized bus, such as the ubiquitous PCI Express (PCIe) bus that is used to connect the main CPU core(s) to plug-in peripheral cards on a PC.

14.5.6.3 Bus Latency

In order to play sound, the game or application must feed audio data periodically into the codecs that ultimately drive the speaker outputs. We call this *servicing* the audio. The rate at which the game or app services its audio is crucial to proper sound production: If packets are sent too infrequently, the buffers will *underflow*, meaning that the device consumes all of the data before a new packet arrives. This causes the audio to drop out briefly while the software catches up. If packets are sent too often, the PCM buffers can *overflow*, causing packets to be lost. This causes the audio to seem to “skip.”

构成数字总线的输入和输出缓冲区的大小决定了音响系统的延迟，换句话说，就是总线会引入多少延迟。如果缓冲区非常小，延迟可以最小化，但这会给 CPU 带来更大的负担，因为它必须更频繁地向缓冲区提供数据。同样，更大的缓冲区虽然可以减少 CPU 负载，但延迟也会更高。我们通常以毫秒为单位测量音频硬件的延迟，而不是以字节为单位测量缓冲区的大小。这样做是因为缓冲区大小取决于数据格式和编解码器支持的压缩程度，但在尝试制作高保真音效时，我们真正关心的是延迟。

多少延迟是可以接受的？这取决于具体应用。专业音频系统要求非常短的延迟——大约 0.5 毫秒。这是因为音频信号通常要通过音频硬件网络进行传输，然后才能彼此同步，并且通常还要与视频信号同步。每当硬件引入延迟时，精确同步就会变得更加困难。

另一方面，游戏机可以容忍更长的延迟。在游戏中，我们关心的只是音频和图形的同步。如果游戏以 60 FPS 的速度渲染，则相当于每帧 1/60 = 16.6 毫秒。只要音频延迟不超过 16 毫秒，我们就知道它会与同一帧的渲染图形同步。（事实上，许多游戏的渲染引擎使用双缓冲或三缓冲，这会在游戏请求绘制帧的时间和该帧实际出现在电视屏幕上的时间之间引入一到两帧的延迟。电视也可能引入延迟。因此，三缓冲 60 Hz 游戏实际上可以容忍 $3 \times 16 = 48$ 毫秒或更长的音频延迟。）PlayStation 3 的 DMA 控制器每 5.5 毫秒运行一次，因此 PS3 音频系统通常配置为音频缓冲区可以容纳 5.5 毫秒的整数倍音频。

14.5.7 资产管理

14.5.7.1 音频片段

最基本的音频资源是剪辑（Clip）——一个拥有自己本地时间线的单一数字声音资源（类似于动画剪辑）。剪辑有时也称为声音缓冲区，因为数字样本数据存储在缓冲区中。剪辑可能封装单声道音频数据（通常用于 3D 声音资源），也可能包含多声道音频（通常用于 2D 资源或 3D 立体声音源）。剪辑可以存储为引擎支持的任何音频文件格式。

The size of the input and output buffers that comprise a digital bus dictates the *latency* of the sound system—in other words, how much delay is introduced by the bus. If the buffers are very small, latency is minimized, but this places a greater burden on the CPU because it must feed the buffers more frequently. Likewise, larger buffers translate into less CPU load but also higher latency. We usually measure the latency of a piece of audio hardware in milliseconds, rather than measuring the size of the buffers in bytes. This is done because buffer size depends upon the data format and the degree of compression supported by the codec, but the latency is really what we care about when trying to produce high-fidelity sound.

How much latency is acceptable? This depends on the application. Professional audio systems require very short latencies—on the order of 0.5 ms. This is because audio signals are often fed through a network of audio hardware before being synchronized with each other and often to a video signal as well. Every time latency is introduced by the hardware, accurate synchronization becomes more difficult.

Game consoles, on the other hand, can tolerate a longer latency. In a game, all we care about is synchronizing the audio and the graphics. If the game is rendering at 60 FPS, this translates into $1/60 = 16.6$ ms per frame. As long as the audio isn't delayed by more than 16 ms, we know it will be in sync with graphics rendered for that same frame. (In fact, many games use double or triple buffering for their rendering engines, which introduces one or two frames of delay between the time the game requests that a frame be drawn and when that frame will actually appear on the TV screen. The television may also introduce a delay. As such, a triple-buffered 60 Hz game can actually tolerate an audio latency of $3 \times 16 = 48$ ms or more.) The PlayStation 3's DMA controller runs every 5.5 ms, so PS3 audio systems are typically configured such that the audio buffers can hold an integer multiple of 5.5 ms worth of audio.

14.5.7 Asset Management

14.5.7.1 Audio Clips

The most atomic audio asset is a *clip*—a single digital sound asset with its own local timeline (analogous to an animation clip). A clip is sometimes called a *sound buffer*, because the digital sample data is stored in a buffer. A clip might encapsulate monophonic audio data (typical for 3D sound assets), or it might contain multi-channel audio (typically used for 2D assets or stereo sound sources in 3D). A clip may be stored in any of the audio file formats supported by your engine.

14.5.7.2 声音提示

声音提示是音频片段及其元数据的集合，元数据描述了这些片段的处理和播放方式。提示通常是游戏请求播放声音的主要方式。（引擎可能支持或不支持播放单个片段。）提示也是一种便捷的分工机制：声音设计师可以使用离线工具制作提示，而无需担心它们在游戏中如何播放或何时播放。游戏程序员可以根据游戏中的相关事件方便地播放提示，而无需操心播放细节的琐碎管理。

有很多方法可以解释和播放 Cue 中的片段集合。Cue 可能包含代表预混 5.1 音乐录音的六个声道的片段。Cue 也可能收集一组原始声音，以便从中进行随机选择，以增加多样性。Cue 还可以设置为按预定义的顺序播放其原始声音集合。Cue 通常会指定其封装的声音是单次播放还是循环播放。

一些音频引擎允许提示提供一个或多个可选的声音片段，这些片段仅在主声音播放过程中中断时播放。例如，语音提示可能包含“声门塞音”，仅在人物对话中断时播放。此功能还可用于提供独特的“尾音”，在循环提示停止时播放。例如，循环播放的机枪声音提示可以使用此“尾音”片段功能，在射击停止时产生合适的回声衰减音。

提示的元数据可能包括其预期以 3D 还是 2D 播放；声源的 FO 最小值、FO 最大值和衰减曲线；群组成员（参见第 14.5.8.1 节）；以及播放声音时可能应用的任何特效、滤波或均衡。在索尼的 Scream 引擎（顽皮狗在其《神秘海域》和《最后生还者》系列中使用的声音引擎）中，提示可以包含任意脚本代码，允许声音设计师完全控制提示播放时封装的声音资源的播放方式。

播放提示

每个支持提示概念的音频引擎都提供了播放提示的 API。此 API 通常是游戏代码请求播放声音的主要方式，有时也是唯一方式。

提示播放 API 通常允许程序员指定提示是否应以 2D 或 3D 声音播放，以提供 3D 位置和

14.5.7.2 Sound Cues

A *sound cue* is a collection of audio clips plus metadata that describes how they should be processed and played. Cues are usually the primary means by which the game can request sounds to be played. (Playing individual clips may or may not be supported by the engine.) Cues also serve as a convenient division-of-labor mechanism: The sound designers can craft the cues using an offline tool, without having to worry about how or when they'll be played in-game. And the game programmers can play the cues conveniently in response to relevant events in the game, without having to worry about micromanaging the details of playback.

There are many ways in which the collection of clips in a cue could be interpreted and played back. A cue might contain clips representing the six channels of a pre-mixed 5.1 music recording. A cue might also collect up a set of raw sounds, from which a random selection can be made, for the sake of variety. A cue might also be set up to play its collection of raw sounds in a predefined sequence. A cue typically specifies whether the sound(s) it encapsulates represent a one-shot sound or a looping sound.

Some audio engines permit a cue to provide one or more optional sound clips that only play if the main sound is *interrupted* midway through playing. For example, a vocal cue might include a “glottal stop” sound that is played only if the person's line of dialog is interrupted. This feature can also be used to provide a distinct “tail” sound that is played when a looping cue is stopped. For example, a looping machine gun sound cue might use this “tail” clip feature to produce a suitable echoing fall-off sound when the firing ceases.

A cue's metadata might include whether it is intended to be played in 3D or 2D; the FO min, FO max and fall-off curve of the sound source; group membership (see Section 14.5.8.1); and possibly any special effects, filtering or equalization that should be applied when the sound is played. In Sony's Scream engine—the sound engine used by Naughty Dog in its *Uncharted* and *The Last of Us* series—a cue can contain arbitrary script code that allows a sound designer to completely control how the encapsulated sound asset(s) are played when the cue plays.

Playing a Cue

Every audio engine that supports the concept of cues provides an API for playing them. This API is usually the primary way—and sometimes the *only* way—for the game code to request that a sound be played.

The cue playing API generally allows the programmer to specify whether the cue should be played as a 2D or 3D sound, to provide 3D position and

速度参数，用于指定声音是循环播放还是仅播放一次，以及指定源缓冲区是内存缓冲区还是流式缓冲区。该 API 通常还允许我们控制音量以及其他可能的播放方面。

大多数 API 都会向调用者返回一个声音句柄。该句柄允许程序在声音播放时跟踪声音，以便在声音结束前对其进行修改或取消。声音句柄通常在底层实现为全局句柄表的索引，而不是指向描述声音实例的数据的原始指针。这样，如果声音自然结束，句柄就会自动“清零”。句柄机制还可以用于确保系统线程安全——如果一个线程终止了声音，其他拥有该声音句柄的线程将自动发现其句柄失效。

14.5.7.3 声音库

3D 音频引擎管理着大量的资源。游戏世界包含大量的物体。每个物体都能产生各种各样的声音。除了 3D 音效之外，我们还有音乐、语音、菜单音效等等。

所有这些音频数据占用了大量空间，我们无法一次性将它们全部保存在内存中。另一方面，单个音频片段的粒度太细，数量太多，无法单独管理。因此，大多数游戏引擎会将其声音片段和提示打包成称为“声音库”的粗略单元。

有些音效库会在游戏启动时加载并永久保存在内存中。例如，玩家角色发出的声音集合始终是必需的，因此我们可以将它们无限期地保存在内存中。其他音效库可能会随着游戏需求的变化而动态地加载和卸载。例如，关卡 A 中的声音可能不会在关卡 B 中使用，因此我们可以仅在播放关卡 A 时加载“A”音效库。例如，在顽皮狗的《最后生还者》中，只有当玩家位于波士顿倾斜的建筑物中时，才会加载雨声、流水声和即将倒塌的横梁的吱嘎声。

某些声音引擎允许在内存中重新定位 SoundBank。此功能可以完全消除游戏过程中加载和卸载大量不同大小的 SoundBank 所造成的内存碎片问题。有关内存重新定位的更多信息，请参阅第 6.2.2.2 节。

velocity parameters, to specify whether the sound should loop or play only once, and to specify whether the source buffer is in-memory or streamed. The API usually also allows us to control the volume of the sound and possibly other aspects of playback.

Most APIs return a *sound handle* to the caller. This handle allows the program to keep track of a sound as it is playing, so that it can be modified or canceled before the sound ends. A sound handle is usually implemented under the hood as an index into a global handle table, rather than as a raw pointer to the data that descibes the sound instance. That way, if the sound ends naturally, the handle can be “nulled out” automatically. A handle mechanism can also be used to make the system thread-safe—if one thread kills the sound, other threads that have handles to the sound will automatically see their handles become invalid.

14.5.7.3 Sound Banks

A 3D audio engine manages a *lot* of assets. The game world contains a large number of objects. Each object can generate a variety of sounds. And in addition to 3D sound effects, we have music, speech, menu sound effects and so on.

All of this audio data takes up an immense amount of space, so we can’t keep it all in memory at once. On the other hand, the individual audio clips are too fine-grained and too numerous for them to be managed on an individual basis. As such, most game engines package their sound clips and cues into coarse units called *sound banks*.

Some sound banks are loaded when the game starts up and left in memory forever. For example, the collection of sounds made by the player character are always needed, so we could keep them in memory indefinitely. Other banks might be loaded and unloaded dynamically as the needs of the game change. For example, the sounds in level A might not be used in level B, so we can load the “A” bank only when level A is being played. For example, in Naughty Dog’s *The Last of Us*, the sounds of rain, flowing water and the creaking of beams on the verge of collapse were only loaded when the player was in the tilted building in Boston.

Some sound engines permit banks to be *relocated* in memory. This feature can entirely eliminate the memory fragmentation problems that would otherwise arise as lots of differently sized banks are loaded and unloaded during gameplay. See Section 6.2.2.2 for more information on memory relocation.

14.5.7.4 流声音

有些声音持续时间很长，无法方便地一次性全部存储在内存中。音乐和语音就是常见的例子。对于这类声音，许多游戏引擎都支持流音频。

流式音频之所以可行，是因为播放声音时，我们真正需要的唯一信息是当前时间点及其附近的信号数据。为了实现流式传输，我们为每个流式声音维护一个相对较小的环形缓冲区。在播放声音之前，我们会将一小部分数据预加载到缓冲区中，然后正常播放。音频管道会在播放过程中消耗环形缓冲区中的数据，从而腾出空间来加载更多数据。只要我们在缓冲区数据耗尽之前不断填充数据，声音就能无缝播放。

14.5.8 混合你的游戏

如果我们播放来自每个游戏对象的所有声音，并进行适当的衰减、空间化和声学建模，使用我们目前讨论过的所有技巧和技术，结果会怎样？我们或许会期待这个问题的答案是“一个令人难以置信的沉浸式、逼真的音景，能让我们获奖并致富！”但实际上，我们听到的只是一片嘈杂。

一款好游戏与一款卓越游戏的区别在于混音——你听到了什么，听到了多少，以及同样重要的，你听不到什么。游戏音效设计师的目标是制作出最终混音，使其达到以下效果：

- 听起来逼真且引人入胜；
- 不会太分散注意力、太烦人或难以听清；
- 有效传达与游戏玩法和/或故事相关的所有信息；以及
- 考虑到游戏中发生的事件和游戏的整体设计，始终保持适当的情绪和色调。

游戏中必须融合各种不同的声音，包括音乐、语音、环境音效（例如雨声、风声、昆虫声或老建筑的嘎吱声）、武器射击、爆炸和车辆等音效，以及物理模拟物体的碰撞、滑动和滚动声。

我们采用了各种技术来确保游戏的混音符合这些目标。我们将在以下章节中探讨其中的一些技术。

14.5.7.4 Streaming Sounds

Some sounds are so long in duration that they cannot be conveniently stored in memory all at once. Music and speech are common examples. For these kinds of sounds, many game engines support *streaming audio*.

Streaming audio is possible because when playing a sound, the only information we actually need is the signal data at and around the current time index. To implement streaming, we maintain a relatively small ring buffer for each streaming sound. Prior to playing the sound, we pre-load a small chunk of it into the buffer, then play the sound normally. The audio pipeline consumes the data from the ring buffer as it plays, making room for us to load more data. As long as we keep filling the buffer with data before it is all consumed, our sound will play seamlessly.

14.5.8 Mixing Your Game

If we were to play every sound coming from every game object, properly attenuated and spatialized and acoustically modeled, using all the techniques and technology we've discussed thus far, what would be the result? We might expect the answer to this question to be "an incredibly immersive and believable soundscape that wins awards and makes us rich!" But what we'd actually get is cacophony.

What separates a good game from a great game is the *mix*—what you hear, how much of it you hear, and just as importantly what you don't hear. The goal of a game's sound designer is to produce a final mix that:

- sounds realistic and emersive;
- isn't too distracting, annoying or difficult to listen to;
- conveys all information relevant to gameplay and/or story effectively; and
- maintains a mood and tonal color that is always appropriate, given the events taking place in the game and the overall design of the game.

All sorts of different kinds of sounds must come together in the game's mix. These include music, speech, ambient sounds like rain, wind, insects or the creak of an old building, sound effects such as weapons fire, explosions and vehicles, and the bumps, slides and rolling sounds made by physically simulated objects.

Various techniques are employed to ensure the mix of the game meets these goals. We'll explore a few of them in the following sections.

14.5.8.1 组

为了提升游戏混音效果，最显而易见的做法就是合理设置 3D 世界中所有源声音的音量。这里最重要的是确保每个声音的增益相对于游戏中的其他声音而言是合适的。例如，脚步声应该比枪声更安静。

在某些游戏中，某些声音的音量需要动态变化。我们通常希望一次性控制所有类型的声音。例如，在激烈的打斗场景中，我们可能希望提高音乐和武器的音量，并降低辅助音效的音量。或者，在角色之间安静对话的时刻，我们可能希望稍微增强对话音量，并降低环境音，以确保对话清晰可闻。

因此，许多音频引擎都支持“组”的概念——这个概念是从我们的老朋友多通道混音台“偷”来的。在混音板上，一组声音输入可以被路由到一个中间混音器电路，并将它们组合成一个单一的“组信号”。然后，该信号的增益可以通过混音板上的一个旋钮来控制，从而使音响工程师能够同时控制所有输入信号的响度。

在软件世界中，群组的实现方式是简单地对声音提示进行分类，而不是将它们的信号物理地混合在一起。例如，我们可以将一个提示分类为音乐、音效、武器、台词等等。然后，引擎可以提供一种方式，使用单个控制值来控制每个类别中所有声音的增益。群组通常还允许通过简单的 API 调用方便地暂停、重新启动和静音整个类别的声音。

一些声音引擎确实提供了一种机制，可以将多组音频信号物理混合成单个信号，就像在混音台上处理群组信号一样。在索尼的 Scream 引擎中，这被称为生成预主子混音。在子混音锁定组中信号的相对增益后，生成的信号可以通过额外的滤波器或其他处理阶段进行路由。这使得声音设计师能够更好地控制游戏的混音。

14.5.8.2 闪避

闪避是一种暂时降低某些声音的音量/增益，以使其他声音更容易听见的技术。例如，当角色说话时，可以自动降低背景噪音的水平，以使对话更容易听见。

14.5.8.1 Groups

The most obvious thing we can do to improve the mix of our game is to set the levels of all the source sounds in the 3D world appropriately. The important thing here is to ensure that each sound's gain is appropriate *relative* to the other sounds in the game. For example, footsteps should be quieter than gunfire.

In some games, the loudness of certain sounds needs to change dynamically. Often we want to control an entire category of sounds at once. For example, during a frenetic fight sequence, we might want to bring up the levels of the music and the weapons, and drop the volume of ancillary sound effects. Or during a quiet moment with characters talking to one another, we might want to boost the speech a little and tone down ambient sounds to ensure the dialog can be heard.

For this reason, many audio engines support the concept of *groups*—a concept "stolen" from our old friend the multi-channel mixing console. On a mixing board, a collection of sound inputs can be routed to an intermediate mixer circuit, combining them into a single "group signal." The gain of this signal can then be controlled with a single knob on the board, thereby allowing the sound engineer to control the loudness of all input signals at once.

In the software world, groups are implemented by simply categorizing sound cues, rather than physically mixing their signals together. For example, we can classify a cue as being music, a sound effect, a weapon, a line of speech and so on. Then, the engine can provide a means of controlling the gains of all sounds in each category with a single control value. Groups usually also allow entire categories of sound to be paused, restarted and muted conveniently with a simple API call.

Some sound engines do also provide a mechanism for physically mixing groups of audio signals into a single signal, just as is done when working with groups on a mixing console. In Sony's Scream engine, this is called generating a *pre-master submix*. After the relative gains of the signals in the group have been locked down by the submix, the resulting signal can be routed through additional filters or other processing stages. This gives the sound designer even more control over the mix of the game.

14.5.8.2 Ducking

Ducking is a technique in which the volume/gain of certain sounds are temporarily reduced in order to make other sounds more audible. For example, when a character is speaking, the level of background noise could be reduced automatically to make the dialog more audible.

闪避可以通过多种方式触发。一种特定类型的声音出现时，可能会触发另一种类型的声音闪避。游戏事件可以通过编程方式触发闪避。任何被认为合适的触发机制都可以用来启动闪避。

闪避操作引起的音量降低通常是通过群组分类系统实现的：当播放某一类别的声音时，它可以自动闪避一个或多个其他类别的声音，闪避幅度各不相同。或者，游戏代码可以调用函数以编程方式闪避一组声音。

也可以通过将一个声音信号路由到不同声音总线上动态范围压缩器 (DRC) 的侧链输入来实现闪避。回想一下 14.5.5 节，DRC 会分析信号的音量特性，并自动适当压缩信号的响度。当侧链输入连接到 DRC 时，它会分析侧链信号来决定如何调整音量。因此，我们可以安排一个信号的响度增加，从而降低另一个信号的动态范围。

14.5.8.3 总线预设和混音快照

许多声音引擎允许声音设计师设置配置参数，保存它们，然后在运行时方便地调用和应用。在索尼的 Scream 引擎中，这些参数有两种基本类型：总线预设和混音快照。

总线预设是一组配置参数，用于控制单条总线（语音总线或主输出总线）上组件的各个方面。例如，一个总线预设可能描述一种特定的混响设置，用于模拟大型开放式大厅、汽车内部或小型杂物间的声学效果。或者，一个总线预设可以控制主输出总线上的 DRC 设置。声音设计师可以创建许多这样的预设，并根据游戏需要在运行时激活相应的预设。

混音快照与增益控制的理念类似。可以预先设定组内各个通道的增益，然后在运行时根据需要应用。

14.5.8.4 实例限制

实例限制是一种控制允许同时播放的声音数量的方法。例如，即使 20 个 NPC 同时开枪，我们也可能只播放距离听众最近的三到四种枪声。实例限制很重要，原因有二：首先，它可以很好地防止出现杂音。其次，声音引擎通常只支持固定数量的同步声音，这可能是由于硬件原因

Ducking can be triggered in numerous ways. The presence of one particular type of sound might be used to duck another category of sounds. A game event might trigger a duck programmatically. Any triggering mechanism that is deemed appropriate can be used to initiate a duck.

The reduction of volume caused by a duck is typically accomplished via the group categorization system: When one category of sound is playing, it can automatically duck one or more other categories by various amounts. Or the game code can call a function to duck a group of sounds programmatically.

Ducking can also be performed by routing one sound signal into the *side-chain* input of the dynamic range compressor (DRC) on a different voice's bus. Recall from Section 14.5.5 that a DRC analyzes the volume characteristics of a signal and automatically compresses the loudness of the signal appropriately. When a side-chain input is connected to a DRC, it analyzes the *side-chain* signal when deciding how to adjust the volume. So, we can arrange for increased loudness in one signal to cause a decrease in the dynamic range of another signal.

14.5.8.3 Bus Presets and Mix Snapshots

Many sound engines allow the sound designer to set up configuration parameters, save them off and then recall and apply them conveniently at runtime. In Sony's Scream engine, these come in two basic flavors: *bus presets* and *mix snapshots*.

A bus preset is a set of configuration parameters that control aspects of the components on a single bus (voice bus or master output bus). For example, a bus preset might describe one particular reverb set-up that mimics, say, the acoustics of a large open hall, or the interior of a car, or a small broom closet. Or a bus preset might control the DRC settings on the master output bus. Many such presets can be created by the sound designer, and the appropriate ones activated at runtime as the game requires.

A mix snapshot is the same kind of idea applied to gain control. The gains of the various channels within a group can be established a priori and then applied at runtime as needed.

14.5.8.4 Instance Limiting

Instance limiting is a means of controlling the number of sounds that are permitted to play simultaneously. For example, even though 20 NPCs are all firing their guns at once, we might only play the three or four gun sounds that are nearest to the listener. Instance limiting is important for two reasons: First, it's a great way to prevent cacophony. Second, a sound engine typically supports only a fixed number of simultaneous voices, either because of hardware

限制（例如，声卡只有 N 个编解码器）或由于软件中的内存或处理器带宽限制，所以我们必须明智地使用它们。

每组限制

实例限制有时会针对不同的声音组应用不同的效果。例如，我们可能会指定最多同时播放四声枪响，最多同时听到三个人说话，最多同时播放五种其他音效，以及最多两条重叠的音乐轨道。

优先排序和语音窃取

在包含大量动态元素的 3D 游戏中，任何给定时刻播放的声音数量可能超过系统所能容纳的声音数量。某些声音引擎支持大量（甚至无限多）的虚拟声部。每个虚拟声部代表一种理论上正在播放的声音，但可以将其静音或暂时停止，以免占用宝贵的硬件或软件资源。引擎会使用各种标准来动态确定在给定时刻哪些虚拟声部应映射到“真实”声部。

限制同时播放的声音数量的最简单方法之一是为每个 3D 声源分配一个最大半径。正如我们在 14.4.3.1 节中看到的，这就是 FO 最大半径。如果听者与声音的距离超出了此距离，则该声音被视为听不见，其虚拟声部会暂时静音或停止，从而释放其资源以供其他声部使用。自动静音虚拟声部的过程称为“声音窃取”。

另一种常见的方法是为每个提示或提示组分配一个优先级。当同时播放过多的虚拟声音时，优先级较低的声音可能会被静音（窃取），以支持优先级较高的声音。

声音引擎还可以提供各种其他机制来控制语音窃取算法的细节。例如，可以给某个提示设定一个最短播放时间，超过该时间后，其语音将被窃取。当语音被窃取时，声音可能会淡出而不是突然中断。此外，某些提示可能会被暂时标记为“不可窃取”，以确保即使优先级设置不同，它们仍能播放。

14.5.8.5 混合游戏内过场动画

在正常的游戏条件下，听者或“虚拟麦克风”通常位于摄像机位置或附近，声源会根据其在环境中的实际位置进行建模。基于距离的衰减、直接和间接声路的确定、语音限制——所有这些

limitations (e.g., the sound card only has *N* codecs) or because of memory or processor bandwidth limitations in the software, so we must use them wisely.

Per-Group Limits

Instance limiting is sometimes applied differently to different groups of sounds. For example, we might specify that we should play up to four guns simultaneously, hear up to three people speaking at a time and allow up to five other sound effects to play at once, plus up to two overlapping music tracks.

Prioritization and Voice Stealing

In a 3D game with lots of dynamic elements, there may be more sounds playing at any given time than the system has voices for. Some sound engines support a large number (or even an infinite number) of *virtual voices*. Each virtual voice represents a sound that is technically playing, but that can be muted or stopped temporarily so it ceases to occupy valuable hardware or software resources. The engine uses various criteria to dynamically determine which virtual voices should be mapped to “real” voices at any given moment.

One of the simplest ways to limit the number of sounds playing simultaneously is to assign a maximum radius to every 3D sound source. As we saw in Section 14.4.3.1, this is the FO max radius. If the listener is beyond this distance from the sound, it is considered inaudible and its virtual voice is temporarily muted or stopped, freeing its resources for use by other voices. The process of automatically silencing virtual voices is called *voice stealing*.

Another common approach is to assign each cue or group of cues a *priority*. When too many virtual voices are playing at once, those with lower priorities can be silenced (stolen) in favor of higher priority voices.

Sound engines may also provide various other mechanisms for controlling the details of the voice stealing algorithm. For example, a cue might be given a minimum play time, after which its voice is permitted to be stolen. Sounds might be faded out rather than cut off abruptly when their voice is stolen. And some cues might be temporarily marked as “unstealable” to ensure that they play, even despite their priority settings.

14.5.8.5 Mixing In-Game Cinematics

Under normal gameplay conditions, the listener or “virtual microphone” is typically positioned at or near the location of the camera, and the sound sources are modeled where they really are in the environment. Distance-based attenuation, direct and indirect sound path determination, voice limiting—all

是使用这些现实位置来确定的。

然而，在游戏过场动画（游戏中暂停玩家控制以展开剧情的环节）中，镜头经常会从玩家头部向外平移。这种情况往往会对我们的3D音频系统造成严重破坏。我们可以将监听器/麦克风锁定在摄像机的位置；但这并不总是合适的。例如，如果有一个两个角色说话的远景镜头，我们可能仍然希望进行混音，以便即使角色的声音在物理上距离太远而无法被听到，也能被听到。在这种情况下，我们可能需要将监听器与摄像机分离，并人为地将其放置在更靠近角色的位置。

游戏过场动画的混音与电影混音非常相似。因此，声音引擎需要能够“打破常规”，做出一些并非物理上真实的效果。

14.5.9 音频引擎调查

现在应该已经显而易见，开发一个 3D 音频引擎是一项艰巨的任务。幸运的是，许多人已经为此付出了巨大的努力，并由此诞生了一系列几乎开箱即用的音频软件。从低级音效库到功能齐全的 3D 音频渲染引擎，应有尽有。

在接下来的章节中，我们将介绍一些最常见的音频库和引擎。其中一些是特定于目标平台的，而另一些则是跨平台的。

14.5.9.1 Windows：通用音频架构

在 PC 游戏发展的早期，不同平台和不同供应商的 PC 声卡的功能集和架构差异巨大。微软试图将所有这些差异封装在其 DirectSound API 中，并由 Windows 驱动模型 (WDM) 和内核音频混音器 (KMixer) 驱动程序提供支持。然而，由于供应商无法就通用的功能集或标准接口集达成一致，相同的功能在不同的声卡上往往以截然不同的方式实现。这要求操作系统管理大量不兼容的驱动程序接口。

在 Windows Vista 及更高版本中，微软引入了一项名为通用音频架构 (UAA) 的新标准。标准 UAA 驱动程序 API 仅支持有限的硬件功能，其余所有功能均由软件实现（尽管硬件制造商

are determined using these realistic positions.

However, during an *in-game cinematic*—a portion of the game in which player control is suspended so that a story moment can take place—the camera often pans out away from the player's head. This kind of thing tends to wreak havoc with our 3D audio system. We could just keep the listener/mic locked to the location of the camera; but this is not always appropriate. For example, if there's a long shot of two characters speaking, we probably still want to mix so that the characters' voices can be heard, even though physically speaking they are too far away to be heard. In this case, we might want to detach the listener from the camera, and artificially position it nearer to the characters.

Mixing in-game cinematics is a lot closer to mixing a film. As such, a sound engine needs to be capable of "breaking the rules" and doing things that aren't necessarily physically realistic.

14.5.9 Audio Engine Survey

It should be evident by now that creating a 3D audio engine is a massive undertaking. Luckily for us, lots of people have already put a great deal of effort into this task, and the result is a wide range of audio software that we can use pretty much out of the box. This ranges from low-level sound libraries all the way to fully featured 3D audio rendering engines.

In the following sections, we'll survey a few of the most common audio libraries and engines. Some of these are specific to a particular target platform, while others are cross-platform.

14.5.9.1 Windows: The Universal Audio Architecture

In the early days of PC gaming, the feature set and architecture of PC sound cards varied a great deal from platform to platform and vendor to vendor. Microsoft attempted to encapsulate all of this diversity within its DirectSound API, supported by the Windows Driver Model (WDM) and the Kernel Audio Mixer (KMixer) driver. However, because vendors could not agree on a common feature set or set of standard interfaces, the same functionality would often be realized in very different ways on different sound cards. This required the operating system to manage a very large number of incompatible driver interfaces.

For Windows Vista and beyond, Microsoft introduced a new standard called the Universal Audio Architecture (UAA). Only a limited set of hardware features are supported by the standard UAA driver API—all remaining features are implemented in software (although hardware manufacturers

仍然可以自由地提供额外的“硬件加速”功能，只要它们提供自定义驱动程序来启用它们即可。虽然 UAA 的引入限制了 Creative Labs 等知名声卡供应商的竞争优势，但它确实达到了预期的效果，即创建了一个可靠、功能丰富的标准，游戏和 PC 应用程序可以方便地使用该标准。

UAA 标准对用户的听觉体验还有另一个积极的影响。在旧版 DirectSound 时代，游戏可以完全控制声卡，这意味着来自操作系统或其他应用程序（例如电子邮件程序）的声音将被“锁定”，游戏运行时它们的声音将无法播放。新的 UAA 架构允许操作系统完全控制通过 PC 扬声器听到的最终混音。多个应用程序终于可以以合理一致的方式共享声卡。在线搜索“通用音频架构”以查找有关 UAA 的更多信息。

在 Windows 上，UAA 由所谓的 Windows 音频会话 API（简称 WASAPI）实现。此 API 并非真正供游戏使用。它仅支持软件中的大多数高级音频处理功能，并且对硬件加速的支持有限。游戏通常会使用 XAudio2 API，下一节将对此进行介绍。

14.5.9.2 XAudio2

XAudio 2 是一款功能强大的低级 API，可用于访问 Xbox 360、Xbox One 和 Windows 上的音频硬件。它取代了 DirectAudio，并提供各种硬件加速功能，包括可编程 DSP 效果、子混音、支持各种压缩和未压缩音频格式，以及多速率处理以减轻主 CPU 的负载。

XAudio2 API 之上有一个名为 X3DAudio 的 3D 音频渲染库。这些 API 也适用于 Windows 平台的 PC 游戏。微软曾提供一款名为“跨平台音频创作工具”（简称 XACT）的强大音频创作工具，旨在与 XNA Game Studio 配合使用，但 XNA 和 XACT 均已不再受支持。

14.5.9.3 Scream 和 BoomRangBuss

在 PS3 和 PS4 上，顽皮狗使用索尼的 3D 音频引擎 Scream 及其合成器库 BoomRangBuss。

PlayStation 3 上的音频硬件非常类似于符合 UAA 标准的音频设备，支持最多八个音频通道，实现完整的 7.1 环绕声

are still free to provide additional “hardware acceleration” features, as long as they provide custom drivers to expose them). Although the introduction of UAA limited the competitive advantage of prominent sound card vendors like Creative Labs, it did have the desired effect of creating a solid, feature-rich standard, which could be used by games and PC applications in a convenient way.

The UAA standard had another positive effect on the user’s aural experience. In the old DirectSound days, a game could take complete control of the sound card, meaning that sounds coming from the OS or other applications such as an email program would be “locked out” and their sounds would not play while the game was running. The new UAA architecture allowed the OS to claim ultimate control over the final mix heard through the PC’s speakers. Multiple applications could finally share the sound card in a reasonable and consistent way. Search online for “Universal Audio Architecture” to find more information on UAA.

The UAA is implemented on Windows by the so-called Windows Audio Session API, or WASAPI for short. This API is not really intended for use by games. It supports most advanced audio processing features in software only, with limited support for hardware acceleration. Instead, games usually make use of the XAudio2 API, which is described in the next section.

14.5.9.2 XAudio2

XAudio 2 is the high-powered low-level API that provides access to the audio hardware on Xbox 360, Xbox One and Windows. It replaces DirectAudio and provides access to a wide range of hardware-accelerated features including programmable DSP effects, submixing, support for a wide range of compressed and uncompressed audio formats, and multirate processing to lighten the load on the main CPU(s).

Atop the XAudio2 API sits a 3D audio rendering library called X3DAudio. These APIs are also available on the Windows platform for use by PC games. Microsoft used to offer a powerful audio authoring tool called the “cross-platform audio creation tool” or XACT for short, which was intended for use with XNA Game Studio, but neither XNA nor XACT are supported any longer.

14.5.9.3 Scream and BoomRangBuss

On the PS3 and PS4, Naughty Dog uses Sony’s 3D audio engine Scream and its synth library, BoomRangBuss.

The audio hardware on a PlayStation 3 is very much like a UAA-compliant audio device, supporting up to eight channels of audio for full 7.1 surround

音效支持，以及硬件混音器和 HDMI、S/PDIF、模拟和 USB/蓝牙输出。该音频硬件封装在一系列 OS 级库中，包括 libaudio、libsynth 和 libmixer。基于这些库，游戏开发者可以自由地实现自己的音频软件堆栈。索尼还提供了一款名为 Scream 的强大 3D 音频堆栈，游戏工作室可以“开箱即用”。Scream 可在 PS3、PS4 和 PSVita 平台上使用。其架构模拟了功能齐全的多通道混音控制台。

在《惊声尖叫》的基础上，顽皮狗还为《神秘海域》和《最后生还者》系列游戏实现了一套专有的3D环境音频系统。该系统提供随机的声障/遮挡建模和基于门户的音频渲染系统，能够渲染高度逼真的音景。

高级 Linux 声音架构

UAA 驱动程序模型在 Linux 中的对应模型称为高级 Linux 声音架构 (ALSA)。该 Linux 内核组件取代了原有的开放声音系统 (OSSv3)，成为向应用程序和游戏提供音频功能的标准方式。有关 ALSA 的更多信息，请参阅 http://www.alsa-project.org/main/index.php/Main_Page。

QNX 声音架构

QNX 声音架构 (QSA) 是 QNX Neutrino 实时操作系统的驱动级音频 API。作为一名游戏程序员，您可能永远不会使用 QNX。但它的文档确实很好地阐述了音频硬件的概念和典型功能集。请参阅 http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_audio%2Fmixer.html 获取这些文档。

14.5.9.4 多平台 3D 音频引擎

目前市面上有许多功能强大、随时可用的跨平台 3D 音频引擎。我们将在下文中概述其中最著名的一些。

- OpenAL 是一个跨平台的 3D 音频渲染 API，其设计旨在模仿 OpenGL 图形库。该库的早期版本是开源的，但现在是经过授权的软件。许多供应商提供 OpenAL API 规范的实现，包括 OpenAL Soft (http://kcat.strangesoft.net/openal.html) 和 AeonWave-OpenAL (http://www.adalin.com)。

sound support, plus a hardware mixer and HDMI, S/PDIF, analog and USB/Bluetooth outputs. This audio hardware is encapsulated by a collection of OS-level libraries including libaudio, libsynth and libmixer. On top of these libraries, game makers are free to implement their own audio software stacks. Sony also provides a powerful 3D-capable audio stack of its own called *Scream* which game studios can use “out of the box.” Scream is available on the PS3, PS4 and PSVita platforms. Its architecture mimics a fully featured multi-channel mixer console.

On top of Scream, Naughty Dog implemented a proprietary 3D environmental audio system for use on the *Uncharted* and *The Last of Us* series. This system provides stochastic obstruction/occlusion modeling and a portal-based audio rendering system that permits rendering a highly realistic soundscape.

Advanced Linux Sound Architecture

The Linux equivalent of the UAA driver model is called the Advanced Linux Sound Architecture (ALSA). This Linux kernel component replaced the original Open Sound System (OSSv3) as the standard way to expose audio functionality to applications and games. See http://www.alsa-project.org/main/index.php/Main_Page for more information on ALSA.

QNX Sound Architecture

QNX Sound Architecture (QSA) is a driver-level audio API for the QNX Neutrino real-time OS. As a game programmer, you'll probably never use QNX. But its documentation does provide an excellent picture of the concepts and the typical feature set of audio hardware. See http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_audio%2Fmixer.html for these docs.

14.5.9.4 Multiplatform 3D Audio Engines

A number of powerful, ready-to-use cross-platform 3D audio engines are available. We'll outline the most well-known of these below.

- *OpenAL* is a cross-platform 3D audio rendering API that has been deliberately designed to mimic the design of the OpenGL graphics library. Early versions of the library were open source, but it is now licensed software. A number of vendors provide implementations of the OpenAL API spec, including OpenAL Soft (http://kcat.strangesoft.net/openal.html and AeonWave-OpenAL (http://www.adalin.com).

• AeonWave 4D 是 Adalin BV 为 Windows 和 Linux 开发的低成本音频库

• FMOD Studio 是一款音频创作工具，具有“专业音频”的外观和体验 (http://www.fmod.org)。功能齐全的运行时 3D 音频 API 允许在 FMOD Studio 中创建的素材在 Windows、Mac、iOS 和 Android 平台上实时渲染。

• Miles Sound System 是 Rad Game Tools (http://www.radgametools.com/miles.htm) 推出的一款热门音频中间件解决方案。它提供了强大的音频处理图，几乎可以在所有你能想到的游戏平台上使用。

• Wwise 是 Audiokinetic 推出的一款 3D 音频渲染引擎（https://www.audiokinetic.com）。值得注意的是，它不是基于多通道混音控制台的概念和功能，而是为声音设计师和程序员提供基于游戏对象和事件的独特界面。

• 虚幻引擎当然提供了自己的3D音频引擎和强大的集成工具链（http://www.unrealengine.com）。想要深入了解虚幻的音频功能集和工具，请参阅[45]。

14.6 游戏特定的音频功能

除了 3D 音频渲染管线之外，游戏通常会实现各种游戏特有的功能和系统。例如：

• 分屏支持。支持分屏播放的多人游戏必须提供某种机制，允许 3D 游戏世界中的多个听众共享客厅中的一组扬声器。

• 物理驱动音频。支持动态、物理模拟对象（如碎片、可破坏物体和布娃娃）的游戏需要一种播放适当音频的方式，以响应撞击、滑动、滚动和破碎。

• 动态音乐系统。许多故事驱动的游戏要求音乐能够实时适应游戏事件的氛围和紧张程度。

• 角色对话系统。人工智能角色在彼此对话以及与玩家角色对话时显得更加真实。

• 声音合成。一些引擎继续提供“从零开始”合成声音的功能，通过组合各种音量和频率的波形（正弦波、方波、锯齿波等）。先进的合成技术也逐渐应用于实时游戏。例如：

- *AeonWave 4D* is a low-cost audio library for Windows and Linux by Adalin B.V.
- *FMOD Studio* is an audio authoring tool that features a “pro audio” look and feel (http://www.fmod.org). A full-featured runtime 3D audio API allows assets created in FMOD Studio to be rendered in real time on the Windows, Mac, iOS and Android platforms.
- *Miles Sound System* is a popular audio middleware solution by Rad Game Tools (http://www.radgametools.com/miles.htm). It provides a powerful audio processing graph and is available on virtually every gaming platform imaginable.
- *Wwise* is a 3D audio rendering engine by Audiokinetic (https://www.audiokinetic.com). It is notably not based around the concepts and features of a multi-channel mixing console, but rather presents the sound designer and programmer with a unique interface based on game objects and events.
- The *Unreal Engine* of course provides its own 3D audio engine and powerful integrated tool chain (http://www.unrealengine.com). For an in-depth look at Unreal’s audio feature set and tools, see [45].

14.6 Game-Specific Audio Features

On top of the 3D audio rendering pipeline, games typically implement all sorts of game-specific features and systems. Some examples include:

- *Split-screen support*. Multiplayer games that support split-screen play must provide some mechanism that allows *multiple listeners* in the 3D game world to *share* a single set of speakers in the living room.
- *Physics-driven audio*. Games that support dynamic, physically simulated objects like debris, destructible objects and rag dolls require a means of playing appropriate audio in response to impacts, sliding, rolling and breaking.
- *Dynamic music system*. Many story-driven games require the music to adapt in real time to the mood and tension of events in the game.
- *Character dialog system*. AI-driven characters seem a great deal more realistic when they speak to one another and to the player’s character.
- *Sound synthesis*. Some engines continue to provide the ability to synthesize sounds “from scratch” by combining various kinds of waveforms (sinusoid, square, sawtooth, etc.) at various volumes and frequencies. Advanced synthesis techniques are also becoming practical for use in real-time games. For example:

◦ 乐器合成器无需使用预先录制的音频即可再现模拟乐器的自然声音。

◦ 基于物理的声音合成涵盖了一系列技术，旨在精确再现物体与虚拟环境进行物理交互时发出的声音。这类系统利用现代物理模拟引擎提供的接触、动量、力、扭矩和变形信息，并结合物体材质的属性及其几何形状，合成出适合撞击、滑动、滚动、弯曲等动作的声音。以下是一些关于这一引人入胜主题的研究链接：http://gamma.cs.unc.edu/research/sound、http://gamma.cs.unc.edu/AUDIO_MATERIAL、http://www.cs.cornell. edu/projects/sound 以及 https://ccrma.stanford.edu/□bilbao/booktop/node14.html。

◦ 车辆引擎合成器旨在根据虚拟引擎的加速度、转速、负载以及车辆的机械运动等输入，重现车辆发出的声音。（顽皮狗的三部《神秘海域》游戏中的车辆追逐场景都使用了各种形式的动态引擎建模，尽管从技术上讲，这些系统并非合成器，因为它们是通过在各种预先录制的声音之间进行交叉淡入淡出来产生输出的。）

◦ 发音语音合成器通过人类声道的3D模型“从零开始”生成人类语音。VocalTractLab (http://www.vocaltractlab.de) 是一款免费工具，可供学生学习和实验语音合成。

• 人群建模。如果游戏涉及人群（观众、城市居民等），则需要某种方法来渲染人群的声音。这并非简单地将大量人声叠加播放那么简单。相反，通常需要将人群建模为多层声音，包括背景氛围和个人发声。

我们不可能在一章内涵盖上述所有内容。不过，我们还是花几页篇幅来介绍一些最常见的游戏专属功能吧。

- *Musical instrument synthesizers* reproduce the natural sound of an analog musical instrument without the use of pre-recorded audio.
- *Physically based sound synthesis* encompasses a broad range of techniques that attempt to accurately reproduce the sound that would be made by an object as it physically interacts with a virtual environment. Such systems make use of the contact, momentum, force, torque and deformation information available from a modern physics simulation engine, in concert with the properties of the material from which the object is made and its geometric shape, in order to synthesize suitable sounds for impacts, sliding, rolling, bending and so on. Here are just a few links to research on this fascinating topic: http://gamma.cs.unc.edu/research/sound, http://gamma.cs.unc.edu/AUDIO_MATERIAL, http://www.cs.cornell.edu/projects/sound, and https://ccrma.stanford.edu/~bilbao/booktop/node14.html.
- *Vehicle engine synthesizers* aim to reproduce the sounds made by a vehicle, given inputs such as the acceleration, RPM and load placed on a virtual engine, and the mechanical movements of the vehicle. (The vehicle chase sequences in Naughty Dog's three *Uncharted* games all used various forms of dynamic engine modeling, although technically these systems were not synthesizers, because they produced their output by cross-fading between various prerecorded sounds.)
- *Articulatory speech synthesizers* produce human speech "from scratch" via a 3D model of the human vocal tract. VocalTractLab (http://www.vocaltractlab.de) is a free tool that allows students to learn about and experiment with vocal synthesis.

- *Crowd Modeling*. Games that feature crowds of people (audiences, city dwellers, etc.) require some means of rendering the sound of that crowd. This is not as simple as playing lots and lots of human voices over top of one another. Instead, it is usually necessary to model the crowd as multiple layers of sounds, including a background ambience plus individual vocalizations.

We can't possibly cover everything from the above list in one chapter. But let's spend a few more pages covering some of the most common game-specific features.

14.6.1 支持分屏

支持分屏多人游戏是一个棘手的问题，因为在虚拟游戏世界中你有多个听众，但他们必须共用玩家客厅中的一组扬声器。如果你只是多次平移声音，为每个听众平移一次，然后将结果均匀地混合到扬声器中，那么结果听起来并不总是合理的。没有完美的解决方案：例如，如果玩家 A 站在爆炸旁边，而玩家 B 站在远处，扮演玩家 B 的人仍然会听到响亮而清晰的爆炸声。游戏所能做的最好的就是拼凑一个混合解决方案，其中一些声音以“物理上正确”的方式处理，其他声音则进行“伪造”，以便为玩家提供最合理的听觉体验。

14.6.2 角色对话框

即使我们为游戏创建的角色看起来像真人照片，即使他们的动作极其逼真，但如果他们不能说话，玩家仍然不会觉得他们真实。语音传达着对游戏至关重要的信息。它是叙事的核心工具，并且巩固了人类玩家与游戏角色之间的情感纽带。语音也可能是玩家对游戏中AI控制角色智力感知的决定性因素。

在 2002 年的游戏开发者大会 (GDC) 上，Bungie 的 Chris Butcher 和 Jaime Griesemer 发表了题为“智能的幻觉：Halo 中 AI 和关卡设计的整合”的演讲 (http://bit.ly/1g7FbhD)。在演讲中，他们分享了一则轶事，讲述了语音在向玩家传达 AI 角色的动机方面有多么重要。在 Halo 中，当星盟小队的精英领袖被杀时，步兵都会惊恐地逃跑。在一次又一次的游戏测试中，似乎没有人明白正是精英的死亡导致了步兵的逃跑。最后，步兵听到的台词是“领袖死了——快跑！”直到那时，游戏测试员才开始真正明白发生了什么！

在本节中，我们将探讨几乎所有基于角色的游戏的角色对话系统中都会用到的一些基本子系统。我们还将讨论顽皮狗在《最后生还者》中用于创建丰富逼真对话的一些具体技巧和技术。如需了解更多信息以及顽皮狗角色对话系统的游戏内视频，请查看我在 2014 年游戏开发者大会 (GDC 2014) 上的演讲，题为“《最后生还者》中的情境感知角色对话”，PDF 和 QuickTime 格式可在以下网址获取：

14.6.1 Supporting Split-Screen

Supporting split-screen multiplayer is a tricky problem, because you have *multiple listeners* in the virtual game world, but they must *share* a single set of speakers in the player's living room. If you simply pan the sounds multiple times, once for each listener, and then mix the results into the speakers evenly, the result won't always sound sensible. There is no perfect solution: For example, if player A is standing right next to an explosion and player B is standing far away, the person playing player B will still hear the explosion loud and clear. The best a game can do is cobble together a hybrid solution, in which some sounds are handled in a "physically correct" way and others are "fudged" in order to produce the most sensible-sounding experience for the players.

14.6.2 Character Dialog

Even if we've created characters for our game that look like photographs of real human beings, and even if they move in astoundingly realistic ways, they still won't seem real to the player until they can *speak* realistically. Speech communicates information crucial to gameplay. It's a central story-telling tool. And it cements the emotional bond between the human player and the characters in the game. Speech can also be the deciding factor in the player's perception of *intelligence* among the AI-controlled characters in a game.

At the Game Developer's Conference (GDC) in 2002, Chris Butcher and Jaime Griesemer of Bungie gave a talk entitled, "The Illusion of Intelligence: The Integration of AI and Level Design in Halo" (http://bit.ly/1g7FbhD). In their talk, they shared an anecdote about how important speech can be to communicating the motivations of an AI-driven character to the player. In *Halo*, when the Elite leader of a squad of Covenant was killed, the grunts would all run away in fear. In playtest after playtest, no one seemed to understand that it was the killing of the Elite that had triggered the grunts to flee. Finally, the grunts were given lines of dialog saying something to the effect of, "Leader dead—run away!" Only then did play testers start to really grok what was going on!

In this section, we'll explore some of the fundamental subsystems you'll find in the character dialog system of pretty much any character-based game. We'll also discuss some of the specific techniques and technologies used by Naughty Dog to create rich, realistic conversations in *The Last of Us*. For more information and in-game videos of Naughty Dog's character dialog system in action, check out the talk I gave at GDC 2014 entitled, "Context-Aware Character Dialog in *The Last of Us*," available in PDF and QuickTime formats at

http://www.gameenginebook.com。

14.6.2.1 赋予角色声音

为游戏角色配音其实很简单——只要在角色需要说话时播放一段预先录制好的合适声音即可。然而，事情远非如此简单。游戏引擎中的对话系统通常相当复杂。以下是其中几个原因：

- 我们需要一种方法来记录每个角色可能说的所有台词，并赋予每句台词某种唯一的 ID，以便游戏可以在需要时触发它们。

- 我们需要确保游戏中每个独特可识别的角色都能拥有清晰且一致的声音。例如，《最后生还者》匹兹堡部分的每个猎人都有八种独特的配音，以确保战斗中每个猎人的声音都不会完全相同。

- 我们可能无法预先知道哪个角色会说出特定的台词，因此我们经常需要多次录制由不同配音演员说出的同一句台词，以便在需要时使用合适的声音说出这句台词。

- 我们通常也希望对话内容丰富多样。因此，大多数对话系统都提供了一种从众多可能对话中随机选择特定对话的方法。

- 语音音频资源通常持续时间较长，这意味着它们会占用大量内存。许多对话是过场动画序列的一部分，因此在整个游戏中只会说一次。因此，将语音资源存储在内存中通常是一种浪费。因此，语音音频资源通常会按需进行流式传输（参见第 14.5.7.4 节）。

通常，其他声音，例如举起重物、跳过障碍物或腹部受到重击时发出的“用力”声，都由处理对话的系统处理。这样做主要是因为角色的用力声需要与其说话的声音相匹配。因此，我们不妨利用对话系统来生成用力声。

14.6.2.2 定义对话行

大多数对话系统在发言请求和选择播放的特定音频片段之间引入了一层间接层。游戏程序员或设计师会请求逻辑对话，每条对话都由一个独特的

http://www.gameenginebook.com.

14.6.2.1 Giving a Character a Voice

It's easy enough to give a game character a voice—simply play an appropriate pre-recorded sound whenever the character needs to speak. However, things are never quite that simple. The dialog system in a game engine is typically a reasonably complex beast. Here are just a few of the reasons why:

- We need a way to catalog all the possible lines of dialog that each character might be called upon to say, and give each of these lines some kind of unique id so that the game can trigger them when needed.
- We need to make sure that each uniquely identifiable character in the game speaks with a recognizable and consistent voice. For example, each of the hunters in the Pittsburgh section of *The Last of Us* was assigned to one of eight unique voices, so that no two hunters in a battle would sound the same.
- We may not know a priori which character is going to be called upon to say a specific line, so we often need to record the same line multiple times, spoken by various voice actors, so that the appropriate voice can be used to say the line when needed.
- We also usually want a lot of variety in the things that are said. So most dialog systems provide a means of selecting specific lines at random from a pool of possibilities.
- Speech audio assets tend to be of relatively long duration, which means they occupy a lot of memory. Many lines of dialog are part of cinematic sequences, and hence are only spoken once in the entire game. For these reasons, it's usually wasteful to store speech assets in memory. Instead, it's typical for speech audio assets to be *streamed* on demand (see Section 14.5.7.4).

Usually other vocal sounds, like the "efforts" made when lifting something heavy, jumping over an obstacle or getting punched in the gut, are handled by the same system that handles spoken dialog. This is done largely because a character's efforts need to match his or her spoken voice. So we may as well leverage the dialog system to produce effort sounds as well.

14.6.2.2 Defining a Line of Dialog

Most dialog systems introduce a level of indirection between a *request* to speak and the choice of the particular *audio clip* to play. The game programmer or designer requests *logical* lines of dialog, each of which is represented by a unique

标识符，例如字符串，或者更好的是散列字符串id（参见第6.4.3.1节）。然后，声音设计师可以用一个或多个音频片段“填充”每个逻辑行，以便在语音质量和具体内容方面提供必要的多样性。

例如，假设角色在逻辑台词中说了类似“我的弹药用完了”这样的话。我们将为这句逻辑台词分配一个唯一 ID 'line-out-of-ammo ，其中前导单引号表示散列字符串 ID。我们还假设有十个不同的角色可能会说这句台词：玩家角色（称他为“drake”）、玩家的伙伴（称她为“elena”）以及最多八个敌方角色（称他们为“pirate-a”到“pirate-h”）。我们需要某种数据结构来定义构成这句逻辑台词的所有物理音频资产。

在顽皮狗，声音设计师使用 Scheme 编程语言，通过自定义语法定义逻辑对话。在下面的示例中，我们将使用类似的语法。不过，实现的细节在这里并不重要。我们感兴趣的只是数据本身的结构：

```
(define-dialog-line 'line-out-of-ammo
    (character 'drake
      (lines
        drk-out-of-ammo-01   ;; "Dammit, I'm out!"
        drk-out-of-ammo-02   ;; "Crap, need more bullets."
        drk-out-of-ammo-03   ;; "Oh, now I'm REALLY mad."
      )
    )
    (character 'elena
      (lines
        eln-out-of-ammo-01   ;; "Help, I'm out!"
        eln-out-of-ammo-02   ;; "Got any more bullets?"
      )
    )
    (character 'pirate-a
      (lines
        pira-out-of-ammo-01  ;; "I'm out!"
        pira-out-of-ammo-02  ;; "Need more ammo!"
        ;; ...
      )
    )
    ;; ...
    (character 'pirate-h
      (lines
        pirh-out-of-ammo-01  ;; "I'm out!"
        pirh-out-of-ammo-02  ;; "Need more ammo!"
```


identifier such as a string, or better yet a hashed string id (see Section 6.4.3.1). The sound designers can then “fill out” each logical line with one or more audio clips in order to provide the necessary variety both in voice quality and in terms of what exactly is said.

For example, let’s imagine a logical line in which the character says something to the effect of, “I’m out of ammo.” We’ll assign this logical dialog line the unique id `'line-out-of-ammo`, where the leading single quote indicates a hashed string id. Let’s assume also that there are ten different characters that might say this line: the player character (call him “drake”), the player’s sidekick (call her “elena”) and up to eight enemy characters (call them “pirate-a” through “pirate-h”). We’ll need some kind of data structure to define all the physical audio assets that make up this one logical line of dialog.

At Naughty Dog, sound designers use the Scheme programming language to define logical dialog lines using a custom syntax. We’ll use a similar syntax in our example below. However, the specifics of the implementation are not important here. All we’re interested in is the structure of the data itself:

```
(define-dialog-line 'line-out-of-ammo
    (character 'drake
      (lines
        drk-out-of-ammo-01   ;; "Dammit, I'm out!"
        drk-out-of-ammo-02   ;; "Crap, need more bullets."
        drk-out-of-ammo-03   ;; "Oh, now I'm REALLY mad."
      )
    )
    (character 'elena
      (lines
        eln-out-of-ammo-01   ;; "Help, I'm out!"
        eln-out-of-ammo-02   ;; "Got any more bullets?"
      )
    )
    (character 'pirate-a
      (lines
        pira-out-of-ammo-01  ;; "I'm out!"
        pira-out-of-ammo-02  ;; "Need more ammo!"
        ;; ...
      )
    )
    ;; ...
    (character 'pirate-h
      (lines
        pirh-out-of-ammo-01  ;; "I'm out!"
        pirh-out-of-ammo-02  ;; "Need more ammo!"
```

```
            ;; ...
        )
    )
)
```

与其像上图那样将对话内容定义在一个庞大的数据结构中，不如按角色将对话内容拆分到不同的文件中。例如，Drake 的所有对话可以放在一个文件中管理，Elena 的对话放在另一个文件中，而所有海盗的对话可以放在第三个文件中。这有助于避免音效设计师之间互相干扰。这也意味着我们可以更高效地管理内存。例如，如果游戏的某个部分没有海盗，就无需将海盗对话的数据保存在内存中。出于同样的原因，按关卡拆分对话数据也是一个好主意。

14.6.2.3 播放一行对话

有了这些数据，对话系统可以轻松地将逻辑台词（例如“line-out-of-ammo”）的请求转换为特定的音频片段。它只需在表中查找角色的特定语音 ID，然后在该角色的各种可能台词中随机选择即可。

通常，最好设置某种机制来确保台词不会重复出现。一种方法是将各台词的索引存储在一个数组中，然后随机打乱其内容。要选择一行，我们只需按顺序循环遍历打乱后的数组即可。所有可能的台词都用完后，我们会重新打乱数组，并注意不要让最近播放的台词出现在第一个位置。这样既能防止所有重复，又能保持台词选择的随机性。

对话台词请求通常由游戏代码（使用 C++、Java、C# 或任何游戏编写语言）发出。游戏设计师也可以通过脚本（Lua、Python 等）请求对话台词。对话系统的 API 通常以简单易用为设计理念。如果 AI 程序员或游戏设计师为了播放一句对话台词而费尽周折，你可能会发现角色的声响异常安静！最好提供一个简单的、“发射后不管”的界面。把所有繁琐的工作留给设计对话系统的程序员。

例如，在《神秘海域3：德雷克的诡计》中，C++代码可以通过调用Npc类的简单成员函数PlayDialog()来请求角色播放一段对话。这些调用会贯穿整个AI决策代码，以便在游戏的关键时刻触发合适的对话。例如：


```
            ;; ...
        )
    )
)
```

Rather than define your dialog lines in one monolithic data structure like the one shown above, it's usually better to break the lines out into separate files by character. For example, all of Drake's lines can be managed in one file, Elena's in another file and all the pirates' lines can be stored in a third file. This helps to prevent the sound designers from stepping on each others' toes. It also means that we can manage our memory more efficiently. For example, if there are no pirates in a given section of the game, there's no need to keep the data for the pirates' dialog lines in memory. It's also a good idea to split the dialog data up by level, for the same reason.

14.6.2.3 Playing a Line of Dialog

Given this data, the dialog system can easily convert a request for a logical line of dialog such as `'line-out-of-ammo` into a specific audio clip. It simply looks up the character's specific voice id in the table, and then makes a random choice amongst the various possible lines for that character.

It's usually a good idea to put in place some kind of mechanism to ensure that lines aren't repeated too often. One way to accomplish this is to store the indices of the various lines in an array and then randomly shuffle its contents. To select a line, we simply cycle through the shuffled array in order. Once all possible lines have been exhausted, we reshuffle the array, taking care that the most recently played line doesn't end up in the first slot. This prevents all repetition while keeping the line selections sounding random.

Dialog line requests are typically made by gameplay code in C++, Java, C# or whatever language your game is written in. Game designers may also request lines of dialog via script (Lua, Python, etc.). The dialog system's API is usually designed with simplicity of use in mind. If an AI programmer or game designer has to jump through a lot of hoops just to get a line of dialog to play, you may discover that your characters are uncannily silent! It's best to provide a simple, fire-and-forget interface. Leave all the hard work to the programmer who is crafting the dialog system.

For example, in *Uncharted 3: Drake's Deception*, C++ code could request a character to play a line of dialog by calling a simple `PlayDialog()` member function of the `Npc` class. These calls would be peppered throughout the AI decision-making code in order to trigger appropriate lines of dialog at key moments in the game. For example:

```
void Skill::OnEvent(const Event& evt)
{
    Npc* pNpc = GetSelf(); // grab a pointer to the NPC

    switch (evt.GetMessage())
    {
    case SID("player-seen"):
        // play a line of dialog...
        pNpc->PlayDialog(SID("line-player-seen"));
        // ... and move to closest cover
        pNpc->MoveTo(GetClosestCover());
        break;
    // ...
    }

    // ...
}
```

14.6.2.4 优先级和中断

如果一个角色已经在说话了，这时又被要求说话，会发生什么？如果他在同一帧中收到多个语音命令，又会怎么样？在这两种情况下，优先级系统都是解决歧义的好方法。

为了实现这样的系统，我们只需为每句台词分配一个优先级即可。当收到说台词的请求时，系统会查看当前正在播放的台词（如果有）的优先级，以及此帧已请求的台词的优先级。它会从中找到优先级最高的台词。如果当前正在播放的台词“获胜”，则继续播放，并忽略请求的台词。如果其中一条请求的台词优先级高于当前台词，或者角色尚未说话，则播放新的台词，并在必要时打断当前台词。

实现打断演讲本身其实是有点棘手的。我们不能进行交叉淡入淡出（即，将正在播放的声音的音量逐渐降低，将新的声音的音量逐渐升高），因为这应用于单个角色的对话时听起来很奇怪，也不对。理想情况下，我们希望在开始新台词之前至少播放某种声门塞音。甚至可以播放一个短语，表示角色对打断感到惊讶和/或恼火，然后播放新的台词。《最后生还者》中的对话系统并没有做任何这些花哨的事情。它只是停止当前台词并立即播放新台词。大多数时候这听起来都很不错。当然，每个游戏都有自己独特的语音模式，在一个游戏中有效的方法在另一个游戏中可能不起作用。所以俗话说，“你的里程可能会有所不同。”


```
void Skill::OnEvent(const Event& evt)
{
    Npc* pNpc = GetSelf(); // grab a pointer to the NPC

    switch (evt.GetMessage())
    {
    case SID("player-seen"):
        // play a line of dialog...
        pNpc->PlayDialog(SID("line-player-seen"));
        // ... and move to closest cover
        pNpc->MoveTo(GetClosestCover());
        break;
    // ...
    }

    // ...
}
```

14.6.2.4 Priority and Interruption

What happens if a character is asked to speak while he's already speaking? What if he receives more than one speech command on the same frame? In both cases, a *priority system* is a good way to resolve ambiguities.

To implement such a system, we simply assign each line of dialog to a priority level. When a request to say a line of dialog comes in, the system looks at the priority of the currently playing line if any, and the priorities of the line or lines that have been requested this frame. It finds the highest priority line among these. If the currently playing line "wins," it continues to play and the requested lines are ignored. If one of the requested lines is higher priority than the current line, or if the character isn't speaking yet, the new line plays, interrupting the current line if necessary.

Implementing the interruption of the speech itself is actually a bit tricky. We can't perform a cross-fade (i.e., fade the volume of the playing sound down and the new sound up) because this sounds strange and wrong when applied to the speech of a single character. Ideally, we'd want to play at least some kind of glottal stop sound just prior to starting the new line. It might even be appropriate to play a short phrase indicating that the character is surprised and/or annoyed by the interruption, and *then* play the new line of dialog. The dialog system in *The Last of Us* doesn't do any of these fancy things. It simply stops the current line and immediately plays the new one. This sounds pretty good most of the time. Of course, each game has its own unique speech patterns, and what works in one game may not work well in another. So as the saying goes, "Your mileage may vary."

14.6.2.5 对话

在《最后生还者》中，顽皮狗希望敌方 NPC 的声音听起来像是在彼此进行真正的对话。这意味着角色需要能够说出相对较长的台词，并且两个或多个角色之间需要来回对话。同样，在《神秘海域 4：盗贼末路》和《神秘海域：失落的遗产》中，我们希望角色在驾驶吉普车穿梭于马达加斯加和印度时能够进行对话。这些对话甚至可以被打断（例如，玩家决定下车探索该区域），并且当玩家返回车辆时，对话会从中断的地方继续。

《最后生还者》《神秘海域4》和《失落的遗产》中的对话由逻辑片段构成。每个片段对应一句逻辑台词，由对话中的特定角色说出。每个片段都被赋予一个唯一的ID，并通过这些ID将各个片段串联成一个对话。例如，让我们看看如何定义以下对话：

A：“嘿，你发现什么了吗？”

B：“没有，我已经找了一个小时了，但什么也没找到。”

答：“那你就闭嘴，继续看！”

这段对话可以用顽皮狗对话系统表达如下：

```
(define-conversation-segment 'conv-searching-for-stuff-01
    :rule []
    :line 'line-did-you-find-anything
                ;; "Hey, did you find anything?"
    :next-seg 'conv-searching-for-stuff-02
)
(define-conversation-segment 'conv-searching-for-stuff-02
    :rule []
    :line 'line-nope-not-yet
                ;; "I've been looking for an hour..."
    :next-seg 'conv-searching-for-stuff-03
)
(define-conversation-segment 'conv-searching-for-stuff-03
    :rule []
    :line 'line-shut-up-keep-looking
                ;; "Well then shut up and keep looking!"
)
```

这种语法乍一看可能有点冗长。但正如我们将在 14.6.2.8 节中看到的那样，像这样拆分对话可以给我们带来很多

14.6.2.5 Conversations

In *The Last of Us*, Naughty Dog wanted the enemy NPCs to sound like they're having real conversations with one another. This meant that the characters would need to be capable of saying relatively long chains of lines, with back-and-forth banter between two or more characters. Likewise, in *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy*, we wanted the characters to have conversations while driving around Madagascar and India in their jeep. These conversations could even be interrupted (for example by the player deciding to exit the jeep to explore the area), and would continue where they left off when the player returned to the vehicle.

Conversations in *The Last of Us*, *Uncharted 4* and *The Lost Legacy* are constructed from logical *segments*. Each segment corresponds to one logical line, spoken by one particular actor in the conversation. Each segment is given a unique id, and the segments are *chained* together into a conversation via these ids. As an example, let's see how we would define the following conversation:

A: "Hey, did you find anything?"

B: "No, I've been looking for an hour and I ain't found nothin'."

A: "Well then shut up and keep looking!"

This conversation could be expressed in the Naughty Dog conversation system as follows:

```
(define-conversation-segment 'conv-searching-for-stuff-01
    :rule []
    :line 'line-did-you-find-anything
                ;; "Hey, did you find anything?"
    :next-seg 'conv-searching-for-stuff-02
)
(define-conversation-segment 'conv-searching-for-stuff-02
    :rule []
    :line 'line-nope-not-yet
                ;; "I've been looking for an hour..."
    :next-seg 'conv-searching-for-stuff-03
)
(define-conversation-segment 'conv-searching-for-stuff-03
    :rule []
    :line 'line-shut-up-keep-looking
                ;; "Well then shut up and keep looking!"
)
```

This syntax might seem a bit verbose at first glace. But as we'll see in Section 14.6.2.8, breaking the conversation out like this gives us a great deal of

灵活性。例如，它允许以自然且合理方便的方式定义分支对话。

14.6.2.6 打断谈话

我们在第 14.6.2.4 节中看到，可以使用简单的优先级系统来处理中断并解决同时请求多个逻辑线路时的争用。

当对话正在进行时，优先级系统仍然可以使用。但在这种情况下，其实现会稍微复杂一些。例如，假设角色 A 和 B 之间有一段对话。A 说了自己的台词，然后 B 说了她的台词，同时 A 等待轮到自己。在 B 说话的时候，A 被要求播放一段完全不同的台词。从技术上讲，他不是说话者，所以根据对话优先级规则（分别应用于每个角色），这不会有问题，台词会播放。但这听起来可能会很刺耳，具体取决于对话的内容。

A: “嘿，你发现什么了吗？”

B: “没有，我已经找了一个小时了……”

A:“看，一个闪亮的物体！”

（被不相关的对话打断）B:“……我什么也没找到。”

为了在《最后生还者》中解决这个问题，我们引入了对话作为“一等实体”的概念。当对话开始时，系统“知道”每个角色都参与了该对话，即使他或她没有说话。每个对话都有一个优先级，优先级规则适用于整个对话，而不是每个角色的单行对话。例如，当角色 A 被要求说“看，一个闪亮的物体！”时，系统知道他当前正在参与“嘿，你找到什么了吗？”对话。可以推测，“看，一个闪亮的物体！”这句话的优先级与当前对话相同或更低，因此不允许打断。

如果打断的台词优先级更高，比如“我的天，他拿枪指着我们！”，那么该台词就可以打断现有的对话。在这种情况下，对话中的所有角色都会被打断。

结果是，打断听起来很自然，也很明智。

A:“嘿，你发现什么了吗？”

B:“没有，我已经找了一个小时了……”

flexibility. For example, it allows branching conversations to be defined in a natural and reasonably convenient way.

14.6.2.6 Interrupting Conversations

We saw in Section 14.6.2.4 that a simple priority system can be used to handle interruptions and to resolve contention when more than one logical line is requested simultaneously.

When *conversations* are in play, a priority system can still be used. But its implementation is a bit more complex in this case. For example, imagine a conversation between characters A and B. A says his line, then B says her line while A waits his turn. During the time that B is speaking, A is asked to play an entirely different line of dialog. He's not tehnically speaking, so by the rules for dialog prioritization, applied to each character individually, there would be no problem and the line would play. But this could sound very jarring, depending on what's being said.

A: "Hey, did you find anything?"

B: "No, I've been looking for an hour and…"

A: "Look, a shiny object!"

(interruption by an unrelated line of dialog)

B: "…I ain't found nothin'."

To overcome this issue on *The Last of Us*, we introduced the concept of conversations as "first-class entities." When a conversation is started, the system "knows" that each of the characters is involved in that conversation, even when he or she isn't speaking. Each conversation has a priority, and the prioritization rules are applied to entire conversations, not the individual lines on a per-character basis. So for example, when charater A is asked to say, "Look, a shiny object!" the system knows that he is currently involved in the "Hey, did you find anything?" conversation. Persumably the line "Look, a shiny object!" is at the same or lower priority as the current conversation, so the interruption isn't allowed.

If the interrupting line is something higher priority like, "Holy cow, he's pointing a gun at us!" then the line *is* allowed to interrupt the existing conversation. In that case, all of the characters in the conversation are interrupted. The result is an interruption that sounds natural and intelligent.

A: "Hey, did you find anything?"

B: "No, I've been looking for an hour and…"

A: “我的天，他拿枪指着我们！”（被更高优先级的对话打断）

B：“抓住他！”
（原来的对话被新的对话打断，A 和 B 进入战斗模式。）

14.6.2.7 排他性

在《最后生还者》中，我们还引入了“独占”的概念。任何台词或对话都可以标记为“非独占”、“派系独占”或“全局独占”。此标记控制着特定台词或对话的打断方式。

- 非专属台词或对话可以叠加在其他台词或对话之上播放。例如，在搜寻玩家的过程中，如果一个猎人自言自语说“啊，这边什么都没有”，而另一个猎人说“我受够了”，这没什么问题。这两个猎人之间并没有对话，所以重叠部分听起来非常自然。

- 派系专属台词或对话会打断该角色派系内所有其他台词或对话。例如，如果玩家（乔尔）在搜索过程中被发现，看到他的猎人可能会说：“他在这里！”其他猎人应该立即停止说话，因为我们希望猎人之间能够互相听到，同时也让玩家知道他们的注意力已经转移。但是，如果乔尔的同伴艾莉当时正在低声警告他，我们可能不想打断她。她不属于猎人团伙，无论猎人是否发现了乔尔，她对乔尔说的话都至关重要。

- 全局专属的线路或对话会打断所有其他线路，跨越派系界限。这在需要所有在听力范围内的角色都对正在说的话做出反应的情况下非常有用。

14.6.2.8 选择和分支对话

人们通常希望对话能够根据玩家的行为、AI角色的决策以及/或者游戏世界状态的其他方面以不同的方式展开。在创作或编辑对话时，编剧和音效设计师不仅希望能够控制对话内容，还希望能够控制在特定时刻触发对话分支的逻辑条件。

A: “Holy cow, he’s pointing a gun at us!”
(interruption by a higher-priority conversation)

B: “Get him!”
(The original conversation is interrupted by the new one, and A and B go into combat mode.)

14.6.2.7 Exclusivity

On *The Last of Us*, we also introduced the concept of *exclusivity*. Any line of dialog or conversation can be marked as either *non-exclusive*, *faction-exclusive* or *globally exclusive*. This mark-up controls how interruptions work for the given line or conversation.

- A *non-exclusive* line or conversation is permitted to play *over top* of other lines or conversations. For example, during a search for the player, it’s not a big problem if one hunter is mumbling to himself, “Huh, there’s nothing over here.” while another hunter is saying, “I’m getting tired of this.” The two hunters aren’t speaking to each other, so the overlap sounds perfectly natural.
- A *faction-exclusive* line or conversation interrupts all other lines or conversations within that character’s faction. For example, if the player (Joel) is spotted during a search, the hunter that saw him might say, “He’s over here!” The other hunters should immediately stop speaking, because we want to make it seem as if the hunters can hear one another, and also to communicate to the player that their collective focus has shifted. However, if Joel’s companion Ellie is whispering a warning to him at the time, we probably do not want to interrupt her. She is not part of the hunter gang, and what she has to say to Joel is relevant whether or not the hunters have spotted him.
- A *globally exclusive* line or conversation interrupts all other lines, across faction boundaries. This is useful in situations in which *every* character within earshot should react to hearing whatever is being said.

14.6.2.8 Choices and Branching Conversations

It’s often desirable to allow conversations to play out in different ways depending on what the player does, on what decisions the AI characters make and/or on other aspects of game world state. When authoring or editing conversations, the writers and sound designers would like to have control not only over which lines are said, but also over the logical conditions that control which branch of the conversation will be taken at any given moment during

游戏玩法。这将创造力交到需要它的人手中，而不是强迫他们通过程序员来工作。

顽皮狗在《最后生还者》中实现了这样一个系统。该系统部分灵感来源于 Valve 早期开发的系统，Elan Ruskin 在 2012 年游戏开发者大会上的演讲“基于规则数据库的情境对话和游戏逻辑”中对此进行了描述。演讲内容可在此处观看：http://www.gdcvault.com/play/1015317/AI-driven

动态对话贯穿。顽皮狗的对话系统与 Valve 的对话系统在许多重要方面有所不同，但两者的核心理念却相似。我们在此仅介绍顽皮狗的系统，因为作者对该系统最为熟悉。

在顽皮狗的对话系统中，每个对话片段可以包含一个或多个备选台词。片段中的每个备选台词都带有一个选择规则。如果规则评估为真，则选择该备选台词；如果规则评估为假，则忽略该备选台词。

一条规则由一个或多个条件组成。每个条件都是一个简单的逻辑表达式，其计算结果为布尔值。表达式 ('health > 5) 和 ('player-death-count == 1) 就是条件的示例。如果一条规则中包含多个条件，则使用布尔运算符 AND 将它们进行逻辑组合。只有当所有条件都满足条件时，该规则的计算结果才为真。

以下是一段对话的示例，其中有三种选择，取决于说话角色的健康状况：

```
(define-conversation-segment 'conv-player-hit-by-bullet
  (
    :rule [ ('health < 25) ]
    :line 'line-i-need-a-doctor
              ;; "I'm bleeding bad... need a doctor!"
  )
  (
    :rule [ ('health < 75) ]
    :line 'line-im-in-trouble
              ;; "Now I'm in real trouble."
  )
  (
    :rule [ ] ;; no criteria acts as an "else" case
    :line 'line-that-was-close
              ;; "Ah! Can't let that happen again!"
  )
)
```


gameplay. This puts the creative power in the hands of the people who need it, instead of forcing them to work through a programmer.

Naughty Dog implemented such a system for use on *The Last of Us*. It was inspired in part by an earlier system developed by Valve and described by Elan Ruskin in his talk, “Rule Databased for Contextual Dialog and Game Logic,” which he delivered at the Game Developer’s Conference in 2012. The talk is available here: http://www.gdcvault.com/play/1015317/AI-driven-Dynamic-Dialog-through. Naughty Dog’s conversation system differs from Valve’s in a number of significant ways, but the core idea behind both systems is similar. We’ll describe the Naughty Dog system here, since that’s the system with which the author has the most experience.

In Naughty Dog’s conversation system, each segment of a conversation can consist of one or more alternative lines of dialog. Each alternative within the segment carries with it a *selection rule*. If the rule evaluates to true, that alternative is selected; if the rule evaluates to false, the alternative is ignored.

A rule is comprised of one or more criteria. Each criterion is a simple logical expression that evaluates to a Boolean. The expressions `('health > 5)` and `('player-death-count == 1)` are examples of criteria. If more than one criterion is provided within a rule, they are logically combined using the Boolean AND operator. A rule only evaluates to true when *all* of its criteria evaluate to true.

Here’s an example of one segment of a conversation, with three alternatives that depend upon the health of the character doing the talking:

```
(define-conversation-segment 'conv-player-hit-by-bullet
  (
    :rule [ ('health < 25) ]
    :line 'line-i-need-a-doctor
              ;; "I'm bleeding bad... need a doctor!"
  )
  (
    :rule [ ('health < 75) ]
    :line 'line-im-in-trouble
              ;; "Now I'm in real trouble."
  )
  (
    :rule [ ] ;; no criteria acts as an "else" case
    :line 'line-that-was-close
              ;; "Ah! Can't let that happen again!"
  )
)
```

分支对话框

通过将对话拆分成多个片段，每个片段包含一条或多条备选台词，我们便可以设计出分支对话。例如，假设艾莉（《最后生还者》中玩家的同伴）在乔尔（玩家角色）中枪后询问他是否安好。如果玩家没有被子弹击中，对话内容如下：

艾莉：“你还好吗？”乔尔：“是的，我很好。”
艾莉：“哎呀。低下头！”

如果乔尔被击中，对话就会有所不同：

艾莉：“你还好吗？”
乔尔：“（喘气）不完全是。”
艾莉：“你在流血！”

我们可以使用上面描述的对话语法来表达这个分支对话：

```
(define-conversation-segment 'conv-shot-at--start
  (
    :rule [ ]
    :line 'line-are-you-ok ;; "Are you OK?"
    :next-seg 'conv-shot-at--health-check
    :next-speaker 'listener ;; *** see comments below
  )
)

(define-conversation-segment 'conv-shot-at--health-check
  (
    :rule [ (('speaker 'shot-recently) == false) ]
    :line 'line-yeah-im-fine ;; "Yeah, I'm fine."
    :next-seg 'conv-shot-at--not-hit
    :next-speaker 'listener ;; *** see comments below
  )
  (
    :rule [ (('speaker 'shot-recently) == true) ]
    :line 'line-not-exactly ;; "(panting) Not exactly."
    :next-seg 'conv-shot-at--hit
    :next-speaker 'listener ;; *** see comments below
  )
)
```


Branching Dialog

By breaking a conversation into segments, each of which contains one or more alternative lines, we open up the possibility of crafting branching conversations. For example, let's consider a conversation in which Ellie (the player's companion in *The Last of Us*) asks Joel (the player character) if he's all right when he's been shot at. If the player wasn't actually hit by the bullet, the conversation goes like this:

Ellie: "Are you OK?"
Joel: "Yeah, I'm fine."
Ellie: "Geez. Keep your head down!"

If Joel has been hit, the conversation plays out differently:

Ellie: "Are you OK?"
Joel: "(panting) Not exactly."
Ellie: "You're bleeding!"

We can express this branching conversation using the conversation syntax described above:

```
(define-conversation-segment 'conv-shot-at--start
  (
    :rule [ ]
    :line 'line-are-you-ok ;; "Are you OK?"
    :next-seg 'conv-shot-at--health-check
    :next-speaker 'listener ;; *** see comments below
  )
)

(define-conversation-segment 'conv-shot-at--health-check
  (
    :rule [ (('speaker 'shot-recently) == false) ]
    :line 'line-yeah-im-fine ;; "Yeah, I'm fine."
    :next-seg 'conv-shot-at--not-hit
    :next-speaker 'listener ;; *** see comments below
  )
  (
    :rule [ (('speaker 'shot-recently) == true) ]
    :line 'line-not-exactly ;; "(panting) Not exactly."
    :next-seg 'conv-shot-at--hit
    :next-speaker 'listener ;; *** see comments below
  )
)
```

```
(define-conversation-segment 'conv-shot-at--not-hit
  (
    :rule [ ]
    :line 'line-keep-head-down ;; "Geez. Keep your head down!"
  )
)

(define-conversation-segment 'conv-shot-at--hit
  (
    :rule [ ]
    :line 'line-youre-bleeding ;; "You're bleeding!"
  )
)
```

说话者和倾听者

上面的分支对话中发生的事情有一个微妙的方面。在两人对话中的任何特定时刻，一个人是说话者，另一个人是听众。随着对话的进行，说话者和听众的角色来回变换。在对话的第一部分 'conv-shot-at--start 中，Ellie 是说话者，Joel 是听众。当我们链接到下一部分 'conv-shot-at--health-check 时，我们为字段 :next-speaker 指定值 'listener。这告诉系统使用当前听众（Joel）作为下一个部分的说话者，从而互换角色。在该部分中，我们通过条件 (('speaker 'shot-recently) == false) 和 (('speaker 'shot-recently) == true) 检查说话者最近是否被枪击过。但现在 Joel 是说话者，所以一切都按我们预期的方式进行。

对于像乔尔和艾莉这样的两个主要角色之间的对话，抽象的说话者/听众系统似乎没什么用。但是，通过保持对话定义的抽象性，我们获得了很大的灵活性。首先，我们可以使用相同的对话规范来定义乔尔询问艾莉是否还好的对话。这样做是可行的，因为整个对话的定义方式与每句台词由哪个角色说无关。此外，对于敌方角色，绝对有必要以通用的方式定义对话，因为我们无法预先知道哪些特定角色会说话。对于敌方战斗聊天，我们通常会动态选择一对角色并开始对话。无论选择哪两个角色，它都必须有效。

说话者/听众系统可以扩展到两人或三人对话。顽皮狗对话系统最多支持三人，尽管我们绝大多数的对话都是在两人之间进行的。


```
(define-conversation-segment 'conv-shot-at--not-hit
  (
    :rule [ ]
    :line 'line-keep-head-down ;; "Geez. Keep your head down!"
  )
)

(define-conversation-segment 'conv-shot-at--hit
  (
    :rule [ ]
    :line 'line-youre-bleeding ;; "You're bleeding!"
  )
)
```

Speaker and Listener

There's a subtle aspect to what's going on in the branching converstion above. At any given moment in a two-person conversation, one person is the speaker and the other is the listener. The roles of speaker and listener ping-pong back and forth as the conversation progresses. In the first segment of the conversation, `'conv-shot-at--start`, Ellie is the speaker and Joel is the listener. When we chain to the next segment, `'conv-shot-at--health-check`, we specify the value `'listener` for the field `:next-speaker`. This tells the system to use the current listener (Joel) as the next segment's speaker, thereby reversing the roles. In that segment, we check whether the speaker has been shot recently via the criteria `(('speaker 'shot-recently) == false)` and `(('speaker 'shot-recently) == true)`. But now Joel is the speaker, so everything works out as we'd expect.

An abstract speaker/listener system doesn't seem all that useful for a conversation between two principal characters like Joel and Ellie. But by keeping the definition of the conversation abstract, we gain a significant amount of flexibility. For one thing, we could use the same conversation specification to define a conversation in which Joel asks Ellie if she's OK. This works because the entire conversation is defined in a way that is independent of which character is saying each line. Moreover, for enemy characters it's absolutely essential that conversations be defined in a generic manner, because we don't know which specific characters will be doing the speaking a priori. For enemy battle chatter, we typically select a pair of characters dynamically and fire off the conversation. It has to work, no matter which two characters are selected.

The speaker/listener system can be extended to two- or three-person conversations. The Naughty Dog conversation system supported up to three listeners, although the vast majority of our conversations were between only

两个字符。

事实词典

规则中的标准引用符号量，例如“生命值”和“玩家死亡数”。这些符号量在底层实现为字典数据结构（本质上是一个包含键值对的表）中的条目。我们称之为事实字典。表 14.1 显示了事实字典的一个示例。

您可能已经在表 14.1 中注意到，字典中的每个值都有一个关联的数据类型。换句话说，字典中的值是变体。变体是一种数据对象，它能够保存各种类型的值，很像 C 或 C++ 中的联合。但是，与联合不同的是，变体还存储有关其当前包含的数据类型的信息。这使我们能够在使用值之前验证其类型。它还允许我们将数据从一种类型转换为另一种类型。例如，如果我们的变体保存整数值 42，我们可以要求变体将其作为浮点值 42.0f 返回给我们。

在《最后生还者》中，每个角色都有自己的事实字典，其中包含关于角色本身的事实，例如生命值、武器类型、意识等级等等。每个“阵营”角色也都有一本事实字典。这使我们能够表达关于整个阵营的事实，例如该阵营中还有多少角色存活。最后，还有一个单独的“全局”事实字典，其中包含关于整个游戏的信息，与阵营无关。诸如游戏时间、当前关卡名称或玩家重试特定任务的次数等信息都可以放入全局事实字典中。

标准语法

在编写条件时，语法允许按名称从任何字典中提取事实。例如，(('self'health) > 5) 告诉系统

钥匙	价值	数据类型
'name	'ellie	StringId
'faction	'buddy	StringId
'health	82	int
'is-joels-friend	真的	bool
…	…	…

表 14.1. 事实字典的示例。

two characters.

Fact Dictionaries

The criteria within a rule reference symbolic quantities like `'health` and `'player-death-count`. These symbolic quantities are implemented under the hood as entries in a *dictionary* data structure—basically a table containing key-value pairs. We call these *fact dictionaries*. An example of a fact dictionary is shown in Table 14.1.

You may have noticed in Table 14.1 that each value in the dictionary has an associated data type. In other words, the values in the dictionary are *variants*. A variant is a data object that is capable of holding values of various types, much like a `union` in C or C++. However, unlike a `union`, a variant also stores information about the type of data it currently contains. This allows us to validate the type of a value prior to using it. It also lets us convert data from one type to another. For example, if our variant holds the integer value `42`, we could ask the variant to return it to us as the floating-point value `42.0f` instead.

In *The Last of Us*, each character has its own fact dictionary containing facts about the character itself like health, weapon type, awareness level and so on. Each “faction” of characters also has a fact dictionary. This allows us to express facts about the faction as a whole, like how many characters remain alive within the group. Finally, there is a singleton “global” fact dictionary that contains information about the game as a whole, without respect to faction. Things like the amount of time spent playing, the name of the current level or how many times the player has retried a particular task are all things that can go into the global fact dictionary.

Criterion Syntax

When writing a criterion, the syntax allows for facts to be pulled from any dictionary by name. For example, `(('self 'health) > 5)` tells the sys-

Key	Value	Data Type
'name	'ellie	StringId
'faction	'buddy	StringId
'health	82	int
'is-joels-friend	true	bool
…	…	…

Table 14.1. An example of a fact dictionary.

tem 抓取角色本身的事实字典，在该字典中查找“健康”事实的值，然后检查它是否大于 5。同样，(('global'seconds-playing) <= 23.5) 指示系统从全局事实字典中查找“seconds-playing”事实，并检查它是否小于或等于 23.5 秒。

如果用户未明确指定字典，例如 ('health > 5)，系统将按照预定义的搜索顺序搜索指定的事实。首先检查角色的事实字典。如果失败，则尝试在与角色派系匹配的字典中查找。最后，如果其他方法均失败，则在全局字典中查找该事实。此“搜索路径”功能使声音设计师在编写标准时尽可能简洁（尽管规则会失去一些特异性和清晰度）。

14.6.2.9 上下文相关对话框

在《最后生还者》中，我们希望敌方角色能够以一种智能的方式喊出玩家的位置。如果玩家躲在商店里，敌人应该喊：“他在商店里！”如果玩家躲在车后，我们希望反派角色说：“他在那辆车后面！”这让角色听起来非常聪明，但实现起来却相对简单。

为了实现这一点，音效设计师用区域标记了我们的游戏世界。每个区域都标有两种位置标签。一种是特定标签，它会将区域标记在一个非常具体的位置，例如“柜台后面”或“收银台旁边”。另一种是通用标签，它会将区域标记在一个更通用的位置，例如“商店里”或“街上”。

为了确定播放哪句台词，系统会确定玩家所在的区域以及敌方 NPC 所在的区域。如果它们位于同一大致区域，则使用玩家的特定标签来选择台词。当 NPC 和玩家位于不同的大致区域时，我们会转而使用玩家的大致区域标签来选择台词。因此，如果敌人和玩家都在商店里，我们可能会选择这样的台词：“他在窗边！”但如果 NPC 在商店里而玩家在街上，我们可能会听到 NPC 说：“他在街上！抓住他！”有关此系统如何工作的说明，请参见图 14.42。

这个非常简单的系统却被证明非常强大。由于需要记录和配置的线路组合数量庞大，设置起来非常困难，但最终的游戏效果证明，所有的努力都是值得的。

14.6.2.10 对话框操作

不带肢体语言的台词通常看起来怪异而不真实。有些台词是作为全身动画的一部分呈现的——例如游戏内的过场动画。但有些台词必须在

tem to grab the fact dictionary of the character itself, look up the value of the `'health` fact in that dictionary and then check if it is greater than 5. Likewise, `(('global 'seconds-playing) <= 23.5)` instructs the system to look up the `'seconds-playing` fact from the global fact dictionary, and check that it is less than or equal to 23.5 seconds.

If the user doesn't specify a dictionary explicity, as in `('health > 5)`, the system searches for the named fact by following a predefined search order. Check the character's fact dictionary first. If that fails, try to find it in the dictionary that matches the character's faction. Finally, if all else fails, look for the fact in the global dictionary. This "search path" feature allows sound designers to be as brief as possible when writing criteria (albeit with the loss of some specificity and clarity in the rules).

14.6.2.9 Context-Sensitive Dialog

In *The Last of Us*, we wanted to have enemy characters call out the location of the player in an intelligent way. If the player is hiding in a store, the enemies should shout, "He's in the store!" If he's hiding behind a car, we want the bad guys to say, "He's behind that car!" This makes the characters sound incredibly intelligent, yet it turns out to be a relatively simple thing to implement.

To make this work, the sound designers mark up our game worlds with regions. Each region is tagged with one of two kinds of *location tags*. A *specific* tag marks the region with a very specifc location like "behind the counter" or "by the cash register." A *general* tag marks the region with a more general location like "in the store" or "in the street."

To determine which line of dialog to play, the system determines within which region the player is located, and within which region the enemy NPC is located. If they are both in the same *general* region, the player's *specific* tag is used to select dialog lines. When the NPC and player reside in *different* general regions, we fall back to using the player's general region tag to select the line. So if the enemy and the player are both in the store, we might select a line like, "He's by the window!" But if the NPC is in the store and the player is out in the street, we might hear the NPC say, "He's out in the street! Get him!" See Figure 14.42 for an illustration of how this system works.

This very simple system proved incredibly powerful. It was difficult to set up due to the sheer number of combinations of lines that had to be recorded and configured, but the final result in-game was worth the effort.

14.6.2.10 Dialog Actions

Lines of dialog delivered without body language usually look uncanny and unrealistic. Some dialog lines are delivered as part of a full-body animation—an in-game cinematic for example. But some lines must be delivered while the

图 14.42. 上下文相关对话框行选择的一般区域和特定区域。

角色正忙着做其他事情，比如走路、跑步或开枪。理想情况下，我们希望用一些手势来为这些对话增添趣味，让它们更加生动。

在《最后生还者》中，我们使用了附加动画技术（参见第 12.6.5 节）实现了一个手势系统。这些手势可以通过 C++ 代码或脚本显式调用。此外，每句台词都可以关联一个脚本，其时间线与音频同步。这使我们能够在关键台词的精确时刻触发手势。

14.6.3 音乐

音乐几乎是任何优秀游戏的重要组成部分。它奠定了游戏基调，激发玩家的紧张感，并能成就（或破坏）一个充满情感的场景。游戏引擎的音乐系统通常承担以下任务：

- 提供将音乐曲目作为流音频剪辑播放的功能（因为音乐剪辑几乎总是太大而无法放入内存）。
- 提供音乐多样性。
- 将音乐与游戏中发生的事件相匹配。
- 从一段音乐无缝过渡到下一段音乐。

Figure 14.42. General and specific regions for context-sensitive dialog line selection.

character is busy doing something else, like walking, running or firing their weapon. Ideally we'd like to spice up such lines of dialog with some gestures to breathe life into them.

On *The Last of Us*, we implemented a gesture system using additive animation technology (see Section 12.6.5). These gestures could be explicitly called out by C++ code or script. In addition, each line of dialog could have a script associated with it whose timeline was synchronized with the audio. This allowed us to trigger gestures at precise moments during key lines of dialog.

14.6.3 Music

Music is an incredibly important aspect of pretty much any good game. It sets the tone, drives the player's sense of tension, and can make (or break) an emotional scene. A game engine's music system is typically charged with the following duties:

- Provide the ability to play back music tracks as streaming audio clips (because music clips are almost always too large to fit in memory).
- Provide musical variety.
- Match the music to the events occurring in the game.
- Seamlessly transition from one piece of music to the next.

- 以合适且令人愉悦的方式将音频与游戏中的其他声音混合。
- 允许暂时降低音乐音量，以增强游戏中特定声音或对话的可听性。
- 允许简短的音乐或声音效果（称为 stinger）暂时打断当前播放的音乐曲目。
- 允许暂停和重新播放音乐。（你不需要整支管弦乐队在游戏的每一秒都演奏宏大的主题曲，你懂的！）

我们通常希望音乐能够随着游戏中发生的事件的紧张程度和/或情绪变化而变化。实现这一点的一种方法是创建多个播放列表，每个播放列表用于游戏中不同程度的紧张或情绪。每个播放列表包含一首或多首音乐，可以随机或连续选择。随着游戏中的紧张程度和情绪的变化（战斗开始和结束，感人的过场动画出现和消失等等），音乐系统会检测到这些变化并根据需要选择新的音乐播放列表。有些游戏会实现"堆栈"音乐选择以增加紧张程度 - 当周围没有敌人时播放平静的音乐，当玩家接近一群毫无防备的敌人时播放紧张的音乐，第一次接触时播放令人吃惊的音乐，战斗期间播放快节奏的音乐。

插曲是另一种将音乐与游戏事件相匹配的方式。插曲是一种简短的音乐片段或音效，可以暂时打断当前正在播放的音乐曲目，或在主音轨音量降低时在其上方播放。例如，当玩家第一次与新敌人出现视线时，我们可能想播放一段不祥的"隆隆"声，提示玩家危险即将来临。或者，当玩家死亡时，我们可能想快速切换到一段"死亡音乐"。这两种情况都可以使用插曲。

在不同的音乐流之间实现流畅的过渡颇具挑战性。我们不能盲目地将毫不相关的音乐片段交叉混合，并期望效果总是完美。两首乐曲的节奏可能不一致，一段乐曲的"节拍"也可能与下一段不一致。关键在于把握好每次过渡的时机。如果节奏不一致，快速的交叉淡入淡出效果会很有用；如果节奏几乎相同，较长的交叉淡入淡出效果可能会更好。这需要反复试验才能做到。即使是让一段乐曲循环播放，也需要音响工程师进行一些调整。

游戏音乐这个话题很广泛，我们在这里无法一一详述。如果你有兴趣了解更多，[45] 是一本不错的入门书。

- Mix the audio with the other sounds in the game in a suitable and pleasing manner.
- Allow music to be temporarily ducked to enhance the audibility of specific sounds or conversations in-game.
- Permit brief pieces of music or sound effects known as *stingers* to temporarily interrupt the currently playing music track.
- Allow music to be paused and restarted. (You don't need a full orchestra playing a grandiose theme during every single second of gameplay, you know!)

We generally expect the music to change to match the changing levels of tension and/or emotional moods of the events happening in the game. One way to accomplish this is to create multiple *playlists*, each of which is intended for a different level of tension or emotional mood in the game. Each playlist contains one or more pieces of music, from which selections may be made randomly or sequentially. As the tension and mood change in the game—as battles begin and end, touching cutscenes come and go and so on—the music system detects these changes and selects new music playlists as appropriate. Some games implement a "stack" of music selections at increasing tension levels—calm music for when no enemies are around, tense music when the player is approaching an unsuspecting group of enemies, startling music at first contact and fast-paced music during battle.

Stingers are another way to match the music to the events in the game. A stinger is a short musical clip or sound effect that can temporarily interrupt the currently playing music track, or play over top of it while the main track's volume is ducked down. For example, the first time the player makes line-of-sight with a new enemy, we might want to play an ominous "rumbling" sound to give the player a cue that danger is near. Or when the player dies, we may want to quickly switch to a snippet of "death music." Both of these are situations in which a stinger might be used.

Transitioning smoothly between different music streams is somewhat of a challenge. We cannot blindly cross-blend between totally unrelated pieces of music and expect it to always sound good. The tempos of the two pieces may not match, and the "beat" of one piece of music might not line up with that of the next. The key is to time each transition properly. A rapid cross-fade can be useful if the tempos don't match; a longer cross-fade might work well if the tempos are nearly identical. This takes some trial and error to get right. Even getting a piece of music to loop properly requires some tweaking by the sound engineer.

The topic of game music is a broad one, and we can't really do it justice here. If you are interested in learning more, [45] is a great book to start with.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

第四部分
游戏玩法

Part IV
Gameplay

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

15
游戏系统简介

到目前为止，本书讨论的所有内容都集中在技术上。我们已经了解到，游戏引擎是一个复杂的、分层的软件系统，构建在目标机器的硬件、驱动程序和操作系统之上。我们已经了解了低级引擎系统如何提供引擎其他部分所需的服务；人机界面设备（例如游戏手柄、键盘、鼠标和其他设备）如何允许人类玩家向引擎提供输入；渲染引擎如何在屏幕上生成 3D 图像；动画系统如何使角色和物体自然移动；碰撞系统如何检测和解决形状之间的相互穿透；物理模拟如何使物体以逼真的物理方式移动；3D 音频引擎如何为我们的游戏世界渲染出逼真且身临其境的音景。但是，尽管这些组件提供了各种强大的功能，但如果将它们全部组合在一起，我们仍然无法创造出真正的游戏！

游戏的定义并非由其技术，而是由其玩法所决定。玩法可以定义为玩游戏的整体体验。“游戏机制”这一术语更具体一些地阐述了这一概念——它通常被定义为控制游戏中各个实体之间交互的一组规则。它还定义了玩家的目标、成功和失败的标准、玩家角色的能力以及非玩家实体的数量和类型。

15
Introduction to Gameplay Systems

Up until now, everything we've talked about in this book has focused on technology. We've learned that a game engine is a complex, layered software system built on top of the hardware, drivers and operating system of the target machine. We've seen how low-level engine systems provide services that are required by the rest of the engine; how human interface devices such as joypads, keyboards, mice and other devices can allow a human player to provide inputs to the engine; how the rendering engine produces 3D images on-screen; how the animation system allows characters and objects to move naturally; how the collision system detects and resolves interpenetrations between shapes; how the physics simulation causes objects to move in physically realistic ways; how the 3D audio engine renders a believable and immersive soundscape for our game world. But despite the wide range of powerful features provided by these components, if we were to put them all together, we *still* wouldn't have a game!

A game is defined not by its technology but by its *gameplay*. Gameplay can be defined as the overall experience of playing a game. The term *game mechanics* pins down this idea a bit more concretely—it is usually defined as the set of *rules* that govern the interactions between the various entities in the game. It also defines the *objectives* of the player(s), *criteria* for success and failure, the player character's *abilities*, the number and types of *non-player entities*

这些元素存在于游戏的虚拟世界和整体游戏体验之中。在许多游戏中，这些元素与引人入胜的故事和丰富的角色阵容交织在一起。然而，故事和角色并非每款电子游戏的必备要素，像《俄罗斯方块》这样大获成功的益智游戏就证明了这一点。谢菲尔德大学的 Ahmed BinSubaih、Steve Maddock 和 Daniela Romano 在他们的论文《"游戏"可移植性调查》（http://www.dcs.shef.ac.uk/intranet/research/resmes/CS0705.pdf）中，将用于实现游戏玩法的软件系统集合称为游戏的 G 因素。在接下来的三章中，我们将探讨定义和管理游戏机制（又称

游戏的玩法（又称 G 因素）。

15.1 游戏世界的剖析

不同游戏类型和游戏之间的游戏玩法设计差异很大。即便如此，大多数 3D 游戏以及相当一部分 2D 游戏都或多或少地遵循一些基本的结构模式。我们将在后续章节中讨论这些模式，但请记住，肯定有一些游戏并不完全符合这些模式。

15.1.1 世界元素

大多数视频游戏发生在二维或三维的虚拟游戏世界中。这个世界通常由众多离散元素组成。通常，这些元素分为两类：静态元素和动态元素。静态元素包括地形、建筑物、道路、桥梁以及几乎所有不移动或不以主动方式与游戏互动的元素。动态元素包括角色、车辆、武器、浮动道具和生命值包、可收集物品、粒子发射器、动态光源、用于检测游戏中重要事件的不可见区域、定义物体路径的样条线等等。图 15.1 展示了游戏世界的细分。

游戏玩法通常集中在动态元素上。显然，静态背景的布局对游戏的进行至关重要。例如，如果在一个空旷的大矩形房间里玩一款基于掩体的射击游戏，乐趣就会大打折扣。然而，实现游戏玩法的软件系统主要关注的是更新动态元素的位置、方向和内部状态，因为它们是随时间变化的元素。"游戏状态"指的是所有动态游戏世界元素的当前状态。

that exist within the game's virtual world and the *overall flow* of the gaming experience as a whole. In many games, these elements are intertwined with a compelling story and a rich cast of characters. However, story and characters are definitely not a necessary part of every video game, as evidenced by wildly successful puzzle games like *Tetris*. In their paper, "A Survey of 'Game' Portability" (http://www.dcs.shef.ac.uk/intranet/research/resmes/CS0705.pdf), Ahmed BinSubaih, Steve Maddock and Daniela Romano of the University of Sheffield refer to the collection of software systems used to implement gameplay as a game's *G-factor*. In the next three chapters, we'll explore the crucial tools and engine systems that define and manage the *game mechanics* (a.k.a. *gameplay*, a.k.a. *G-factor*) of a game.

15.1 Anatomy of a Game World

Gameplay designs vary widely from genre to genre and from game to game. That said, most 3D games, and a good number of 2D games as well, conform more or less to a few basic structural patterns. We'll discuss these patterns in the following sections, but please keep in mind that there are bound to be games out there that do not fit neatly into this mold.

15.1.1 World Elements

Most video games take place in a two- or three-dimensional virtual *game world*. This world is typically comprised of numerous discrete elements. Generally, these elements fall into two categories: static elements and dynamic elements. Static elements include terrain, buildings, roads, bridges and pretty much anything that doesn't move or interact with gameplay in an active way. Dynamic elements include characters, vehicles, weaponry, floating power-ups and health packs, collectible objects, particle emitters, dynamic lights, invisible regions used to detect important events in the game, splines that define the paths of objects and so on. This breakdown of the game world is illustrated in Figure 15.1.

Gameplay is generally concentrated within the dynamic elements of a game. Clearly, the layout of the static background plays a crucial role in how the game plays out. For example, a cover-based shooter wouldn't be very much fun if it were played in a big, empty, rectangular room. However, the software systems that implement gameplay are primarily concerned with updating the locations, orientations and internal states of the dynamic elements, since they are the elements that change over time. The term *game state* refers to the current state of all dynamic game world elements taken as a whole.

图 15.1。《神秘海域：失落的遗产》（© 2017/TM SIE。由顽皮狗创建和开发，PlayStation 4）中的游戏世界，展示了静态和动态元素。

动态元素与静态元素的比例也因游戏而异。大多数 3D 游戏由相对较少的动态元素组成，这些元素在相对较大的静态背景区域内移动。其他游戏，例如经典街机游戏《小行星》或 Xbox 360 复古热门游戏《几何战争》，则没有任何静态元素（除了黑屏）。游戏的动态元素通常比静态元素更耗费 CPU 资源，因此大多数 3D 游戏都限制在有限数量的动态元素上。但是，动态元素与静态元素的比例越高，游戏世界在玩家眼中就越“生动”。随着游戏硬件越来越强大，游戏的动静比也越来越高。

值得注意的是，游戏世界中动态元素和静态元素之间的界限通常比较模糊。例如，在街机游戏《Hydro Thunder》中，瀑布是动态的，因为它们的纹理是动画的，底部有动态的雾气效果，而且它们可以放置在游戏世界中，并由游戏设计师独立于地形和水面进行定位。然而，从工程学的角度来看，瀑布被视为静态元素，因为它们不与……交互。

Figure 15.1. A game world from *Uncharted: The Lost Legacy* (© 2017/™ SIE. Created and developed by Naughty Dog, PlayStation 4) showing static and dynamic elements.

The ratio of dynamic to static elements also varies from game to game. Most 3D games consist of a relatively small number of dynamic elements moving about within a relatively large static background area. Other games, like the arcade classic *Asteroids* or the Xbox 360 retro hit *Geometry Wars*, have no static elements to speak of (other than a black screen). The dynamic elements of a game are usually more expensive than the static elements in terms of CPU resources, so most 3D games are constrained to a limited number of dynamic elements. However, the higher the ratio of dynamic to static elements, the more “alive” the game world can seem to the player. As gaming hardware becomes more and more powerful, games are achieving higher and higher dynamic-to-static ratios.

It’s important to note that the distinction between the dynamic and static elements in a game world is often a bit blurry. For example, in the arcade game *Hydro Thunder*, the waterfalls were dynamic in the sense that their textures were animated, they had dynamic mist effects at their bases, and they could be placed into the game world and positioned by a game designer independently of the terrain and water surface. However, from an engineering standpoint, waterfalls were treated as static elements because they did not interact with the

比赛中的船只不能以任何方式被遮挡（除了遮挡玩家对隐藏的加速道具和秘密通道的视线）。不同的游戏引擎对静态元素和动态元素的划分有不同的界限，有些引擎甚至根本不划分（即，所有元素都可能成为动态元素）。

静态和动态之间的区别主要作为一种优化工具——当我们知道对象的状态不会改变时，我们可以减少工作量。例如，当我们知道一个网格是静态的并且永远不会移动时，它的光照可以以静态顶点光照、光照贴图、阴影贴图、静态环境光遮蔽信息或预计算辐射传递 (PRT) 球谐函数系数的形式进行预计算。实际上，任何必须在运行时为动态世界元素进行的计算，在应用于静态元素时，都非常适合进行预计算或省略。

具有可破坏环境的游戏是游戏世界中静态元素和动态元素之间界限模糊的一个例子。例如，我们可能会为每个静态元素定义三个版本——未损坏的版本、损坏的版本和完全损坏的版本。这些背景元素在大多数情况下表现得像静态世界元素，但它们可以在爆炸过程中动态切换，从而产生受损的幻觉。实际上，静态和动态世界元素只是一系列可能优化中的两个极端。随着我们的优化方法不断变化并适应游戏设计的需求，我们在这两类元素之间划定的界限（如果我们真的划定了界限的话）也会随之变化。

15.1.1.1 静态几何

静态世界元素的几何形状通常在 Maya 等工具中定义。它可能是一个巨型三角形网格，也可能被分解成离散的碎片。场景的静态部分有时由实例化几何体构建而成。实例化是一种内存节省技术，它使用相对少量的独特三角形网格在整个游戏世界中以不同的位置和方向多次渲染，以提供多样性的视觉效果。例如，3D 建模师可能会创建五种不同类型的短墙段，然后将它们随机组合在一起，以构建绵延数英里、外观独特的墙壁。

静态视觉元素和碰撞数据也可以通过笔刷几何体构建。这种几何体起源于 Quake 系列引擎。笔刷将形状描述为一组凸面体，每个凸面体由一组平面包围。笔刷几何体创建快速简便，并且能够很好地集成到基于 BSP 树的渲染引擎中。笔刷对于快速构建游戏世界的内容非常有用。这使得

boats in the race in any way (other than to obscure the player's view of hidden boost power-ups and secret passageways). Different game engines draw different lines between static and dynamic elements, and some don't draw a distinction at all (i.e., everything is potentially a dynamic element).

The distinction between static and dynamic serves primarily as an optimization tool—we can do less work when we know that the state of an object isn't going to change. For example, when we know a mesh is static and will never move, its lighting can be precomputed in the form of static vertex lighting, light maps, shadow maps, static ambient occlusion information or precomputed radiance transfer (PRT) spherical harmonics coefficients. Virtually any computation that must be done at runtime for a dynamic world element is a good candidate for precomputation or omission when applied to a static element.

Games with destructible environments are an example of how the line between the static and dynamic elements in a game world can blur. For instance, we might define three versions of every static element—an undamaged version, a damaged version, and a fully destroyed version. These background elements act like static world elements most of the time, but they can be swapped dynamically during an explosion to produce the illusion of becoming damaged. In reality, static and dynamic world elements are just two extremes along a gamut of possible optimizations. Where we draw the line between the two categories (if we draw one at all) shifts as our optimization methodologies change and adapt to the needs of the game design.

15.1.1.1 Static Geometry

The geometry of a static world element is often defined in a tool like Maya. It might be one giant triangle mesh, or it might be broken up into discrete pieces. The static portions of the scene are sometimes built out of *instanced geometry*. Instancing is a memory conservation technique in which a relatively small number of unique triangle meshes are rendered multiple times throughout the game world, at different locations and orientations, in order to provide the illusion of variety. For example, a 3D modeler might create five different kinds of short wall sections and then piece them together in random combinations in order to construct miles of unique-looking walls.

Static visual elements and collision data might also be constructed from *brush geometry*. This kind of geometry originated with the Quake family of engines. A *brush* describes a shape as a collection of convex volumes, each bounded by a set of planes. Brush geometry is fast and easy to create and integrates well into a BSP-tree-based rendering engine. Brushes can be really useful for rapidly blocking out the contents of a game world. This allows

在成本较低的情况下，尽早测试游戏玩法。如果布局证明其价值，美术团队可以添加纹理贴图并微调笔刷几何形状，或者将其替换为更精细的自定义网格资源。另一方面，如果关卡需要重新设计，笔刷几何形状可以轻松修改，而无需给美术团队带来大量额外工作。

15.1.2 世界区块

当游戏发生在一个非常大的虚拟世界中时，它通常会被划分为多个独立的可玩区域，我们称之为世界区块。区块也称为关卡、地图、阶段或区域。玩家在游戏过程中通常只能看到少数几个区块，并且随着游戏的展开，玩家会从一个区块推进到另一个区块。

最初，“关卡”的概念是为了在早期游戏硬件的内存限制下提供更丰富的游戏玩法而发明的。内存中一次只能存在一个关卡，但玩家可以逐级前进，获得更丰富的整体体验。从那时起，游戏设计向许多方向发展，如今基于线性关卡的游戏已不常见。有些游戏本质上仍然是线性的，但世界区块之间的界限对玩家来说通常不像以前那么明显。其他游戏使用星型拓扑结构，玩家从中心枢纽区域开始，可以从枢纽随机访问其他区域（可能只有在解锁后才能访问）。其他游戏使用类似图形的拓扑结构，其中区域以任意方式相互连接。还有一些游戏营造出广阔开放世界的氛围，并使用细节层次 (LOD) 技术来减少内存开销并提高性能。

尽管现代游戏设计丰富多彩，但除了最小的几个世界之外，所有游戏世界仍然被划分成某种形式的区块。这样做的原因有很多。首先，内存限制仍然是一个重要的制约因素（在拥有无限内存的游戏机上市之前，这种情况将一直存在！）。世界区块也是一种控制游戏整体流程的便捷机制。区块还可以作为一种分工机制；每个区块可以由相对较小的设计师和美术团队构建和管理。世界区块如图 15.2 所示。

15.1.3 高级游戏流程

游戏的高层流程定义了玩家目标的序列、树状图或图表。目标有时被称为任务、阶段、等级（该术语也适用于世界区块）或波次（如果游戏主要涉及击败成群的进攻敌人）。高层流程还提供了每个目标的成功定义（例如，清除所有敌人并获得钥匙）以及惩罚

gameplay to be tested early, when it is cheap to do so. If the layout proves its worth, the art team can either texture map and fine-tune the brush geometry or replace it with more-detailed custom mesh assets. On the other hand, if the level requires redesign, the brush geometry can be easily revised without creating a lot of extra work for the art team.

15.1.2 World Chunks

When a game takes place in a very large virtual world, it is typically divided into discrete playable regions, which we'll call *world chunks*. Chunks are also known as *levels*, *maps*, *stages* or *areas*. The player can usually see only a handful of chunks at any given moment while playing the game, and he or she progresses from chunk to chunk as the game unfolds.

Originally, the concept of "levels" was invented as a mechanism to provide greater variety of gameplay within the memory limitations of early gaming hardware. Only one level could exist in memory at a time, but the player could progress from level to level for a much richer overall experience. Since then, game designs have branched out in many directions, and linear level-based games are much less common today. Some games are essentially still linear, but the delineations between world chunks are usually not as obvious to the player as they once were. Other games use a star topology, in which the player starts in a central hub area and can access other areas at random from the hub (perhaps only after they have been unlocked). Others use a graph-like topology, where areas are connected to one another in arbitrary ways. Still others provide the illusion of a vast, open world, and use level-of-detail (LOD) techniques to reduce memory overhead and improve performance.

Despite the richness of modern game designs, all but the smallest of game worlds are still divided into chunks of some kind. This is done for a number of reasons. First of all, memory limitations are still an important constraint (and will be until game machines with infinite memory hit the market!). World chunks are also a convenient mechanism for controlling the overall flow of the game. Chunks can serve as a division-of-labor mechanism as well; each chunk can be constructed and managed by a relatively small group of designers and artists. World chunks are illustrated in Figure 15.2.

15.1.3 High-Level Game Flow

A game's *high-level flow* defines a sequence, tree or graph of player *objectives*. Objectives are sometimes called *tasks*, *stages*, *levels* (a term that can also apply to world chunks) or *waves* (if the game is primarily about defeating hordes of attacking enemies). The high-level flow also provides the definition of success for each objective (e.g., clear all the enemies and get the key) and the penalty

图 15.2。许多游戏世界由于各种原因被划分为多个块，包括内存限制、控制游戏世界流程的需要以及作为开发过程中的分工机制。

失败时（例如，返回当前区域的起点，在此过程中可能失去一条“生命”）。在剧情驱动的游戏中，此流程可能还包含各种游戏内影片，用于随着剧情的展开加深玩家对故事的理解。这些序列有时被称为过场动画、游戏内过场动画 (IGC) 或非交互序列 (NIS)。当它们离线渲染并以全屏影片形式播放时，此类序列通常被称为全动态视频 (FMV)。

早期的游戏将玩家的目标一一映射到特定的世界块（因此“级别”一词具有双重含义）。例如，在《大金刚》中，每个新级别都会为马里奥提出一个新的目标（即，到达结构的顶部并进入下一级）。然而，世界块和目标之间的这种一对一映射在现代游戏设计中并不那么流行。每个目标都与一个或多个世界块相关联，但块和目标之间的耦合仍然很松散。这种设计提供了独立改变游戏目标和世界细分的灵活性，这在开发游戏时从逻辑和实践的角度来看非常有用。许多游戏将其目标分组为更粗略的游戏部分，通常称为章节或动作。典型的游戏架构如图 15.3 所示。

Figure 15.2. Many game worlds are divided into chunks for various reasons, including memory limitations, the need to control the flow of the game through the world, and as a division-of-labor mechanism during development.

for failure (e.g., go back to the start of the current area, possibly losing a “life” in the process). In a story-driven game, this flow might also include various in-game movies that serve to advance the player’s understanding of the story as it unfolds. These sequences are sometimes called *cut-scenes*, *in-game cinematics* (IGC) or *noninteractive sequences* (NIS). When they are rendered offline and played back as a full-screen movie, such sequences are usually called *full-motion videos* (FMV).

Early games mapped the objectives of the player one-to-one to particular world chunks (hence the dual meaning of the term “level”). For example, in *Donkey Kong*, each new level presents Mario with a new objective (namely, to reach the top of the structure and progress to the next level). However, this one-to-one mapping between world chunks and objectives is less popular in modern game design. Each objective is associated with one or more world chunks, but the coupling between chunks and objectives remains deliberately loose. This kind of design offers the flexibility to alter game objectives and world subdivision independently, which is extremely helpful from a logistic and practical standpoint when developing a game. Many games group their objectives into coarser sections of gameplay, often called *chapters* or *acts*. A typical gameplay architecture is shown in Figure 15.3.

图 15.3 游戏目标通常按序列、树或广义图排列，每个目标映射到一个或多个游戏世界块。

15.2 实现动态元素：游戏对象

游戏中的动态元素通常以面向对象的方式设计。这种方法直观自然，并且与游戏设计师对世界构建方式的理念非常吻合。他或她可以将角色、车辆、漂浮的医疗包、爆炸的桶以及游戏中移动的无数其他动态物体可视化。因此，希望能够在游戏世界编辑器中创建和操作这些元素是很自然的。同样，程序员通常会发现在运行时将动态元素实现为高度自主的代理是很自然的。在本书中，我们将使用术语游戏对象 (GO) 来指代游戏世界中的几乎任何动态元素。然而，这个术语绝不是业界的标准。游戏对象通常被称为实体、参与者或代理，术语列表还在继续。

Figure 15.3. Gameplay objectives are typically arranged in a sequence, a tree, or a generalized graph, and each one maps to one or more game world chunks.

15.2 Implementing Dynamic Elements: Game Objects

The dynamic elements of a game are usually designed in an object-oriented fashion. This approach is intuitive and natural and maps well to the game designer's notion of how the world is constructed. He or she can visualize characters, vehicles, floating health packs, exploding barrels and myriad other dynamic objects moving about in the game. So it is only natural to want to be able to create and manipulate these elements in the game world editor. Likewise, programmers usually find it natural to implement dynamic elements as largely autonomous agents at runtime. In this book, we'll use the term *game object* (GO) to refer to virtually any dynamic element within a game world. However, this terminology is by no means standard within the industry. Game objects are commonly referred to as *entities*, *actors* or *agents*, and the list of terms goes on.

按照面向对象设计的惯例，游戏对象本质上是属性（对象的当前状态）和行为（状态如何随时间变化以及响应事件而变化）的集合。游戏对象通常按类型分类。不同类型的对象具有不同的属性模式和行为。同一类型的所有实例共享相同的属性模式和相同的行为集，但属性值因实例而异。（请注意，如果游戏对象的行为是由数据驱动的，例如通过脚本代码或一组控制对象对事件响应的数据驱动规则，那么行为也会因实例而异。）

类型和类型实例之间的区别至关重要。例如，吃豆人游戏涉及四种游戏对象类型：幽灵、弹丸、能量药丸和吃豆人。然而，在任意时刻，最多可能有四个“幽灵”类型的实例，50-100 个“弹丸”类型的实例，四个“能量药丸”类型的实例和一个“吃豆人”类型的实例。

大多数面向对象系统都提供了属性、行为或两者的继承机制。继承有助于代码和设计的重用。虽然不同游戏的继承机制差异很大，但大多数游戏引擎都以某种形式支持继承。

15.2.1 游戏对象模型

在计算机科学中，“对象模型”一词有两种既相关又不同的含义。它可以指某种编程语言或形式化设计语言所提供的一组功能。例如，我们可能会提到 C++ 对象模型或 OMT 对象模型。它也可以指特定的面向对象编程接口（即，为解决特定问题而设计的类、方法和相互关系的集合）。后者用法的一个例子是 Microsoft Excel 对象模型，它允许外部程序以各种方式控制 Excel。（有关“对象模型”一词的进一步讨论，请参阅 http://en.wikipedia.org/wiki/Object_model 。）

在本书中，我们将使用“游戏对象模型”一词来描述游戏引擎提供的用于对虚拟游戏世界中的动态实体进行建模和模拟的功能。从这个意义上讲，“游戏对象模型”一词兼具上述两个定义的特点：

- 游戏的对象模型是一种特定的面向对象编程接口，旨在解决模拟构成特定游戏的特定实体集的特定问题。
- 此外，游戏的对象模型通常会扩展引擎编写的编程语言。如果游戏实现

As is customary in object-oriented design, a game object is essentially a collection of *attributes* (the current state of the object) and *behaviors* (how the state changes over time and in response to events). Game objects are usually classified by *type*. Different types of objects have different attribute schemas and different behaviors. All *instances* of a particular type share the same attribute schema and the same set of behaviors, but the *values* of the attributes differ from instance to instance. (Note that if a game object's *behavior* is data-driven, say through script code or via a set of data-driven rules governing the object's responses to events, then behavior too can vary on an instance-by-instance basis.)

The distinction between a *type* and an *instance* of a type is a crucial one. For example, the game of *Pac-Man* involves four game object types: ghosts, pellets, power pills and Pac-Man. However, at any moment in time, there may be up to four instances of the type "ghost," 50–100 instances of the type "pellet," four "power pill" instances and one instance of the "Pac-Man" type.

Most object-oriented systems provide some mechanism for the *inheritance* of attributes, behavior or both. Inheritance encourages code and design reuse. The specifics of how inheritance works vary widely from game to game, but most game engines support it in some form.

15.2.1 Game Object Models

In computer science, the term *object model* has two related but distinct meanings. It can refer to the set of features provided by a particular programming language or formal design language. For example, we might speak of the *C++ object model* or the *OMT object model*. It can also refer to a specific object-oriented programming interface (i.e., a collection of classes, methods and interrelationships designed to solve a particular problem). One example of this latter usage is the *Microsoft Excel object model*, which allows external programs to control Excel in various ways. (See http://en.wikipedia.org/wiki/Object_model for further discussion of the term *object model*.)

In this book, we will use the term *game object model* to describe the facilities provided by a game engine in order to permit the dynamic entities in the virtual game world to be modeled and simulated. In this sense, the term *game object model* has aspects of *both* of the definitions given above:

- A game's object model is a specific object-oriented programming interface intended to solve the particular problem of simulating the specific set of entities that make up a particular game.
- Additionally, a game's object model often extends the programming language in which the engine was written. If the game is implemented

在非面向对象语言（例如 C）中，程序员可以添加面向对象功能。即使游戏是用 C++ 等面向对象语言编写的，也经常会添加反射、持久化和网络复制等高级功能。游戏对象模型有时会融合多种语言的特性。例如，游戏引擎可能会将 C 或 C++ 等编译型编程语言与 Python、Lua 或 Pawn 等脚本语言相结合，并提供一个可以通过任一语言访问的统一对象模型。

15.2.2 工具端设计与运行时设计

通过世界编辑器向设计师呈现的对象模型（在第 15.4 节中讨论）不必与在运行时实现游戏的对象模型相同。

- 工具端游戏对象模型可能在运行时使用完全没有本机面向对象功能的语言（如 C）来实现。
- 工具端的单个 GO 类型可能在运行时实现为类的集合（而不是像人们最初期望的那样实现为单个类）。
- 每个工具端 GO 在运行时可能只不过是一个唯一的 id，其所有状态数据都存储在表或松散耦合对象的集合中。

因此，游戏实际上有两个不同但紧密相关的对象模型：

- 工具端对象模型由设计师在世界编辑器中看到的游戏对象类型集定义。
- 运行时对象模型由程序员在运行时实现工具端对象模型时所使用的语言结构和软件系统定义。运行时对象模型可能与工具端模型完全相同，或直接映射到工具端模型，也可能与工具端模型在底层完全不同。

在某些游戏引擎中，工具端和运行时设计之间的界限模糊不清，甚至根本不存在。而在其他引擎中，两者界限却非常清晰。在某些引擎中，工具端和运行时的实现实际上是共享的。而在其他引擎中，运行时的实现与工具端的视角几乎完全不同。实现的某些方面几乎总是会潜移默化地渗透到工具端设计中，游戏设计师必须意识到他们所设计的游戏世界对性能和内存消耗的影响。

in a non-object-oriented language like C, object-oriented facilities can be added by the programmers. And even if the game is written in an object-oriented language like C++, advanced features like reflection, persistence and network replication are often added. A game object model sometimes melds the features of multiple languages. For example, a game engine might combine a compiled programming language such as C or C++ with a scripting language like Python, Lua or Pawn and provide a unified object model that can be accessed from either language.

15.2.2 Tool-Side Design versus Runtime Design

The object model presented to the designers via the world editor (discussed in Section 15.4) needn't be the same object model used to implement the game at runtime.

- The tool-side game object model might be implemented at runtime using a language with no native object-oriented features at all, like C.
- A single GO type on the tool side might be implemented as a collection of classes at runtime (rather than as a single class as one might at first expect).
- Each tool-side GO might be nothing more than a unique id at runtime, with all of its state data stored in tables or collections of loosely coupled objects.

Therefore, a game really has two distinct but closely interrelated object models:

- The *tool-side object model* is defined by the set of *game object types* seen by the designers within the world editor.
- The *runtime object model* is defined by whatever set of language constructs and software systems the programmers have used to implement the tool-side object model at runtime. The runtime object model might be identical to the tool-side model or map directly to it, or it might be entirely different than the tool-side model under the hood.

In some game engines, the line between the tool-side and runtime designs is blurred or nonexistent. In others, it is very well delineated. In some engines, the implementation is actually shared between the tools and the runtime. In others, the runtime implementation looks almost totally alien relative to the tool-side view of things. Some aspects of the implementation almost always creep up into the tool-side design, and game designers must be cognizant of the performance- and memory-consumption impacts of the game worlds they

构造以及它们设计的游戏规则和对象行为。也就是说，几乎所有游戏引擎都具有某种形式的工具端对象模型以及该对象模型的相应运行时实现。

15.3 数据驱动的游戏引擎

在游戏开发的早期，游戏大多由程序员硬编码完成。工具（如果有的话）也非常简陋。这种做法之所以有效，是因为典型游戏中的内容量极少，而且门槛也并不高，这在一定程度上要归功于早期游戏硬件所能提供的原始图像和音效。

如今，游戏的复杂程度已大幅提升，质量标准也高得惊人，以至于游戏内容常常被比作好莱坞大片的电脑特效。游戏团队的规模也大幅扩张，但游戏内容数量的增长速度却远超团队规模。在以 Xbox One 和 PlayStation 4 为代表的第八代主机中，游戏团队经常表示，他们需要在规模与上一代并无太大差异的情况下，制作出十倍于上一代的游戏内容。这种趋势意味着游戏团队必须能够以极其高效的方式制作海量内容。

工程资源常常成为生产瓶颈，因为高质量的工程人才有限且价格昂贵，而且由于计算机编程固有的复杂性，工程师制作内容的速度往往比美术师和游戏设计师慢得多。现在大多数团队认为，将内容创作的部分权力直接交给负责制作内容的人员——也就是设计师和美术师——是一个好主意。当游戏的行为可以全部或部分由美术师和设计师提供的数据控制，而不是完全由程序员编写的软件控制时，我们就说引擎是数据驱动的。

数据驱动架构可以充分发挥所有团队成员的潜力，并减轻工程团队的压力，从而提高团队效率。它还可以缩短迭代时间。无论开发者是想对游戏内容进行细微调整，还是彻底修改整个关卡，数据驱动设计都能让开发者快速看到更改的效果，理想情况下几乎不需要工程师的帮助。这节省了宝贵的时间，并能让团队将游戏打磨到极高的质量水平。

话虽如此，我们必须意识到数据驱动的功能往往需要付出高昂的代价。必须提供工具，让游戏设计师和

construct and the gameplay rules and object behaviors they design. That said, virtually all game engines have some form of tool-side object model and a corresponding runtime implementation of that object model.

15.3 Data-Driven Game Engines

In the early days of game development, games were largely hard-coded by programmers. Tools, if any, were primitive. This worked because the amount of content in a typical game was miniscule, and the bar wasn't particularly high, thanks in part to the primitive graphics and sound of which early game hardware was capable.

Today, games are orders of magnitude more complex, and the quality bar is so high that game content is often compared to the computer-generated effects in Hollywood blockbusters. Game teams have grown much larger, but the amount of game content is growing faster than team size. In the eighth generation of consoles, defined by the Xbox One and the PlayStation 4, game teams routinely speak of the need to produce ten times the content with teams that are not that much larger than in the previous generation. This trend means that a game team must be capable of producing very large amounts of content in an extremely efficient manner.

Engineering resources are often a production bottleneck because high-quality engineering talent is limited and expensive and because engineers tend to produce content much more slowly than artists and game designers (due to the complexities inherent in computer programming). Most teams now believe that it's a good idea to put at least some of the power to create content directly into the hands of the folks responsible for producing that content—namely the designers and the artists. When the behavior of a game can be controlled, in whole or in part, by *data* provided by artists and designers rather than exclusively by *software* produced by programmers, we say the engine is *data driven*.

Data-driven architectures can improve team efficiency by fully leveraging all staff members to their fullest potential and by taking some of the heat off the engineering team. It can also lead to improved *iteration times*. Whether a developer wants to make a slight tweak to the game's content or completely revise an entire level, a data-driven design allows the developer to see the effects of the changes quickly, ideally with little or no help from an engineer. This saves valuable time and can permit the team to polish their game to a very high level of quality.

That being said, it's important to realize that data-driven features often come at a heavy cost. Tools must be provided to allow game designers and

艺术家需要以数据驱动的方式定义游戏内容。运行时代码必须进行修改，以便以稳健的方式处理各种可能的输入。游戏中还必须提供工具，以便艺术家和设计师预览其作品并解决问题。所有这些软件的编写、测试和维护都需要大量的时间和精力。

遗憾的是，许多团队一头扎进数据驱动架构，却没有停下来研究他们的努力对特定游戏设计和团队成员特定需求的影响。仓促之下，这些团队往往会大大超出目标，开发出过于复杂的工具和引擎系统，这些系统难以使用、漏洞百出，几乎无法适应项目不断变化的需求。讽刺的是，在努力实现数据驱动设计的优势的过程中，团队的生产力很容易比传统的硬编码方法低得多。

每个游戏引擎都应该包含一些数据驱动的组件，但游戏团队在选择引擎的哪些方面进行数据驱动时必须格外谨慎。至关重要的是，要权衡创建数据驱动或快速迭代功能的成本，以及该功能预计在项目过程中为团队节省的时间。在设计和实现数据驱动工具和引擎系统时，牢记 KISS 原则（“保持简单，愚蠢”）也至关重要。套用爱因斯坦的话来说，游戏引擎中的一切都应该尽可能简单，但不能过于简单。

15.4 游戏世界编辑器

我们已经讨论过数据驱动的资源创建工具，例如 Maya、Photoshop、Havok 内容工具等等。这些工具可以生成单独的资源，供渲染引擎、动画系统、音频系统、物理系统等使用。在游戏玩法领域，与这些工具类似的是游戏世界编辑器——一个（或一套）工具，允许定义游戏世界区块，并填充静态和动态元素。

所有商业游戏引擎都具有某种世界编辑器工具。

- 著名的Radiant工具用于为Quake和Doom系列引擎创建地图。Radiant的屏幕截图如图15.4所示。
- Valve 的 Source 引擎（驱动《半条命 2》、《橙盒》、《军团要塞 2》、《传送门》系列、《求生之路》系列和《泰坦陨落》的引擎）提供了一个名为 Hammer 的编辑器（之前以 Worldcraft 和 The Forge 的名称发行）。图 15.5 展示了 Hammer 的屏幕截图。

artists to define game content in a data-driven manner. The runtime code must be changed to handle the wide range of possible inputs in a robust way. Tools must also be provided in-game to allow artists and designers to preview their work and troubleshoot problems. All of this software requires significant time and effort to write, test and maintain.

Sadly, many teams make a mad rush into data-driven architectures without stopping to study the impacts of their efforts on their particular game design and the specific needs of their team members. In their haste, such teams often dramatically overshoot the mark, producing overly complex tools and engine systems that are difficult to use, bug-ridden and virtually impossible to adapt to the changing requirements of the project. Ironically, in their efforts to realize the benefits of a data-driven design, a team can easily end up with significantly lower productivity than the old-fashioned hard-coded methods.

Every game engine should have some data-driven components, but a game team must exercise extreme care when selecting which aspects of the engine to data-drive. It's crucial to weigh the costs of creating a data-driven or rapid-iteration feature against the amount of time the feature is expected to save the team over the course of the project. It's also incredibly important to keep the KISS mantra ("keep it simple, stupid") in mind when designing and implementing data-driven tools and engine systems. To paraphrase Albert Einstein, everything in a game engine should be made as simple as possible, but no simpler.

15.4 The Game World Editor

We've already discussed data-driven asset-creation tools, such as Maya, Photoshop, Havok content tools and so on. These tools generate individual assets for consumption by the rendering engine, animation system, audio system, physics system and so on. The analog to these tools in the gameplay space is the *game world editor*—a tool (or a suite of tools) that permits game world chunks to be defined and populated with static and dynamic elements.

All commercial game engines have some kind of world editor tool.

- A well-known tool called *Radiant* is used to create maps for the *Quake* and *Doom* family of engines. A screenshot of Radiant is shown in Figure 15.4.
- Valve's *Source* engine, the engine that drives *Half-Life 2*, *The Orange Box*, *Team Fortress 2*, the *Portal* series, the *Left 4 Dead* series and *Titanfall*, provides an editor called *Hammer* (previously distributed under the names *Worldcraft* and *The Forge*). Figure 15.5 shows a screenshot of Hammer.

图 15.4.Quake 和 Doom 系列引擎的 Radiant 世界编辑器。

• Crytek 的 CRYENGINE 提供了一套强大的世界创建和编辑工具。这些工具支持同时实时编辑多平台游戏环境，包括 2D 和真正的立体 3D 格式。Crytek 的沙盒编辑器如图 15.6 所示。

游戏世界编辑器通常允许指定游戏对象的初始状态（即其属性值）。大多数游戏世界编辑器还允许用户控制游戏世界中动态对象的行为。这种控制可能是通过数据驱动的配置参数（例如，对象 A 应以不可见状态启动，对象 B 应在生成时立即攻击玩家，对象 C 应为易燃状态等）进行的，也可能是通过脚本语言进行行为控制，从而将游戏设计师的任务转移到编程领域。一些世界编辑器甚至允许定义全新类型的游戏对象，几乎无需程序员干预。

Figure 15.4. The Radiant world editor for the Quake and Doom family of engines.

- Crytek's CRYENGINE provides a powerful suite of world creation and editing tools. These tools support real-time editing of multiplatform game environments simultaneously, both in 2D and true stereoscopic 3D. Crytek's Sandbox editor is depicted in Figure 15.6.

The game world editor generally permits the initial states of game objects (i.e., the values of their attributes) to be specified. Most game world editors also give their users some sort of ability to control the *behaviors* of the dynamic objects in the game world. This control might be via data-driven configuration parameters (e.g., object A should start in an invisible state, object B should immediately attack the player when spawned, object C is flammable, etc.), or behavioral control might be via a scripting language, thereby shifting the game designers' tasks into the realm of programming. Some world editors even allow entirely new types of game objects to be defined, with little or no programmer intervention.

图 15.5. Valve 的 Source 引擎 Hammer 编辑器。

Figure 15.5. Valve's Hammer editor for the Source engine.

图 15.6。CRYENGINE 的沙盒编辑器。（参见彩色图 XXV。）

Figure 15.6. The Sandbox editor for CRYENGINE. (See Color Plate XXV.)

15.4.1 游戏世界编辑器的典型特征

游戏世界编辑器的设计和布局千差万别，但大多数编辑器都提供了一套相当标准的功能。这些功能包括但不限于以下内容。

15.4.1.1 世界块的创建和管理

世界创建的单位通常为区块（也称为关卡或地图——参见第 15.1.2 节）。游戏世界编辑器通常允许创建新区块，并重命名、拆分、合并或销毁现有区块。每个区块可以链接到一个或多个静态网格和/或其他静态数据元素，例如 AI 导航地图、玩家可抓取的壁架描述、掩体点定义等等。在某些引擎中，区块由单个背景网格定义，且离不开背景网格。在其他引擎中，区块可能独立存在，可能由边界框（例如 AABB、OBB 或任意多边形区域）定义，并且可以由零个或多个网格和/或画刷几何体填充（参见第 1.7.2.1 节）。

一些世界编辑器提供专用工具来创作地形、水体和其他特殊静态元素。在其他引擎中，这些元素可能使用标准 DC□□C 应用程序创作，但会以某种方式进行标记，以向资源调节管道和/或运行时引擎表明它们是特殊的。（例如，在《神秘海域》和《最后生还者》系列中，水体被创作为三角形网格，但它被贴图了一种特殊材质，表明它应该被视为水。）有时，特殊的世界元素是在单独的独立工具中创建和编辑的。例如，《荣誉勋章：太平洋突袭》中的高场地形是使用从艺电内部另一个团队获得的工具的定制版本创作的，因为这比尝试将地形编辑器集成到当时项目使用的世界编辑器 Radiant 中更为便捷。

15.4.1.2 游戏世界可视化

对于游戏世界编辑器的用户来说，能够将游戏世界的内容可视化非常重要。因此，几乎所有游戏世界编辑器都提供世界的三维透视视图和/或二维正交投影。视图窗格通常分为四个部分，其中三个用于顶部、侧面和正面正交立面图，一个用于 3D 透视视图。

有些编辑器通过直接集成到工具中的自定义渲染引擎提供这些世界观。其他编辑器则直接集成到

15.4.1 Typical Features of a Game World Editor

The design and layout of game world editors vary widely, but most editors provide a reasonably standard set of features. These include, but are certainly not limited to, the following.

15.4.1.1 World Chunk Creation and Management

The unit of world creation is usually a chunk (also known as a level or map—see Section 15.1.2). The game world editor typically allows new chunks to be created and existing chunks to be renamed, broken up, combined or destroyed. Each chunk can be linked to one or more static meshes and/or other static data elements such as AI navigation maps, descriptions of ledges that can be grabbed by the player, cover point definitions and so on. In some engines, a chunk is defined by a single background mesh and cannot exist without one. In other engines, a chunk may have an independent existence, perhaps defined by a bounding volume (e.g., AABB, OBB or arbitrary polygonal region), and can be populated by zero or more meshes and/or brush geometry (see Section 1.7.2.1).

Some world editors provide dedicated tools for authoring terrain, water and other specialized static elements. In other engines, these elements might be authored using standard DCC applications but tagged in some way to indicate to the asset conditioning pipeline and/or the runtime engine that they are special. (For example, in the *Uncharted* and *The Last of Us* series, water was authored as a triangle mesh, but it was mapped with a special material that indicated that it was to be treated as water.) Sometimes, special world elements are created and edited in a separate, stand-alone tool. For example, the height field terrain in *Medal of Honor: Pacific Assault* was authored using a customized version of a tool obtained from another team within Electronic Arts because this was more expedient than trying to integrate a terrain editor into Radiant, the world editor being used on the project at the time.

15.4.1.2 Game World Visualization

It's important for the user of a game world editor to be able to visualize the contents of the game world. As such, virtually all game world editors provide a three-dimensional perspective view of the world and/or a two-dimensional orthographic projection. It's common to see the view pane divided into four sections, three for top, side and front orthographic elevations and one for the 3D perspective view.

Some editors provide these world views via a custom rendering engine integrated directly into the tool. Other editors are themselves integrated into

像 Maya 或 3ds Max 这样的 3D 几何编辑器，因此他们可以简单地利用这些工具的视口。还有一些编辑器被设计为与实际的游戏引擎通信，并使用它渲染 3D 透视视图。有些编辑器甚至集成到了引擎本身。

15.4.1.3 导航

显然，如果用户无法在游戏世界中移动，世界编辑器就没什么用。在正交视图中，能够滚动、放大和缩小非常重要。在 3D 视图中，可以使用各种摄像机控制方案。可以聚焦于单个对象并围绕其旋转。也可以切换到“飞行”模式，在该模式下，摄像机围绕其焦点旋转，并可以前后、上下和左右移动。

一些编辑器提供了一系列便捷的导航功能。这些功能包括：只需按一下键即可选中并聚焦于某个对象；保存多个相关摄像机位置并在它们之间切换；提供多种用于粗略导航和精细控制的摄像机移动速度模式；以及类似网页浏览器的导航历史记录，可用于在游戏世界中跳转等等。

15.4.1.4 选择

游戏世界编辑器的主要设计目的是让用户能够用静态和动态元素填充游戏世界。因此，用户能够选择单个元素进行编辑至关重要。有些编辑器一次只允许选择一个对象，而更高级的编辑器则允许同时选择多个对象。对象可以通过正交视图中的橡皮筋框选择，也可以通过 3D 视图中的光线投射样式选择。许多编辑器还会以滚动列表或树状视图的形式显示所有世界元素的列表，以便用户可以通过名称查找和选择对象。有些世界编辑器还允许命名并保存所选对象以供日后检索。

游戏世界中的物体通常非常密集。因此，有时由于其他物体的遮挡，选择所需的物体可能会很困难。这个问题可以通过多种方式解决。当使用射线投射在 3D 环境中选择物体时，编辑器可能会允许用户循环浏览射线当前相交的所有物体，而不是始终选择最近的物体。许多编辑器允许将当前选定的物体暂时隐藏在视图之外。这样，如果您第一次没有找到想要的物体，您可以随时将其隐藏并重试。正如我们将在下一节中看到的，图层也是减少混乱并提高用户成功选择物体能力的有效方法。

a 3D geometry editor like Maya or 3ds Max, so they can simply leverage the tool's viewports. Still other editors are designed to communicate with the actual game engine and use it to render the 3D perspective view. Some editors are even integrated into the engine itself.

15.4.1.3 Navigation

Clearly, a world editor wouldn't be of much use if the user weren't able to move around within the game world. In an orthographic view, it's important to be able to scroll and zoom in and out. In a 3D view, various camera control schemes are used. It may be possible to focus on an individual object and rotate around it. It may also be possible to switch into a "fly through" mode where the camera rotates about its own focal point and can be moved forward, backward, up and down and panned left and right.

Some editors provide a host of convenience features for navigation. These include the ability to select an object and focus in on it with a single key press, the ability to save various relevant camera locations and then jump between them, various camera movement speed modes for coarse navigation and fine camera control, a Web-browser-like navigation history that can be used to jump around the game world and so on.

15.4.1.4 Selection

A game world editor is primarily designed to allow the user to populate a game world with static and dynamic elements. As such, it's important for the user to be able to select individual elements for editing. Some editors only allow a single object to be selected at a time, while more-advanced editors permit multiobject selections. Objects might be selected via a rubber-band box in the orthographic view or by ray cast style picking in the 3D view. Many editors also display a list of all world elements in a scrolling list or tree view so that objects can be found and selected by name. Some world editors also allow selections to be named and saved for later retrieval.

Game worlds are often quite densely populated. As such, it can sometimes be difficult to select a desired object because other objects are in the way. This problem can be overcome in a number of ways. When using a ray cast to select objects in 3D, the editor might allow the user to cycle through all of the objects that the ray is currently intersecting rather than always selecting the nearest one. Many editors allow the currently selected object(s) to be temporarily hidden from view. That way, if you don't get the object you want the first time, you can always hide it and try again. As we'll see in the next section, layers can also be an effective way to reduce clutter and improve the user's ability to select objects successfully.

15.4.1.5 层

一些编辑器还允许将对象分组到预定义或用户定义的图层中。这是一个非常有用的功能，可以合理地组织游戏世界的内容。可以隐藏或显示整个图层，以减少屏幕上的混乱。图层可以使用颜色编码以便于识别。图层也可以成为分工策略的重要组成部分。例如，当光照团队处理世界区块时，他们可以隐藏场景中所有与光照无关的元素。

更重要的是，如果游戏世界编辑器能够单独加载和保存图层，那么当多个人同时处理同一个世界块时，就可以避免冲突。例如，所有灯光可能存储在一个图层中，所有背景几何体存储在另一个图层中，所有AI角色存储在第三个图层中。由于每个图层都完全独立，因此灯光、背景和NPC团队可以同时处理同一个世界块。

15.4.1.6 属性网格

构成游戏世界块的静态和动态元素通常具有各种属性（也称为特性），可供用户编辑。属性可以是简单的键值对，并且仅限于简单的原子数据类型，例如布尔值、整数、浮点数和字符串。某些编辑器支持更复杂的属性，包括数据数组和嵌套复合数据结构。也可能支持更复杂的数据类型，例如矢量、RGB 颜色以及对外部资源（音频文件、网格、动画等）的引用。

大多数世界编辑器会将当前选定对象的属性显示在可滚动的属性网格视图中。图 15.7 展示了一个属性网格的示例。该网格允许用户查看每个属性的当前值，并通过键入、使用复选框或下拉组合框、上下拖动微调控件等方式来编辑这些值。

编辑多对象选择

在支持多对象选择的编辑器中，属性网格也可能支持多对象编辑。此高级功能会显示所选对象中所有对象的属性组合。如果某个属性在所选对象中的所有对象中具有相同的值，则该值将按原样显示；在网格中编辑该值会导致所有选定对象中的属性值更新。如果所选对象中各个对象的属性值不同，属性网格通常不显示任何值。在这种情况下，如果新值

15.4.1.5 Layers

Some editors also allow objects to be grouped into predefined or user-defined *layers*. This can be an incredibly useful feature, allowing the contents of the game world to be organized sensibly. Entire layers can be hidden or shown to reduce clutter on-screen. Layers might be color-coded for easy identification. Layers can be an important part of a division-of-labor strategy, as well. For example, when the lighting team is working on a world chunk, they can hide all of the elements in the scene that are not relevant to lighting.

What's more, if the game world editor is capable of loading and saving layers individually, conflicts can be avoided when multiple people are working on a single world chunk at the same time. For example, all of the lights might be stored in one layer, all of the background geometry in another and all AI characters in a third. Since each layer is totally independent, the lighting, background and NPC teams can all work simultaneously on the same world chunk.

15.4.1.6 Property Grid

The static and dynamic elements that populate a game world chunk typically have various properties (also known as attributes) that can be edited by the user. Properties might be simple key-value pairs and be limited to simple atomic data types like Booleans, integers, floating-point numbers and strings. In some editors, more complex properties are supported, including arrays of data and nested compound data structures. More complex data types may be supported too, such as vectors, RGB colors and references to external assets (audio files, meshes, animations, etc.)

Most world editors display the attributes of the currently selected object(s) in a scrollable property grid view. An example of a property grid is shown in Figure 15.7. The grid allows the user to see the current values of each attribute and edit the values by typing, using check boxes or drop-down combo boxes, dragging spinner controls up and down and so on.

Editing Multiobject Selections

In editors that support multiobject selection, the property grid may support multiobject editing as well. This advanced feature displays an amalgam of the attributes of all objects in the selection. If a particular attribute has the same value across all objects in the selection, the value is shown as-is, and editing the value in the grid causes the property value to be updated in all selected objects. If the attribute's value differs from object to object within the selection, the property grid typically shows no value at all. In this case, if a new value

图 15.7. 典型的属性网格。

在网格中的字段中输入，它将覆盖所有选定对象中的值，使它们全部一致。

当选择包含异构对象集合（即类型不同的对象）时，会出现另一个问题。每种类型的对象都可能具有不同的属性集，因此属性网格必须仅显示选择中所有对象类型共有的属性。然而，这仍然很有用，因为游戏对象类型通常继承自一个通用基类型。例如，大多数对象都具有位置和方向。在异构选择中，即使更具体的属性暂时隐藏在视图之外，用户仍然可以编辑这些共享属性。

自由形式属性

通常，与对象关联的属性集及其数据类型是根据每个对象类型定义的。例如，可渲染对象具有位置、方向、比例和网格，而光源具有位置、方向、颜色、强度和光源类型。某些编辑器还允许用户根据每个实例定义其他“自由形式”的属性。

Figure 15.7. A typical property grid.

is typed into the field in the grid, it will overwrite the values in all selected objects, bringing them all into agreement.

Another problem arises when the selection contains a heterogeneous collection of objects (i.e., objects whose types differ). Each type of object can potentially have a different set of attributes, so the property grid must display only those attributes that are common to all object types in the selection. This can still be useful, however, because game object types often inherit from a common base type. For example, most objects have a position and orientation. In a heterogeneous selection, the user can still edit these shared attributes even though more-specific attributes are temporarily hidden from view.

Free-Form Properties

Normally, the set of properties associated with an object, and the data types of those properties, are defined on a per-object-type basis. For example, a renderable object has a position, orientation, scale and mesh, while a light has position, orientation, color, intensity and light type. Some editors also allow additional “free-form” properties to be defined by the user on a per-instance basis.

这些属性通常以键值对的扁平列表形式实现。用户可以自由选择每个自由格式属性的名称（键）、数据类型和值。这对于设计新的游戏功能原型或实现一次性场景非常有用。

15.4.1.7 物体放置和对齐辅助工具

某些对象属性会被世界编辑器以特殊方式处理。通常，对象的位置、方向和比例可以通过正交视口和透视视口中的特殊手柄来控制，就像在 Maya 或 Max 中一样。此外，资源链接通常也需要以特殊方式处理。例如，如果我们更改与世界场景中某个对象关联的网格，编辑器应该在正交视口和 3D 透视视口中显示该网格。因此，游戏世界编辑器必须对这些属性有特殊的了解——它不能像处理大多数对象属性那样对这些属性进行通用处理。

许多世界编辑器除了基本的平移、旋转和缩放工具外，还提供了一系列对象放置和对齐辅助功能。其中许多功能大量借鉴了 Photoshop、Maya、Visio 等商业图形和 3D 建模工具的功能集。例如，对齐网格、对齐地形、对齐对象等等。

15.4.1.8 特殊对象类型

正如某些对象属性必须由世界编辑器以特殊方式处理一样，某些类型的对象也需要特殊处理。示例包括：

- 光源。由于光源没有网格，世界编辑器通常使用特殊图标来表示光源。编辑器可能会尝试显示光源对场景中几何体的近似效果，以便设计师可以实时移动光源，并大致了解场景的最终效果。

- 粒子发射器。在基于独立渲染引擎构建的编辑器中，粒子效果的可视化也可能存在问题。在这种情况下，粒子发射器可能仅使用图标显示，或者编辑器可能会尝试模拟粒子效果。当然，如果编辑器在游戏中，或者可以与正在运行的游戏进行通信以进行实时调整，则不会出现问题。

- 声源。正如我们在第14章中讨论的那样，3D渲染引擎将声源建模为3D点或体积。在世界编辑器中提供专门的编辑工具可能会很方便。例如，如果声音设计师能够将

These properties are usually implemented as a flat list of key-value pairs. The user is free to choose the name (key) of each free-form property, along with its data type and its value. This can be incredibly useful for prototyping new gameplay features or implementing one-off scenarios.

15.4.1.7 Object Placement and Alignment Aids

Some object properties are treated in a special way by the world editor. Typically the position, orientation and scale of an object can be controlled via special handles in the orthographic and perspective viewports, just like in Maya or Max. In addition, asset linkages often need to be handled in a special way. For example, if we change the mesh associated with an object in the world, the editor should display this mesh in the orthographic and 3D perspective viewports. As such, the game world editor must have special knowledge of these properties—it cannot treat them generically, as it can most object properties.

Many world editors provide a host of object placement and alignment aids in addition to the basic translation, rotation and scale tools. Many of these features borrow heavily from the feature sets of commercial graphics and 3D modeling tools like Photoshop, Maya, Visio and others. Examples include snap to grid, snap to terrain, align to object and many more.

15.4.1.8 Special Object Types

Just as some object properties must be handled in a special way by the world editor, certain types of objects also require special handling. Examples include:

- *Lights*. The world editor usually uses special icons to represent lights, since they have no mesh. The editor may attempt to display the light's approximate effect on the geometry in the scene as well, so that designers can move lights around in real time and get a reasonably good feel for how the scene will ultimately look.
- *Particle emitters*. Visualization of particle effects can also be problematic in editors that are built on a stand-alone rendering engine. In this case, particle emitters might be displayed using icons only, or some attempt might be made to emulate the particle effect in the editor. Of course, this is not a problem if the editor is in-game or can communicate with the running game for live tweaking.
- *Sound sources*. As we discussed in Chapter 14, a 3D rendering engine models sound sources as 3D points or volumes. It may be convenient to provide specialized editing tools for these in the world editor. For example, sound designers will find it helpful if they can visualize the

全向声音发射器的最大半径，或定向发射器的方向矢量和锥体。

• 区域。区域是游戏用来检测相关事件（例如物体进入或离开该区域）或为各种目的划分区域的一个空间体积。一些游戏引擎将区域限制为球体或定向盒子，而其他一些引擎可能允许区域为从上方看时呈任意凸多边形，且边严格为水平。还有一些引擎可能允许区域由更复杂的几何形状构成，例如 k-DOP（参见第 13.3.4.5 节）。如果区域始终是球形，那么设计人员可能能够使用属性网格中的“半径”属性来实现，但要定义或修改任意形状区域的范围，几乎肯定需要特殊的编辑工具。

• 样条线。样条线是由一组控制点和可能的切向量定义的三维曲线，具体取决于所使用的数学曲线类型。Catmull-Rom 样条线通常使用，因为它们完全由一组控制点（没有切线）定义，并且曲线始终经过所有控制点。但无论支持哪种类型的样条线，世界编辑器通常都需要提供在其视口中显示样条线的功能，并且用户必须能够选择和操作单个控制点。一些世界编辑器实际上支持两种选择模式 - 用于选择场景中对象的“粗略”模式和用于选择所选对象的各个组件（例如样条线的控制点或区域的顶点）的“精细”模式。

• AI 的导航网格。在许多游戏中，NPC 通过在游戏世界的可导航区域内运行寻路算法进行导航。这些可导航区域必须定义，而世界编辑器通常在 AI 设计师创建、可视化和编辑这些区域方面发挥着核心作用。例如，导航网格是一个二维三角形网格，它提供了可导航区域边界的简单描述，并为路径查找器提供连接信息。

• 其他自定义数据。当然，每个游戏都有其特定的数据需求。世界编辑器可能需要为这些数据提供自定义的可视化和编辑功能。例如，描述游戏空间内的“可供性”（窗户、门、可能的攻击或防御点），供AI系统使用；或者描述掩体点或可抓取壁架等几何特征，供玩家角色和/或NPC使用。

maximum radius of an omnidirectional sound emitter, or the direction vector and cone of a directional emitter.

- *Regions*. A region is a volume of space that is used by the game to detect relevant events such as objects entering or leaving the volume or to demark areas for various purposes. Some game engines restrict regions to being modeled as spheres or oriented boxes, while others may permit arbitrary convex polygonal shapes when viewed from above, with strictly horizontal sides. Still others might allow regions to be constructed out of more complex geometry, such as *k*-DOPs (see Section 13.3.4.5). If regions are always spherical then the designers might be able to make do with a “Radius” property in the property grid, but to define or modify the extents of an arbitrarily shaped region, a special-case editing tool is almost certainly required.
- *Splines*. A spline is a three-dimensional curve defined by a set of control points and possibly tangent vectors at the points, depending on the type of mathematical curve used. Catmull-Rom splines are commonly used because they are fully defined by a set of control points (without tangents), and the curve always passes through all of the control points. But no matter what type of splines are supported, the world editor typically needs to provide the ability to display the splines in its viewports, and the user must be able to select and manipulate individual control points. Some world editors actually support two selection modes—a “coarse” mode for selecting objects in the scene and a “fine” mode for selecting the individual components of a selected object, such as the control points of a spline or the vertices of a region.
- *Nav meshes for AI*. In many games, NPCs navigate by running pathfinding algorithms within the navigable regions of the game world. These navigable regions must be defined, and the world editor usually plays a central role in allowing AI designers to create, visualize and edit these regions. For example, a *nav mesh* is a 2D triangle mesh that provides a simple description of the boundaries of the navigable region, as well as providing connectivity information to the path finder.
- *Other custom data*. Of course, every game has its own specific data requirements. The world editor may be called upon to provide custom visualization and editing facilities for these pieces of data. Examples include a description of the “affordances” (windows, doorways, possible points of attack or defense) within a play space for use by the AI system, or geometric features that describe things like cover points or grabbable ledges for use by the player character and/or NPCs.

15.4.1.9 保存和加载世界区块

当然，如果无法加载和保存世界区块，任何世界编辑器都称不上完整。不同引擎加载和保存世界区块的粒度差异很大。有些引擎将每个世界区块存储在单个文件中，而有些引擎则允许独立加载和保存各个图层。数据格式也因引擎而异。有些引擎使用自定义二进制格式，有些则使用 XML 或 JSON 等文本格式。每种设计都有其优缺点，但每个编辑器都提供了以某种形式加载和保存世界区块的功能——每个游戏引擎都能够加载世界区块，以便在运行时进行游戏。

15.4.1.10 快速迭代

优秀的游戏世界编辑器通常支持一定程度的动态调整，以实现快速迭代。一些编辑器在游戏内部运行，允许用户立即看到更改的效果。另一些编辑器则提供从编辑器到正在运行的游戏的实时连接。还有一些世界编辑器完全离线运行，可以作为独立工具或作为 Lightwave 或 Maya 等 DCC 应用程序的插件。这些工具有时允许将修改后的数据动态地重新加载到正在运行的游戏中。具体机制并不重要——重要的是用户拥有相当短的往返迭代时间（即从更改游戏世界到在游戏中看到更改效果的时间）。重要的是要意识到迭代不□□必是即时的。迭代时间应该与所做更改的范围和频率相称。例如，我们可能期望调整角色的最大生命值是一个非常快的操作，但如果要对整个世界块的光照环境进行重大更改，则可能需要更长的迭代时间。

15.4.2 综合资产管理工具

在某些引擎中，游戏世界编辑器与游戏资源数据库管理的其他方面集成在一起，例如定义网格和材质属性、定义动画、混合树、动画状态机、设置对象的碰撞和物理属性、管理纹理资源等等。（有关游戏资源数据库的讨论，请参见第 7.2.1.2 节。）这种设计最著名的例子或许是 UnrealEd，它是用于创建基于虚幻引擎的游戏内容的编辑器。UnrealEd 直接集成到游戏引擎中，因此在编辑器中所做的任何更改都会直接影响正在运行的游戏中动态元素。这使得快速迭代变得非常容易。但 UnrealEd 不仅仅是一款游戏。

15.4.1.9 Saving and Loading World Chunks

Of course, no world editor would be complete if it were unable to load and save world chunks. The granularity with which world chunks can be loaded and saved differs widely from engine to engine. Some engines store each world chunk in a single file, while others allow individual layers to be loaded and saved independently. Data formats also vary across engines. Some use custom binary formats, others text formats like XML or JSON. Each design has its pros and cons, but every editor provides the ability to load and save world chunks in some form—and every game engine is capable of loading world chunks so that they can be played at runtime.

15.4.1.10 Rapid Iteration

A good game world editor usually supports some degree of dynamic tweaking for rapid iteration. Some editors run within the game itself, allowing the user to see the effects of his or her changes immediately. Others provide a live connection from the editor to the running game. Still other world editors operate entirely offline, either as a stand-alone tool or as a plug-in to a DCC application like Lightwave or Maya. These tools sometimes permit modified data to be reloaded dynamically into the running game. The specific mechanism isn't important—all that matters is that users have a reasonably short *round-trip iteration time* (i.e., the time between making a change to the game world and seeing the effects of that change in-game). It's important to realize that iterations don't have to be instantaneous. Iteration times should be commensurate with the scope and frequency of the changes being made. For example, we might expect tweaking a character's maximum health to be a very fast operation, but when making major changes to the lighting environment for an entire world chunk, a much longer iteration time might be acceptable.

15.4.2 Integrated Asset Management Tools

In some engines, the game world editor is integrated with other aspects of game asset database management, such as defining mesh and material properties, defining animations, blend trees, animation state machines, setting up collision and physical properties of objects, managing texture resources and so on. (See Section 7.2.1.2 for a discussion of the game asset database.)

Perhaps the best-known example of this design in action is *UnrealEd*, the editor used to create content for games built on the Unreal Engine. UnrealEd is integrated directly into the game engine, so any changes made in the editor are made directly to the dynamic elements in the running game. This makes rapid iteration very easy to achieve. But UnrealEd is much more than a game

图 15.8.UnrealEd 的通用浏览器提供对整个游戏资产数据库的访问。

世界编辑器——它实际上是一个完整的内容创建包。它管理着整个游戏资源数据库，从动画、音频剪辑、三角形网格、纹理、材质、着色器等等。虚幻编辑器为用户提供了统一、实时、所见即所得的整个资源数据库视图，使其成为任何快速、高效的游戏开发流程的强大助力。图 15.8 和 15.9 展示了虚幻编辑器的一些屏幕截图。

15.4.2.1 数据处理成本

在 7.2.1 节中，我们了解到资产调节管道 (ACP) 将游戏资产从其各种源格式转换为游戏引擎所需的格式。这通常包含两个步骤。首先，资产从 DCC 应用程序导出为平台无关的中间格式，该格式仅包含与游戏相关的数据。其次，资产被处理为针对特定平台优化的格式。在一个面向多游戏平台的项目中，单个平台无关的资产会在第二阶段生成多个平台特定资产。

不同工具管道之间的一个关键区别在于执行第二个平台特定优化步骤的时机。UnrealEd 在资源首次导入编辑器时执行此步骤。这种方法在

Figure 15.8. UnrealEd's Generic Browser provides access to the entire game asset database.

world editor—it is actually a complete content-creation package. It manages the entire database of game assets, from animations to audio clips to triangle meshes to textures to materials and shaders and much more. UnrealEd provides its user with a unified, real-time, WYSIWYG view into the entire asset database, making it a powerful enabler of any rapid, efficient game development process. A few screenshots from UnrealEd are shown in Figures 15.8 and 15.9.

15.4.2.1 Data Processing Costs

In Section 7.2.1, we learned that the asset conditioning pipeline (ACP) converts game assets from their various source formats into the formats required by the game engine. This is typically a two-step process. First, the asset is exported from the DCC application to a platform-independent intermediate format that only contains the data that is relevant to the game. Second, the asset is processed into a format that is optimized for a specific platform. On a project targeting multiple gaming platforms, a single platform-independent asset gives rise to multiple platform-specific assets during this second phase.

One of the key differences between tools pipelines is the point at which this second platform-specific optimization step is performed. UnrealEd performs it when assets are first imported into the editor. This approach pays off in

图 15.9。UnrealEd 还提供了一个世界编辑器。

在关卡设计迭代中，迭代速度更快。然而，这会增加更改网格、动画、音频等源资源的成本。其他引擎，例如 Source 引擎和 Quake 引擎，在游戏运行前烘焙关卡时会支付资源优化成本。Halo 允许用户随时更改原始资源；这些资源在首次加载到引擎时会被转换为优化形式，并且结果会被缓存，以避免每次游戏运行时都执行不必要的优化步骤。

Figure 15.9. UnrealEd also provides a world editor.

rapid iteration time when iterating on level design. However, it can make the cost of changing source assets like meshes, animations, audio assets and so on more painful. Other engines like the Source engine and the *Quake* engine pay the asset optimization cost when baking out the level prior to running the game. *Halo* gives the user the option to change raw assets at any time; they are converted into an optimized form when they are first loaded into the engine, and the results are cached to prevent the optimization step from being performed needlessly every time the game is run.

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

16 运行时游戏基础系统

16.1 游戏基础系统的组成部分

大多数游戏引擎都提供一套运行时软件组件，这些组件共同构成一个框架，游戏的独特规则、目标和动态世界元素都可在此框架上构建。游戏行业中，这些组件没有统一的标准名称，但我们将它们统称为引擎的玩法基础系统。如果可以合理地在引擎和游戏之间划一条界线，即游戏引擎和游戏本身，那么这些系统就位于这条界线的下方。理论上，我们可以构建与游戏无关的玩法基础系统。然而，在实践中，这些系统几乎总是包含特定于游戏类型或游戏的细节。事实上，引擎和游戏之间的界线或许可以被形象地想象成一个巨大的模糊——一条贯穿这些组件的渐变弧线，将引擎与游戏连接起来。在某些游戏引擎中，我们甚至可以将玩法基础系统完全视为引擎-游戏界线之上。游戏引擎之间的差异在其玩法组件的设计和实现方面最为明显。话虽如此，不同引擎之间存在着数量惊人的共同模式，这些共同点将成为我们在此讨论的主题。

16 Runtime Gameplay Foundation Systems

16.1 Components of the Gameplay Foundation System

Most game engines provide a suite of runtime software components that together provide a framework upon which a game's unique rules, objectives and dynamic world elements can be constructed. There is no standard name for these components within the game industry, but we will refer to them collectively as the engine's *gameplay foundation system*. If a line between engine and game can reasonably be drawn between the game engine and the game itself, then these systems lie just beneath this line. In theory, one can construct gameplay foundation systems that are for the most part game-agnostic. However, in practice, these systems almost always contain genre- or game-specific details. In fact, the line between the engine and the game can probably be best visualized as one big blur—a gradient that arcs across these components as it links the engine to the game. In some game engines, one might even go so far as to consider the gameplay foundation systems as lying entirely *above* the engine-game line. The differences between game engines are most acute when it comes to the design and implementation of their gameplay components. That said, there are a surprising number of common patterns across engines, and those commonalities will be the topic of our discussions here.

每个游戏引擎处理游戏性软件设计问题的方式略有不同。然而，大多数引擎都以某种形式提供了以下主要子系统：

• 运行时游戏对象模型。这是通过世界编辑器向游戏设计师宣传的抽象游戏对象模型的实现。

• 关卡管理和流式传输。该系统负责加载和卸载游戏发生的虚拟世界的内容。在许多引擎中，关卡数据会在游戏过程中被流式传输到内存中，从而营造出一个巨大无缝世界的幻觉（但实际上它被分解成多个离散的区块）。

• 实时对象模型更新。为了使游戏世界中的游戏对象能够自主运行，每个对象必须定期更新。这正是游戏引擎中所有独立系统真正整合成一个紧密整体的关键所在。

• 消息传递和事件处理。大多数游戏对象需要相互通信。这通常通过抽象的消息传递系统完成。对象间的消息通常表示游戏世界状态的变化，称为事件。因此，在许多工作室中，消息传递系统被称为事件系统。

• 脚本。使用 C 或 C++ 等语言编写高级游戏逻辑可能非常繁琐。为了提高生产力、实现快速迭代，并赋予团队中非程序员更多权力，游戏引擎中通常会集成一种脚本语言。这种语言可能是基于文本的，例如 Python 或 Lua，也可能是图形化语言，例如虚幻引擎的蓝图。

• 目标和游戏流程管理。该子系统管理玩家的目标和游戏的整体流程。这通常用玩家目标的序列、树状图或广义图表来描述。目标通常被分为几个章节，尤其是在像许多现代游戏一样高度依赖故事驱动的游戏的情况下。游戏流程管理系统管理游戏的整体流程，跟踪玩家的目标完成情况，并在目标完成时引导玩家从游戏世界的一个区域进入下一个区域。一些设计师将其称为游戏的“脊柱”。

在这些主要系统中，运行时对象模型可能是最复杂的。它通常提供以下大部分（如果不是全部）功能：

• 动态生成和销毁游戏对象。游戏世界中的动态元素通常需要在游戏过程中出现和消失。健康

Every game engine approaches the problem of gameplay software design a bit differently. However, most engines provide the following major subsystems in some form:

- *Runtime game object model*. This is an implementation of the abstract game object model advertised to the game designers via the world editor.
- *Level management and streaming*. This system loads and unloads the contents of the virtual worlds in which gameplay takes place. In many engines, level data is streamed into memory during gameplay, thus providing the illusion of a large seamless world (when in fact it is broken into discrete chunks).
- *Real-time object model updating*. In order to permit the game objects in the world to behave autonomously, each object must be updated periodically. This is where all of the disparate systems in a game engine truly come together into a cohesive whole.
- *Messaging and event handling*. Most game objects need to communicate with one another. This is usually done via an abstract messaging system. Inter-object messages often signal changes in the state of the game world called *events*. So the messaging system is referred to as the *event system* in many studios.
- *Scripting*. Programming high-level game logic in a language like C or C++ can be cumbersome. To improve productivity, allow rapid iteration, and put more power into the hands of the non-programmers on the team, a scripting language is often integrated into the game engine. This language might be text-based, like Python or Lua, or it might be a graphical language, like Unreal's Blueprints.
- *Objectives and game flow management*. This subsystem manages the player's objectives and the overall flow of the game. This is usually described by a sequence, a tree, or a generalized graph of player objectives. Objectives are often grouped into chapters, especially if the game is highly story-driven as many modern games are. The game flow management system manages the overall flow of the game, tracks the player's accomplishment of objectives and gates the player from one area of the game world to the next as the objectives are accomplished. Some designers refer to this as the "spine" of the game.

Of these major systems, the runtime object model is probably the most complex. It typically provides most, if not all, of the following features:

- *Spawning and destroying game objects dynamically*. The dynamic elements in a game world often need to come and go during gameplay. Health

拾取物品包后，它们会消失；爆炸出现后又消散；当你以为自己已经通关时，敌人的增援部队会神秘地从角落出现。许多游戏引擎提供了一个系统来管理与动态生成的游戏对象相关的内存和其他资源。而其他引擎则完全不允许动态创建或销毁游戏对象。

• 与底层引擎系统的连接。每个游戏对象都与一个或多个底层引擎系统存在某种连接。大多数游戏对象在视觉上由可渲染的三角形网格表示。有些对象具有粒子效果。许多对象会产生声音。有些对象会生成动画。许多对象具有碰撞，有些对象由物理引擎动态模拟。游戏基础系统的主要职责之一是确保每个游戏对象都能访问其所依赖的引擎系统的服务。

• 实时模拟对象行为。游戏引擎的核心是基于代理模型的实时动态计算机模拟。这只是一种更贴切的说法，即游戏引擎需要随时间动态更新所有游戏对象的状态。对象可能需要按照特定的顺序进行更新，这部分取决于对象之间的依赖关系，部分取决于它们对各个引擎子系统的依赖关系，部分取决于这些引擎子系统之间的相互依赖关系。

• 能够定义新的游戏对象类型。每个游戏的需求都会随着游戏的开发而变化和发展。游戏对象模型必须足够灵活，以便轻松添加新的对象类型并将其暴露给世界编辑器。理想情况下，应该能够以完全数据驱动的方式定义新的对象类型。然而，在许多引擎中，添加新的游戏对象类型需要程序员的帮助。

• 唯一的对象 ID。典型的游戏世界包含数百甚至数千个不同类型的独立游戏对象。在运行时，能够识别或搜索特定对象非常重要。这意味着每个对象都需要某种唯一的标识符。人类可读的名称是最方便的 ID，但我们必须警惕在运行时使用字符串的性能成本。整数 ID 是最有效的选择，但对于人类游戏开发者来说，使用它们非常困难。可以说，最好的解决方案是使用散列字符串 ID（参见第 6.4.3.1 节）作为我们的对象标识符，因为它们与整数一样高效

packs disappear once they have been picked up, explosions appear and then dissipate and enemy reinforcements mysteriously come from around a corner just when you think you've cleared the level. Many game engines provide a system for managing the memory and other resources associated with dynamically spawned game objects. Other engines simply disallow dynamic creation or destruction of game objects altogether.

- *Linkage to low-level engine systems*. Every game object has some kind of linkage to one or more underlying engine systems. Most game objects are visually represented by renderable triangle meshes. Some have particle effects. Many generate sounds. Some animate. Many have collision, and some are dynamically simulated by the physics engine. One of the primary responsibilities of the gameplay foundation system is to ensure that every game object has access to the services of the engine systems upon which it depends.
- *Real-time simulation of object behaviors*. At its core, a game engine is a real-time dynamic computer simulation of an agent-based model. This is just a fancy way of saying that the game engine needs to update the states of all the game objects dynamically over time. The objects may need to be updated in a very particular order, dictated in part by dependencies between the objects, in part by their dependencies on various engine subsystems, and in part because of the interdependencies between those engine subsystems themselves.
- *Ability to define new game object types*. Every game's requirements change and evolve as the game is developed. It's important that the game object model be flexible enough to permit new object types to be added easily and exposed to the world editor. In an ideal world, it should be possible to define a new type of object in an entirely data-driven manner. However, in many engines, the services of a programmer are required in order to add new game object types.
- *Unique object ids*. Typical game worlds contain hundreds or even thousands of individual game objects of various types. At runtime, it's important to be able to identify or search for a particular object. This means each object needs some kind of unique identifier. A human-readable name is the most convenient kind of id, but we must be wary of the performance costs of using strings at runtime. Integer ids are the most efficient choice, but they are very difficult for human game developers to work with. Arguably the best solution is to use hashed string ids (see Section 6.4.3.1) as our object identifiers, as they are as efficient as integers

但可以转换回字符串形式以便于阅读。

• 游戏对象查询。游戏基础系统必须提供一些在游戏世界中查找对象的方法。我们可能希望通过唯一 ID 查找特定对象，或者查找特定类型的所有对象，又或者希望基于任意条件执行高级查询（例如，查找玩家角色 20 米半径范围内的所有敌人）。

• 游戏对象引用。找到对象后，我们需要某种机制来保存对它们的引用，可以是在单个函数内短暂保存，也可以是长时间保存。对象引用可以像指向 C++ 类实例的指针一样简单，也可以是更复杂的引用，例如句柄或引用计数智能指针。

• 有限状态机支持。许多类型的游戏对象最好用有限状态机 (FSM) 建模。一些游戏引擎允许游戏对象处于多种可能状态之一，每种状态都有其自身的属性和行为特征。

• 网络复制。在联网多人游戏中，多台游戏机通过局域网或互联网连接在一起。特定游戏对象的状态通常由一台机器拥有和管理。然而，该对象的状态也必须复制（传达）到参与多人游戏的其他机器，以便所有玩家都能获得一致的对象视图。

• 保存和加载/对象持久性。许多游戏引擎允许将世界中游戏对象的当前状态保存到磁盘并在以后重新加载。这可能是为了支持"随处保存"的游戏保存系统或作为实现网络复制的一种方式，或者它可能只是加载在世界编辑器工具中创作的游戏世界块的主要方式。对象持久性通常需要某些语言功能，例如运行时类型识别（RTTI），反射和抽象构造。RTTI 和反射为软件提供了一种在运行时动态确定对象类型及其类提供的属性和方法的方法。抽象构造允许创建类的实例，而不必对类的名称进行硬编码 - 在将对象实例从磁盘序列化到内存时，这是一个非常有用的功能。如果您选择的语言本身不支持 RTTI，反射和抽象构造，则可以手动添加这些功能。

我们将用本章的剩余部分深入研究每一个子系统。

but can be converted back into string form for ease of reading.

- *Game object queries*. The gameplay foundation system must provide some means of finding objects within the game world. We might want to find a specific object by its unique id, or all the objects of a particular type, or we might want to perform advanced queries based on arbitrary criteria (e.g., find all enemies within a 20 m radius of the player character).
- *Game object references*. Once we've found the objects, we need some mechanism for holding *references* to them, either briefly within a single function or for much longer periods of time. An object reference might be as simple as a pointer to a C++ class instance, or it might be something more sophisticated, like a handle or a reference-counted smart pointer.
- *Finite state machine support*. Many types of game objects are best modeled as finite state machines (FSM). Some game engines provide the ability for a game object to exist in one of many possible states, each with its own attributes and behavioral characteristics.
- *Network replication*. In a networked multiplayer game, multiple game machines are connected together via a LAN or the Internet. The state of a particular game object is usually owned and managed by one machine. However, that object's state must also be *replicated* (communicated) to the other machines involved in the multiplayer game so that all players have a consistent view of the object.
- *Saving and loading / object persistence*. Many game engines allow the current states of the game objects in the world to be saved to disk and later reloaded. This might be done to support a "save anywhere" save-game system or as a way of implementing network replication, or it might simply be the primary means of loading game world chunks that were authored in the world editor tool. Object persistence usually requires certain language features, such as *runtime type identification* (RTTI), *reflection* and *abstract construction*. RTTI and reflection provide software with a means of determining an object's *type*, and what *attributes* and *methods* its class provides, dynamically at runtime. Abstract construction allows instances of a class to be created without having to hard-code the name of the class—a very useful feature when serializing an object instance into memory from disk. If RTTI, reflection and abstract construction are not natively supported in your language of choice, these features can be added manually.

We'll spend the remainder of this chapter delving into each of these subsystems in depth.

16.2 运行时对象模型架构

在世界编辑器中，游戏设计师会看到一个抽象的游戏对象模型，该模型定义了游戏中可以存在的各种动态元素、它们的行为方式以及它们所具有的属性。在运行时，游戏基础系统必须提供该对象模型的具体实现。这是迄今为止所有游戏基础系统中最大的组件。

运行时对象模型的实现可能与抽象工具端对象模型有任何相似之处，也可能完全不同。例如，它可能根本不是用面向对象编程语言实现的，或者它可能使用一组相互连接的类实例来表示单个抽象游戏对象。无论其设计如何，运行时对象模型都必须忠实地再现世界编辑器所宣传的对象类型、属性和行为。

运行时对象模型是世界编辑器中呈现给设计师的抽象工具端对象模型在游戏中的体现。设计千差万别，但大多数游戏引擎都遵循以下两种基本架构风格之一：

- 以对象为中心。在这种风格中，每个工具端游戏对象在运行时都由单个类实例或一组相互关联的实例表示。每个对象都有一组属性和行为，这些属性和行为封装在该对象所属的类（或多个类）中。

  游戏世界只是游戏对象的集合。

- 以属性为中心。在这种风格中，每个工具端游戏对象仅由一个唯一的 ID 表示（实现为整数、散列字符串 ID 或字符串）。每个游戏对象的属性分布在多个数据表中，每个属性类型一个，并由对象 ID 作为键（而不是集中在单个类实例或互连实例集合中）。属性本身通常实现为硬编码类的实例。游戏对象的行为由组成它的属性集合隐式定义。例如，如果一个对象具有“Health”属性，那么它可能受到伤害、失去健康并最终死亡。如果一个对象具有“MeshInstance”属性，那么它可以作为三角形网格的实例在 3D 中渲染。

每种架构风格都有其独特的优缺点。我们将详细探讨每种风格，并指出其中一种风格相对于另一种风格有哪些显著的潜在优势。

16.2 Runtime Object Model Architectures

In the world editor, the game designer is presented with an abstract game object model, which defines the various types of dynamic elements that can exist in the game, how they behave and what kinds of attributes they have. At runtime, the gameplay foundation system must provide a concrete implementation of this object model. This is by far the largest component of any gameplay foundation system.

The runtime object model implementation may or may not bear any resemblance to the abstract tool-side object model. For example, it might not be implemented in an object-oriented programming language at all, or it might use a collection of interconnected class instances to represent a single abstract game object. Whatever its design, the runtime object model must provide a faithful reproduction of the object types, attributes and behaviors advertised by the world editor.

The runtime object model is the in-game manifestation of the abstract tool-side object model presented to the designers in the world editor. Designs vary widely, but most game engines follow one of two basic architectural styles:

- *Object-centric*. In this style, each tool-side game object is represented at runtime by a single class instance or a small collection of interconnected instances. Each object has a set of *attributes* and *behaviors* that are encapsulated within the class (or classes) of which the object is an instance. The game world is just a collection of game objects.
- *Property-centric*. In this style, each tool-side game object is represented only by a unique id (implemented as an integer, hashed string id or string). The *properties* of each game object are distributed across many data tables, one per property type, and keyed by object id (rather than being centralized within a single class instance or collection of interconnected instances). The properties themselves are often implemented as instances of hard-coded classes. The *behavior* of a game object is implicitly defined by the collection of properties of which it is composed. For example, if an object has the “Health” property, then it can be damaged, lose health and eventually die. If an object has the “MeshInstance” property, then it can be rendered in 3D as an instance of a triangle mesh.

There are distinct advantages and disadvantages to each of these architectural styles. We'll investigate each one in some detail and note where one style has significant potential benefits over the other as they arise.

16.2.1 以对象为中心的架构

在以对象为中心的游戏世界对象架构中，每个逻辑游戏对象都被实现为一个类的实例，或者可能是一组相互关联的类实例。在这个广泛的框架下，存在许多不同的设计方案。我们将在以下章节中探讨一些最常见的设计方案。

16.2.1.1 一个简单的 C 语言基于对象模型：Hydro Thunder

游戏对象模型完全不需要用 C++ 这样的面向对象语言来实现。例如，由圣地亚哥 Midway Home Entertainment 开发的热门街机游戏 Hydro Thunder 完全是用 C 语言编写的。Hydro 采用了非常简单的游戏对象模型，仅包含以下几种对象类型：

- 船只（玩家和 AI 控制），
- 浮动的蓝色和红色增强图标，
- 环境动画对象（轨道边的动物等），
- 水面，
- 坡道，
- 瀑布，
- 粒子效果，
- 赛道区域（相互连接的二维多边形区域，共同定义船只可以比赛的水域区域），
- 静态几何（地形、树叶、轨道两侧的建筑物等），以及
- 二维平视显示器 (HUD) 元素。

图 16.1 展示了 Hydro Thunder 的一些截图。请注意两张截图中悬浮的助推图标，以及左图中游过的鲨鱼（环境动画对象的一个□□例子）。

Hydro 有一个名为 World_t 的 C 结构体，用于存储和管理游戏世界（例如一条赛道）的内容。该世界包含指向各种游戏对象数组的指针。静态几何体是一个网格实例。水面、瀑布和粒子效果分别由自定义数据结构表示。游戏中的船只、加速图标和其他动态对象由一个名为 WorldOb_t 的通用结构体实例（即世界对象）表示。这相当于我们在本章中定义的 Hydro 游戏对象。

16.2.1 Object-Centric Architectures

In an *object-centric* game world object architecture, each logical game object is implemented as an instance of a class, or possibly a collection of interconnected class instances. Under this broad umbrella, many different designs are possible. We'll investigate a few of the most common designs in the following sections.

16.2.1.1 A Simple Object-Based Model in C: Hydro Thunder

Game object models needn't be implemented in an object-oriented language like C++ at all. For example, the arcade hit *Hydro Thunder*, by Midway Home Entertainment in San Diego, was written entirely in C. *Hydro* employed a very simple game object model consisting of only a few object types:

- boats (player- and AI-controlled),
- floating blue and red boost icons,
- ambient animated objects (animals on the side of the track, etc.),
- the water surface,
- ramps,
- waterfalls,
- particle effects,
- race track sectors (two-dimensional polygonal regions connected to one another that together define the watery region in which boats could race),
- static geometry (terrain, foliage, buildings along the sides of the track, etc.), and
- two-dimensional heads-up display (HUD) elements.

A few screenshots of *Hydro Thunder* are shown in Figure 16.1. Notice the hovering boost icons in both screenshots and the shark swimming by in the left image (an example of an ambient animated object).

Hydro had a C struct named `World_t` that stored and managed the contents of a game world (i.e., a single race track). The world contained pointers to arrays of various kinds of game objects. The static geometry was a single mesh instance. The water surface, waterfalls and particle effects were each represented by custom data structures. The boats, boost icons and other dynamic objects in the game were represented by instances of a general-purpose struct called `WorldOb_t` (i.e., a world object). This was *Hydro*'s equivalent of a *game object* as we've defined it in this chapter.

图 16.1. 圣地亚哥 Midway Home Entertainment 开发的街机游戏 Hydro Thunder 的截图。

WorldOb_t 数据结构包含数据成员，这些数据成员编码了对象的位置和方向、用于渲染它的 3D 网格、一组碰撞球、简单的动画状态信息（Hydro 仅支持刚性分层动画）、速度、质量和浮力等物理属性以及游戏中所有动态对象共有的其他数据。此外，每个 WorldOb_t 包含三个指针：一个 void*“用户数据”指针、一个指向自定义“更新”函数的指针和一个指向自定义“绘制”函数的指针。因此，虽然 Hydro Thunder 在最严格的意义上不是面向对象的，但 Hydro 引擎确实扩展了其非面向对象语言（C）以支持两个重要的 OOP 特性的基本实现：继承和多态性。用户数据指针允许每种类型的游戏对象维护特定于其类型的自定义状态信息，同时继承所有世界对象的共同特性。例如，Banshee 船的助推机制与 Rad Hazard 不同，并且每个助推机制都需要不同的状态信息来管理其展开和收起动画。这两个函数指针的作用类似于虚函数，允许世界对象具有多态行为（通过其“更新”函数）和多态视觉外观（通过其“绘制”函数）。

```
struct WorldOb_s
{
    Orient_t m_transform;    /* position/rotation */
    Mesh3d*  m_pMesh;        /* 3D mesh */
    /* ... */
    void*    m_pUserData;    /* custom state */

    void     (*m_pUpdate)(); /* polymorphic update */
    void     (*m_pDraw)();   /* polymorphic draw */
};
typedef struct WorldOb_s WorldOb_t;
```

Figure 16.1. Screenshots from the arcade game *Hydro Thunder*, developed by Midway Home Entertainment in San Diego.

The `WorldOb_t` data structure contained data members encoding the position and orientation of the object, the 3D mesh used to render it, a set of collision spheres, simple animation state information (*Hydro* only supported rigid hierarchical animation), physical properties like velocity, mass and buoyancy, and other data common to all of the dynamic objects in the game. In addition, each `WorldOb_t` contained three pointers: a `void*` “user data” pointer, a pointer to a custom “update” function and a pointer to a custom “draw” function. So while *Hydro Thunder* was not object-oriented in the strictest sense, the *Hydro* engine did extend its non-object-oriented language (C) to support rudimentary implementations of two important OOP features: *inheritance* and *polymorphism*. The user data pointer permitted each type of game object to maintain custom state information specific to its type while inheriting the features common to all world objects. For example, the *Banshee* boat had a different booster mechanism than the *Rad Hazard*, and each booster mechanism required different state information to manage its deployment and stowing animations. The two function pointers acted like *virtual functions*, allowing world objects to have polymorphic behaviors (via their “update” functions) and polymorphic visual appearances (via their “draw” functions).

```
struct WorldOb_s
{
    Orient_t m_transform;    /* position/rotation */
    Mesh3d*  m_pMesh;        /* 3D mesh */
    /* ... */
    void*    m_pUserData;    /* custom state */

    void     (*m_pUpdate)(); /* polymorphic update */
    void     (*m_pDraw)();   /* polymorphic draw */
};
typedef struct WorldOb_s WorldOb_t;
```

图 16.2。游戏 Pac-Man 的假设类层次结构。

16.2.1.2 单体类层次结构

人们很自然地会想要对游戏对象类型进行分类。这往□□往会引导游戏程序员选择支持继承的面向对象语言。类层次结构是表示一系列相互关联的游戏对象类型集合的最直观、最直接的方式。因此，大多数商业游戏引擎都采用基于类层次结构的技术也就不足为奇了。

图 16.2 展示了一个简单的类层次结构，可用于实现游戏 Pac-Man。此层次结构（与许多层次结构一样）植根于一个名为 GameObject 的通用类，该类可能提供所有对象类型所需的一些功能，例如 RTTI 或序列化。MovableObject 类表示任何具有位置和方向的对象。RenderableObject 类使对象能够被渲染（在传统的 Pac-Man 游戏中，通过精灵图渲染；在现代的 3D Pac-Man 游戏中，可能通过三角形网格渲染）。从 RenderableObject 派生出构成游戏的幽灵、Pac-Man、小球和能量药丸等类。这只是一个假设的例子，但它阐明了大多数游戏对象类层次结构的基本思想——即通用的通用功能往往存在于层次结构的根部，而朝向层次结构叶子的类则倾向于添加越来越具体的功能。

游戏对象类层次结构通常一开始很小且简单，这种形式可以成为描述游戏对象类型集合的一种强大而直观的方式。然而，随着类层次结构的增长，它们往往会同时加深和扩展，从而导致我所说的“单片类层次结构”。当游戏对象模型中几乎所有类都继承自一个公共基类时，就会出现这种层次结构。虚幻引擎的游戏对象模型就是一个典型的例子，如图 16.3 所示。

Figure 16.2. A hypothetical class hierarchy for the game *Pac-Man*.

16.2.1.2 Monolithic Class Hierarchies

It's natural to want to classify game object types taxonomically. This tends to lead game programmers toward an object-oriented language that supports inheritance. A class hierarchy is the most intuitive and straightforward way to represent a collection of interrelated game object types. So it is not surprising that the majority of commercial game engines employ a class hierarchy based technique.

Figure 16.2 shows a simple class hierarchy that could be used to implement the game *Pac-Man*. This hierarchy is rooted (as many are) at a common class called `GameObject`, which might provide some facilities needed by all object types, such as RTTI or serialization. The `MovableObject` class represents any object that has a position and orientation. `RenderableObject` gives the object an ability to be rendered (in the case of traditional *Pac-Man*, via a sprite, or in the case of a modern 3D *Pac-Man* game, perhaps via a triangle mesh). From `RenderableObject` are derived classes for the ghosts, Pac-Man, pellets and power pills that make up the game. This is just a hypothetical example, but it illustrates the basic ideas that underlie most game object class hierarchies—namely that common, generic functionality tends to exist at the root of the hierarchy, while classes toward the leaves of the hierarchy tend to add increasingly specific functionality.

A game object class hierarchy usually begins small and simple, and in that form, it can be a powerful and intuitive way to describe a collection of game object types. However, as class hierarchies grow, they have a tendency to deepen and widen simultaneously, leading to what I call a *monolithic class hierarchy*. This kind of hierarchy arises when virtually all classes in the game object model inherit from a single, common base class. The Unreal Engine's game object model is a classic example, as Figure 16.3 illustrates.

图 16.3.虚幻引擎中游戏对象类层次结构的摘录。

16.2.1.3 深层、宽层级结构的问题

庞大的类层次结构往往会给游戏开发团队带来各种各样的问题。类层次结构越深、越宽，这些问题就越严重。在接下来的章节中，我们将探讨由宽而深的类层次结构引起的一些最常见的问题。

理解、维护和修改类

类在类层次结构中的位置越深，理解、维护和修改就越困难。这是因为要理解一个类，你实际上也需要理解它的所有父类。例如，修改

Figure 16.3. An excerpt from the game object class hierarchy in the *Unreal* engine.

16.2.1.3 Problems with Deep, Wide Hierarchies

Monolithic class hierarchies tend to cause problems for the game development team for a wide range of reasons. The deeper and wider a class hierarchy grows, the more extreme these problems can become. In the following sections, we'll explore some of the most common problems caused by wide, deep class hierarchies.

Understanding, Maintaining and Modifying Classes

The deeper a class lies within a class hierarchy, the harder it is to understand, maintain and modify. This is because to understand a class, you really need to understand all of its parent classes as well. For example, modifying the

派生类中看似无害的虚函数的行为可能会违反众多基类中任何一个所做的假设，从而导致难以发现的细微错误。

无法描述多维分类法

层级结构本质上是根据一套特定的标准体系（称为分类学）对对象进行分类。例如，生物分类学（也称为 alpha 分类学）根据基因相似性对所有生物进行分类，使用一棵包含八个级别的树：界、界、门、纲、目、科、属和种。在树的每个级别，都使用不同的标准将地球上无数的生命形式划分成越来越精细的组别。

任何层次结构的最大问题之一是，它只能沿着单一“轴”（根据一组特定的标准）在树的每一层级对对象进行分类。一旦为特定层次结构选定了标准，就很难甚至不可能沿着一组完全不同的“轴”进行分类。例如，生物分类学根据遗传特征对对象进行分类，但它并未提及生物体的颜色。为了按颜色对生物体进行分类，我们需要一个完全不同的树状结构。

在面向对象编程中，层次化分类的这种局限性通常表现为类层次结构过于宽泛、深度过深且令人困惑。分析真实游戏的类层次结构时，我们常常会发现，其结构试图将多种不同的分类标准融合到一棵类树中。在其他情况下，为了适应一种在最初设计层次结构时未曾预料到的新对象类型，类层次结构会做出一些调整。例如，想象一下图 16.4 中描述的看似合乎逻辑的描述不同类型车辆的类层次结构。

图 16.4. 描述各种车辆的看似合乎逻辑的类别层次结构。

behavior of an innocuous-looking virtual function in a derived class could violate the assumptions made by any one of the many base classes, leading to subtle, difficult-to-find bugs.

Inability to Describe Multidimensional Taxonomies

A hierarchy inherently classifies objects according to a particular system of criteria known as a *taxonomy*. For example, *biological taxonomy* (also known as *alpha taxonomy*) classifies all living things according to genetic similarities, using a tree with eight levels: domain, kingdom, phylum, class, order, family, genus and species. At each level of the tree, a different criterion is used to divide the myriad life forms on our planet into more and more refined groups.

One of the biggest problems with any hierarchy is that it can only classify objects along a single “axis”—according to one particular set of criteria—at each level of the tree. Once the criteria have been chosen for a particular hierarchy, it becomes difficult or impossible to classify along an entirely different set of “axes.” For example, biological taxonomy classifies objects according to genetic traits, but it says nothing about the colors of the organisms. In order to classify organisms by color, we’d need an entirely different tree structure.

In object-oriented programming, this limitation of hierarchical classification often manifests itself in the form of wide, deep and confusing class hierarchies. When one analyzes a real game’s class hierarchy, one often finds that its structure attempts to meld a number of different classification criteria into a single class tree. In other cases, concessions are made in the class hierarchy to accommodate a new type of object whose characteristics were not anticipated when the hierarchy was first designed. For example, imagine the seemingly logical class hierarchy describing different types of vehicles, depicted in Figure 16.4.

Figure 16.4. A seemingly logical class hierarchy describing various kinds of vehicles.

如果游戏设计师向程序员宣布他们现在想在游戏中加入一款两栖车辆，会发生什么情况？这种车辆不符合现有的分类体系。这可能会让程序员感到恐慌，或者更有可能的是，以各种丑陋且容易出错的方式“破解”他们的类层次结构。

多重继承：致命钻石

解决两栖车辆问题的一个方案是利用 C++ 的多重继承 (MI) 特性，如图 16.5 所示。乍一看，这似乎是一个不错的解决方案。然而，C++ 中的多重继承会带来许多实际问题。例如，多重继承可能导致一个对象包含其基类成员的多个副本——这种情况被称为“致命钻石”或“死亡钻石”。（更多详情，请参见第 3.1.1.3 节。）

构建一个有效、易于理解和维护的多重继承 (MI) 类层次结构的难度通常大于其带来的好处。因此，大多数游戏工作室禁止或严格限制在其类层次结构中使用多重继承。

图 16.5。两栖车辆的菱形类别层次结构。

混合类

有些团队确实允许有限形式的混合继承 (MI)，即一个类可以有任意数量的父类，但只有一个祖父类。换句话说，一个类只能从主继承层次结构中的一个类继承，但它也可以从任意数量的混合类（没有基类的独立类）继承。这允许将通用功能分解成一个混合类，然后在需要的地方将其修补到主继承层次结构中。如图 16.6 所示。然而，正如我们将在下面看到的，组合或聚合这些类通常比从它们继承更好。

What happens when the game designers announce to the programmers that they now want the game to include an *amphibious* vehicle? Such a vehicle does not fit into the existing taxonomic system. This may cause the programmers to panic or, more likely, to “hack” their class hierarchy in various ugly and error-prone ways.

Multiple Inheritance: The Deadly Diamond

One solution to the amphibious vehicle problem is to utilize C++’s multiple inheritance (MI) features, as shown in Figure 16.5. At first glance, this seems like a good solution. However, multiple inheritance in C++ poses a number of practical problems. For example, multiple inheritance can lead to an object that contains multiple copies of its base class’ members—a condition known as the “deadly diamond” or “diamond of death.” (See Section 3.1.1.3 for more details.)

The difficulties in building an MI class hierarchy that works and that is understandable and maintainable usually outweigh the benefits. As a result, most game studios prohibit or severely limit the use of multiple inheritance in their class hierarchies.

Figure 16.5. A diamond-shaped class hierarchy for amphibious vehicles.

Mix-In Classes

Some teams do permit a limited form of MI, in which a class may have any number of parent classes but only *one* grandparent. In other words, a class may inherit from one and only one class in the main inheritance hierarchy, but it may also inherit from any number of *mix-in classes* (stand-alone classes with no base class). This permits common functionality to be factored out into a mix-in class and then spot-patched into the main hierarchy wherever it is needed. This is shown in Figure 16.6. However, as we’ll see below, it’s usually better to *compose* or *aggregate* such classes than to *inherit* from them.

图 16.6 包含混合类的类层次结构。MHealth 混合类为继承自它的任何类添加了生命值和被击杀的能力。MCarryable 混合类允许继承自它的对象被角色携带。

泡沫效应

当最初设计一个单体类层次结构时，根类通常非常简单，每个类只暴露一组极小的功能。然而，随着游戏中添加越来越多的功能，在两个或多个不相关的类之间共享代码的需求开始导致功能在层次结构中“向上冒泡”。

例如，我们一开始的设计可能只有木箱可以漂浮在水面上。然而，当我们的游戏设计师看到这些炫酷的漂浮木箱后，他们开始要求添加其他类型的漂浮物体，例如角色、纸片、车辆等等。由于在设计层次结构时，“漂浮与非漂浮”并非最初的分类标准之一，程序员很快发现需要为类层次结构中完全不相关的类添加漂浮功能。多重继承是不被接受的，因此程序员决定将漂浮代码向上移动到层次结构中，放入所有需要漂浮的对象共用的基类中。相比于在多个类中重复漂浮代码，一些从这个公共基类派生出来的类无法漂浮的问题看起来要小得多。（甚至可能会添加一个名为 m_bCanFloat 的布尔成员变量来明确区分。）最终的结果是，漂浮最终成为类层次结构中根对象的一个□□特性（以及游戏中几乎所有其他特性）。

Unreal 中的 Actor 类就是这种“冒泡效应”的典型例子。它包含用于管理渲染、动画、物理、世界交互、音效、多人游戏的网络复制的数据成员和代码，

Figure 16.6. A class hierarchy with mix-in classes. The MHealth mix-in class adds the notion of health and the ability to be killed to any class that inherits it. The MCarryable mix-in class allows an object that inherits it to be carried by a Character.

The Bubble-Up Effect

When a monolithic class hierarchy is first designed, the root class or classes are usually very simple, each one exposing only a minimal feature set. However, as more and more functionality is added to the game, the desire to *share code* between two or more *unrelated* classes begins to cause features to “bubble up” the hierarchy.

For example, we might start out with a design in which only wooden crates can float in water. However, once our game designers see those cool floating crates, they begin to ask for other kinds of floating objects, like characters, bits of paper, vehicles and so on. Because “floating versus non-floating” was not one of the original classification criteria when the hierarchy was designed, the programmers quickly discover the need to add flotation to classes that are totally unrelated within the class hierarchy. Multiple inheritance is frowned upon, so the programmers decide to move the flotation code up the hierarchy, into a base class that is common to all objects that need to float. The fact that some of the classes that derive from this common base class *cannot float* is seen as less of a problem than duplicating the flotation code across multiple classes. (A Boolean member variable called something like `m_bCanFloat` might even be added to make the distinction clear.) The ultimate result is that flotation eventually becomes a feature of the root object in the class hierarchy (along with pretty much every other feature in the game).

The `Actor` class in Unreal is a classic example of this “bubble-up effect.” It contains data members and code for managing rendering, animation, physics, world interaction, audio effects, network replication for multiplayer games,

对象的创建和销毁、Actor 迭代（即，遍历所有符合特定条件的 Actor 并对其执行某些操作的能力）以及消息广播。当功能被允许“冒泡”到单体类层次结构中最根的类时，封装各种引擎子系统的功能会非常困难。

16.2.1.4 使用组合来简化层次结构

造成类层次结构过于庞大的最常见原因或许是面向对象设计中过度使用“is-a”（是一个）关系。例如，在游戏的 GUI 中，程序员可能会决定从名为 Rectangle 的类派生出 Window 类，理由是 GUI 窗口始终是矩形。然而，窗口并非矩形——它有一个矩形，矩形定义了它的边界。因此，针对这个特定设计问题，一个更可行的解决方案是将 Rectangle 类的实例嵌入到 Window 类中，或者为 Window 类提供一个指向 Rectangle 的指针或引用。

在面向对象设计中，“has-a”关系被称为组合。
在组合中，类 A 要么直接包含类 B 的实例，要么包含指向类 B 实例的指针或引用。严格来说，为了使“组合”一词适用，类 A 必须拥有类 B。这意味着当创建类 A 的实例时，它也会自动创建类 B 的实例；当 A 的实例被销毁时，它的 B 实例也会被销毁。我们还可以通过指针或引用将类链接到另一个类，而无需其中一个类管理另一个类的生命周期。在这种情况下，该技术通常称为聚合。

将 Is-A 转换为 Has-A

将“is-a”关系转换为“has-a”关系，可以有效地减少游戏类层次结构的宽度、深度和复杂性。为了说明这一点，我们来看看图 16.7 所示的假设整体式层次结构。根 GameObject 类提供了所有游戏对象所需的一些基本功能（例如，RTTI、反射、通过序列化实现的持久化、网络复制等）。MovableObject 类表示任何具有变换（即位置、方向和可选比例）的游戏对象。RenderableObject 类增加了在屏幕上渲染的能力。（并非所有游戏对象都需要渲染——例如，不可见的 TriggerRegion 类可以直接从 MovableObject 派生。）CollidableObject 类为其实例提供碰撞信息。AnimatingObject 类通过骨骼关节层次结构赋予其实例动画的能力。最后，PhysicalObject 类为其实例提供了

object creation and destruction, actor iteration (i.e., the ability to iterate over all actors meeting a certain criteria and perform some operation on them), and message broadcasting. Encapsulating the functionality of various engine subsystems is difficult when features are permitted to “bubble up” to the rootmost classes in a monolithic class hierarchy.

16.2.1.4 Using Composition to Simplify the Hierarchy

Perhaps the most prevalent cause of monolithic class hierarchies is over-use of the “is-a” relationship in object-oriented design. For example, in a game’s GUI, a programmer might decide to derive the class `Window` from a class called `Rectangle`, using the logic that GUI windows are always rectangular. However, a window *is not a* rectangle—it *has a* rectangle, which defines its boundary. So a more workable solution to this particular design problem is to embed an instance of the `Rectangle` class inside the `Window` class, or to give the `Window` a pointer or reference to a `Rectangle`.

In object-oriented design, the “has-a” relationship is known as *composition*. In composition, a class A either contains an *instance* of class B directly, or contains a *pointer* or *reference* to an instance of B. Strictly speaking, in order for the term “composition” to be applicable, class A must *own* class B. This means that when an instance of class A is created, it automatically creates an instance of class B as well; when that instance of A is destroyed, its instance of B is destroyed, too. We can also link classes to one another via a pointer or reference *without* having one of the classes manage the other’s lifetime. In that case, the technique is usually called *aggregation*.

Converting Is-A to Has-A

Converting “is-a” relationships into “has-a” relationships can be a useful technique for reducing the width, depth and complexity of a game’s class hierarchy. To illustrate, let’s take a look at the hypothetical monolithic hierarchy shown in Figure 16.7. The root `GameObject` class provides some basic functionality required by all game objects (e.g., RTTI, reflection, persistence via serialization, network replication, etc.). The `MovableObject` class represents any game object that has a *transform* (i.e., a position, orientation and optional scale). `RenderableObject` adds the ability to be rendered onscreen. (Not all game objects need to be rendered—for example, an invisible `TriggerRegion` class could be derived directly from `MovableObject`.) The `CollidableObject` class provides collision information to its instances. The `AnimatingObject` class grants to its instances the ability to be animated via a skeletal joint hierarchy. Finally, the `PhysicalObject` gives its instances the

物理模拟的能力（例如，刚体在重力影响下下落并在游戏世界中弹跳）。

这种类层次结构的一大问题是，它限制了我们在创建新游戏对象类型的设计选择。如果我们想要定义一个物理模拟的对象类型，我们就必须从 PhysicalObject 派生它的类，即使它可能不需要骨骼动画。如果我们想要一个具有碰撞的游戏对象类，它必须从 CollidableObject 继承，即使它可能是不可见的，因此不需要 RenderableObject 的服务。

图 16.7. 假设的游戏对象类层次结构仅使用继承

to 联系 这课程。

图 16.7 所示的层次结构的第二个问题是，现有类的功能难以扩展。例如，假设我们想要支持变形目标动画，因此我们从 AnimatingObject 派生出两个新类，分别名为 SkeletalObject 和 Morph TargetObject 。如果我们希望这两个新类都具有物理模拟能力，我们就不得不将 PhysicalObject 重构为两个几乎相同的类，一个派生自 SkeletalObject ，一个派生自 MorphTargetObject ，或者转向多重继承。

解决这些问题的一个方法是将 GameObject 的各种功能隔离到独立的类中，每个类提供单一的、定义明确的服务。这样的类有时被称为组件或服务对象。组件化设计允许我们只选择我们创建的每种游戏对象所需的功能。此外，它允许每个功能进行维护、扩展或重构，而不会影响其他功能。各个组件也更容易理解，更容易测试，因为它们彼此分离。一些组件类直接对应于单个引擎子系统，例如渲染、动画、碰撞、物理、音频等。这使得这些子系统在集成在一起供特定游戏对象使用时保持独特且封装良好。

图 16.8 展示了将类层次结构重构为组件后的样子。在这个修改后的设计中，GameObject 类充当枢纽，包含指向我们定义的每个可选组件的指针。 MeshInstance 组件取代了 RenderableObject 类——它表示一个三角形网格的实例，并封装了如何渲染它的知识。同样， AnimationController 组件取代了 AnimatingObject ，向 GameObject 公开了骨骼动画服务。 Transform 类取代了 MovableObject ，它维护了对象的位置、方向和比例。 RigidBody 类表示游戏对象的碰撞几何形状，并为其 GameObject 提供了一个与低级碰撞和物理系统的接口，取代了

ability to be physically simulated (e.g., a rigid body falling under the influence of gravity and bouncing around in the game world).

One big problem with this class hierarchy is that it limits our design choices when creating new types of game objects. If we want to define an object type that is physically simulated, we are forced to derive its class from `PhysicalObject` even though it may not require skeletal animation. If we want a game object class with collision, it must inherit from `CollidableObject` even though it may be invisible and hence not require the services of `RenderableObject`.

Figure 16.7. A hypothetical game object class hierarchy using only inheritance to associate the classes.

A second problem with the hierarchy shown in Figure 16.7 is that it is difficult to extend the functionality of the existing classes. For example, let's imagine we want to support morph target animation, so we derive two new classes from `AnimatingObject` called `SkeletalObject` and `MorphTargetObject`. If we wanted both of these new classes to have the ability to be physically simulated, we'd be forced to refactor `PhysicalObject` into two nearly identical classes, one derived from `SkeletalObject` and one from `MorphTargetObject`, or turn to multiple inheritance.

One solution to these problems is to isolate the various features of a `GameObject` into independent classes, each of which provides a single, well-defined service. Such classes are sometimes called *components* or *service objects*. A componentized design allows us to select only those features we need for each type of game object we create. In addition, it permits each feature to be maintained, extended or refactored without affecting the others. The individual components are also easier to understand, and easier to test, because they are decoupled from one another. Some component classes correspond directly to a single engine subsystem, such as rendering, animation, collision, physics, audio, etc. This allows these subsystems to remain distinct and well-encapsulated when they are integrated together for use by a particular game object.

Figure 16.8 shows how our class hierarchy might look after refactoring it into components. In this revised design, the `GameObject` class acts like a hub, containing pointers to each of the optional components we've defined. The `MeshInstance` component is our replacement for the `RenderableObject` class—it represents an instance of a triangle mesh and encapsulates the knowledge of how to render it. Likewise, the `AnimationController` component replaces `AnimatingObject`, exposing skeletal animation services to the `GameObject`. Class `Transform` replaces `MovableObject` by maintaining the position, orientation and scale of the object. The `RigidBody` class represents the collision geometry of a game object and provides its `GameObject` with an interface into the low-level collision and physics systems, replacing

图 16.8。我们假设的游戏对象类层次结构，重构后更倾向于类组合而不是继承。

CollidableObject 和 PhysicalObject 。

组件创建和所有权

在这种设计中，通常“中心”类拥有其组件，这意味着它管理组件的生命周期。但 GameObject 应该如何“知道”要创建哪些组件呢？有很多方法可以解决这个问题，但最简单的方法之一是向根 GameObject 类提供指向所有可能组件的指针。每种唯一类型的游戏对象都被定义为 GameObject 的派生类。在 GameObject 构造函数中，所有组件指针最初都设置为 nullptr。每个派生类的构造函数都可以自由创建它可能需要的任何组件。为方便起见，默认的 GameObject 析构函数可以自动清理所有组件。在这种设计中，从 GameObject 派生的类的层次结构作为我们在游戏中想要的对象类型的主要分类，而组件类则作为可选的附加功能。

此类层次结构的组件创建和销毁逻辑的一种可能实现如下所示。然而，务必注意，此代码仅仅是一个示例——即使在采用基本相同类层次结构设计的引擎之间，实现细节也可能存在很大差异。

```
class GameObject
{
protected:
```

Figure 16.8. Our hypothetical game object class hierarchy, refactored to favor class composition over inheritance.

`CollidableObject` and `PhysicalObject`.

Component Creation and Ownership

In this kind of design, it is typical for the “hub” class to *own* its components, meaning that it manages their *lifetimes*. But how should a `GameObject` “know” which components to create? There are numerous ways to solve this problem, but one of the simplest is to provide the root `GameObject` class with pointers to all possible components. Each unique type of game object is defined as a derived class of `GameObject`. In the `GameObject` constructor, all of the component pointers are initially set to `nullptr`. Each derived class’s constructor is then free to create whatever components it may need. For convenience, the default `GameObject` destructor can clean up all of the components automatically. In this design, the hierarchy of classes derived from `GameObject` serves as the primary taxonomy for the kinds of objects we want in our game, and the component classes serve as optional add-on features.

One possible implementation of the component creation and destruction logic for this kind of hierarchy is shown below. However, it’s important to realize that this code is just an example—implementation details vary widely, even between engines that employ essentially the same kind of class hierarchy design.

```
class GameObject
{
protected:
```

```
    // My transform (position, rotation, scale).
    Transform            m_transform;

    // Standard components:
    MeshInstance*        m_pMeshInst;
    AnimationController* m_pAnimController;
    RigidBody*           m_pRigidBody;

public:
    GameObject()
    {
        // Assume no components by default.
        // Derived classes will override.
        m_pMeshInst = nullptr;
        m_pAnimController = nullptr;
        m_pRigidBody = nullptr;
    }

    ~GameObject()
    {
        // Automatically delete any components created by
        // derived classes. (Deleting null pointers OK.)
        delete m_pMeshInst;
        delete m_pAnimController;
        delete m_pRigidBody;
    }

    // ...
};

class Vehicle : public GameObject
{
protected:
    // Add some more components specific to Vehicles...
    Chassis* m_pChassis;
    Engine*  m_pEngine;
    // ...

public:
    Vehicle()
    {
        // Construct standard GameObject components.
        m_pMeshInst = new MeshInstance;
        m_pRigidBody = new RigidBody;

        // NOTE: We'll assume the animation controller
        // must be provided with a reference to the mesh
```



```
    // My transform (position, rotation, scale).
    Transform            m_transform;

    // Standard components:
    MeshInstance*        m_pMeshInst;
    AnimationController* m_pAnimController;
    RigidBody*           m_pRigidBody;

public:
    GameObject()
    {
        // Assume no components by default.
        // Derived classes will override.
        m_pMeshInst = nullptr;
        m_pAnimController = nullptr;
        m_pRigidBody = nullptr;
    }

    ~GameObject()
    {
        // Automatically delete any components created by
        // derived classes. (Deleting null pointers OK.)
        delete m_pMeshInst;
        delete m_pAnimController;
        delete m_pRigidBody;
    }

    // ...
};

class Vehicle : public GameObject
{
protected:
    // Add some more components specific to Vehicles...
    Chassis* m_pChassis;
    Engine*  m_pEngine;
    // ...

public:
    Vehicle()
    {
        // Construct standard GameObject components.
        m_pMeshInst = new MeshInstance;
        m_pRigidBody = new RigidBody;

        // NOTE: We'll assume the animation controller
        // must be provided with a reference to the mesh
```

```
        // instance so that it can provide it with a
        // matrix palette.
        m_pAnimController
         = new AnimationController(*m_pMeshInst);

        // Construct vehicle-specific components.
        m_pChassis = new Chassis(*this,
                                 *m_pAnimController);

        m_pEngine = new Engine(*this);
    }

    ~Vehicle()
    {
        // Only need to destroy vehicle-specific
        // components, as GameObject cleans up the
        // standard components for us.
        delete m_pChassis;
        delete m_pEngine;
    }
};
```

16.2.1.5 通用组件

另一种更灵活（但实现起来也更棘手）的方案是为根游戏对象类提供一个通用的组件链表。这种设计中的组件通常都派生自一个公共基类——这使我们能够遍历链表并执行多态操作，例如询问每个组件的类型，或依次将事件传递给每个组件进行处理。这种设计使得根游戏对象类几乎不考虑可用的组件类型，从而在许多情况下允许在不修改游戏对象类的情况下创建新类型的组件。它还允许特定游戏对象包含每种组件类型的任意数量的实例。（硬编码设计只允许固定数量的实例，这取决于游戏对象类中指向每个组件的指针数量。）这种设计如图 16.9 所示。与硬编码组件模型相比，它的实现难度更大，因为游戏对象代码必须以完全通用的方式编写。同样，组件类也不能假设在特定游戏对象的上下文中可能存在或不存在其他组件。在硬编码组件指针和使用通用的组件链表之间做出选择并非易事。两种设计并没有明显的优劣之分——它们各有利弊，不同的游戏团队也会采用不同的方法。


```
        // instance so that it can provide it with a
        // matrix palette.
        m_pAnimController
         = new AnimationController(*m_pMeshInst);

        // Construct vehicle-specific components.
        m_pChassis = new Chassis(*this,
                                 *m_pAnimController);

        m_pEngine = new Engine(*this);
    }

    ~Vehicle()
    {
        // Only need to destroy vehicle-specific
        // components, as GameObject cleans up the
        // standard components for us.
        delete m_pChassis;
        delete m_pEngine;
    }
};
```

16.2.1.5 Generic Components

Another more flexible (but also trickier to implement) alternative is to provide the root game object class with a generic linked list of components. The components in such a design usually all derive from a common base class—this allows us to iterate over the linked list and perform polymorphic operations, such as asking each component what type it is or passing an event to each component in turn for possible handling. This design allows the root game object class to be largely oblivious to the component types that are available and thereby permits new types of components to be created without modifying the game object class in many cases. It also allows a particular game object to contain an arbitrary number of instances of each type of component. (The hard-coded design permits only a fixed number, determined by how many pointers to each component exist within the game object class.)

This kind of design is illustrated in Figure 16.9. It is trickier to implement than a hard-coded component model because the game object code must be written in a totally generic way. The component classes can likewise make no assumptions about what other components might or might not exist within the context of a particular game object. The choice between hard-coding the component pointers or using a generic linked list of components is not an easy one to make. Neither design is clearly superior—they each have their pros and cons, and different game teams take different approaches.

图 16.9。组件的链接列表可以通过允许中心游戏对象不知道任何特定组件的细节来提供灵活性。

16.2.1.6 纯组件模型

如果我们将组件化概念发挥到极致，会发生什么？我们会将所有功能从根 GameObject 类中移出，并放入各种组件类中。此时，游戏对象类实际上就是一个没有行为的容器，它拥有唯一的 ID 和一堆指向其组件的指针，但除此之外，它本身不包含任何逻辑。那么，为什么不完全消除这个类呢？一种方法是为每个组件提供游戏对象唯一 ID 的副本。现在，这些组件通过 ID 链接在一起，形成一个逻辑分组。如果能够通过 ID 快速查找任何组件，我们就不再需要 GameObject 这个“中心”类了。我将使用术语“纯组件模型”来描述这种架构。如图 16.10 所示。

图 16.10。在纯组件模型中，逻辑游戏对象由许多组件组成，但这些组件仅通过共享唯一 ID 间接链接在一起。

Figure 16.9. A linked list of components can provide flexibility by allowing the hub game object to be unaware of the details of any particular component.

16.2.1.6 Pure Component Models

What would happen if we were to take the componentization concept to its extreme? We would move literally *all* of the functionality out of our root `GameObject` class into various component classes. At this point, the game object class would quite literally be a behavior-less container, with a unique id and a bunch of pointers to its components, but otherwise containing no logic of its own. So why not eliminate the class entirely? One way to do this is to give each component a copy of the game object's unique id. The components are now linked together into a logical grouping by id. Given a way to quickly look up any component by id, we would no longer need the `GameObject` "hub" class at all. I will use the term *pure component model* to describe this kind of architecture. It is illustrated in Figure 16.10.

Figure 16.10. In a pure component model, a logical game object is comprised of many components, but the components are linked together only indirectly, by sharing a unique id.

纯组件模型并不像听起来那么简单，也并非没有问题。首先，我们仍然需要某种方式来定义游戏所需的各种具体类型的游戏对象，然后安排在创建该类型的实例时实例化正确的组件类。我们的 GameObject 层次结构过去常常为我们处理组件的构造。相反，我们可以使用工厂模式，在该模式中，我们为每个游戏对象类型定义一个工厂类，并使用一个虚拟构造函数来重写该函数，以便为每种游戏对象类型创建适当的组件。或者，我们可以转向数据驱动模型，其中游戏对象类型定义在一个文本文件中，引擎可以解析该文件，并在实例化类型时参考该文件。

纯组件设计的另一个问题是组件间通信。我们的中心游戏对象充当了“枢纽”，负责协调各个组件之间的通信。在纯组件架构中，我们需要一种高效的方式让组成单个游戏对象的组件之间相互通信。这可以通过让每个组件使用游戏对象的唯一 ID 来查找其他组件来实现。然而，我们可能需要一种更高效的机制——例如，可以将组件预先连接到一个循环链表中。

同样，在纯组件化模型中，从一个游戏对象向另一个游戏对象发送消息也很困难。我们无法再与 GameObject 实例通信，因此我们要么需要预先知道要与哪个组件通信，要么必须向组成该游戏对象的所有组件进行多播。这两种选择都不是理想的选择。

纯组件模型可以并且已经在实际的游戏项目中应用。这类模型各有利弊，但同样，它们并不比其他任何替代设计明显更优。除非你参与研发工作，否则你应该选择你最熟悉、最有信心、并且最符合你正在构建的特定游戏需求的架构。

16.2.2 以属性为中心的架构

经常使用面向对象编程语言的程序员，往往会自然而然地将对象视为包含属性（数据成员）和行为（方法、成员函数）的对象。这就是以对象为中心的观点：

- 对象1
  - 位置 = (0, 3, 15)
  - 方向 = (0, 43, 0)

A pure component model is not quite as simple as it first sounds, and it is not without its share of problems. For one thing, we still need some way of defining the various concrete types of game objects our game needs and then arranging for the correct component classes to be instantiated whenever an instance of the type is created. Our `GameObject` hierarchy used to handle construction of components for us. Instead, we might use a factory pattern, in which we define factory classes, one per game object type, with a virtual construction function that is overridden to create the proper components for each game object type. Or we might turn to a data-driven model, where the game object types are defined in a text file that can be parsed by the engine and consulted whenever a type is instantiated.

Another issue with a components-only design is inter-component communication. Our central `GameObject` acted as a “hub,” marshalling communications between the various components. In pure component architectures, we need an efficient way for the components making up a single game object to talk to one another. This could be done by having each component look up the other components using the game object’s unique id. However, we probably want a much more efficient mechanism—for example the components could be prewired into a circular linked list.

In the same sense, sending messages from one game object to another is difficult in a pure componentized model. We can no longer communicate with the `GameObject` instance, so we either need to know a priori with which component we wish to communicate, or we must multicast to all components that make up the game object in question. Neither option is ideal.

Pure component models can and have been made to work on real game projects. These kinds of models have their pros and cons, but again, they are not clearly better than any of the alternative designs. Unless you’re part of a research and development effort, you should probably choose the architecture with which you are most comfortable and confident, and which best fits the needs of the particular game you are building.

16.2.2 Property-Centric Architectures

Programmers who work frequently in an object-oriented programming language tend to think naturally in terms of objects that contain attributes (data members) and behaviors (methods, member functions). This is the *object-centric view*:

- Object1
  - Position = (0, 3, 15)
  - Orientation = (0, 43, 0)

- 对象 2
  - 位置 = （− 12 , 0, 8）
  - 健康 = 15
- 对象 3
  - 方向 = (0, − 87 , 10)

然而，我们可以主要从属性的角度来思考，而不是从对象的角度。我们定义一个游戏对象可能拥有的所有属性的集合。然后，对于每个属性，我们构建一个表，其中包含每个拥有该属性的游戏对象对应的属性值。属性值由对象的唯一 ID 作为键。这就是我们所说的“以属性为中心”的视图：

- 位置
  - 对象 1 = (0, 3, 15)
  - 对象 2 = (− 12 , 0, 8)
- 方向
  - 对象 1 = (0, 43, 0)
  - 对象 3 = (0, − 87 , 10)
- 健康
  - 对象 2 = 15

以属性为中心的对象模型已在许多商业游戏中得到成功运用，包括《杀出重围 2》和《神偷》系列游戏。有关这些项目如何设计其对象系统的详细信息，请参阅第 16.2.2.5 节。

以属性为中心的设计与其说是对象模型，不如说更类似于关系数据库。每个属性都像关系数据库中的一张表，游戏对象的唯一 ID 是其主键。当然，在面向对象设计中，对象不仅由其属性定义，还由其行为定义。如果我们只有属性表，那么我们在哪里实现行为呢？这个问题的答案因引擎而异，但最常见的是，行为在以下一个或两个地方实现：

- 在房产本身，和/或
- 通过脚本代码。

让我们进一步探讨这些想法。

- Object2
  - Position = (−12, 0, 8)
  - Health = 15
- Object3
  - Orientation = (0, −87, 10)

However, it is possible to think primarily in terms of the attributes, rather than the objects. We define the set of all properties that a game object might have. Then for each property, we build a table containing the values of that property corresponding to each game object that has it. The property values are keyed by the objects' unique ids. This is what we will call the *property-centric view*:

- Position
  - Object1 = (0, 3, 15)
  - Object2 = (−12, 0, 8)
- Orientation
  - Object1 = (0, 43, 0)
  - Object3 = (0, −87, 10)
- Health
  - Object2 = 15

Property-centric object models have been used very successfully on many commercial games, including *Deus Ex 2* and the *Thief* series of games. See Section 16.2.2.5 for more details on exactly how these projects designed their object systems.

A property-centric design is more akin to a relational database than an object model. Each attribute acts like a table in a relational database, with the game objects' unique id as the *primary key*. Of course, in object-oriented design, an object is defined not only by its *attributes*, but also by its *behavior*. If all we have are tables of properties, then where do we implement the behavior? The answer to this question varies somewhat from engine to engine, but most often the behaviors are implemented in one or both of the following places:

- in the properties themselves, and/or
- via script code.

Let's explore each of these ideas further.

16.2.2.1 通过属性类实现行为

每种类型的属性都可以实现为一个属性类。属性可以简单到单个布尔值或浮点值，也可以复杂到可渲染的三角形网格或 AI“大脑”。每个属性类都可以通过其硬编码方法（成员函数）提供行为。特定游戏对象的整体行为由其所有属性行为的集合决定。

例如，如果某个游戏对象包含 Health 属性的实例，则它可能受到伤害并最终被摧毁或杀死。Health 对象可以通过适当降低对象的健康等级来响应针对该游戏对象的任何攻击。属性对象还可以与同一游戏对象内的其他属性对象通信，以产生协作行为。例如，当 Health 属性检测到并响应攻击时，它可能会向 AnimatedSkeleton 属性发送一条消息，从而允许游戏对象播放合适的命中反应动画。同样，当 Health 属性检测到游戏对象即将死亡或被摧毁时，它可以与 RigidBodyDynamics 属性通信，以激活物理驱动的爆炸或“布娃娃”尸体模拟。

16.2.2.2 通过脚本实现行为

另一种选择是将属性值作为原始数据存储在一个或多个类似数据库的表中，并使用脚本代码实现游戏对象的行为。每个游戏对象都可以拥有一个名为 ScriptId 的特殊属性，该属性（如果存在）指定用于管理对象行为的脚本代码块（脚本函数；如果脚本语言本身是面向对象的，则为脚本对象）。脚本代码还可用于允许游戏对象响应游戏世界中发生的事件。有关事件系统的更多详细信息，请参阅第 16.8 节；有关游戏脚本语言的讨论，请参阅第 16.9 节。

在一些以属性为中心的引擎中，工程师会提供一组核心的硬编码属性类，但也提供了允许游戏设计师和程序员完全在脚本中实现新属性类型的功能。例如，这种方法在《地牢围攻》项目中就得到了成功的应用。

16.2.2.3 属性与组件

值得注意的是，在 16.2.2.5 节中引用的许多作者都使用术语“组件”来指代我在这里所说的“属性对象”。在 16.2.1.4 节中，我使用术语“组件”来指代以对象为中心的设计中的子对象，这与属性对象并不完全相同。

16.2.2.1 Implementing Behavior via Property Classes

Each type of property can be implemented as a *property class*. Properties can be as simple as a single Boolean or floating-point value or as complex as a renderable triangle mesh or an AI “brain.” Each property class can provide behaviors via its hard-coded methods (member functions). The overall behavior of a particular game object is determined by the aggregation of the behaviors of all its properties.

For example, if a game object contains an instance of the `Health` property, it can be damaged and eventually destroyed or killed. The `Health` object can respond to any attacks made on the game object by decrementing the object’s health level appropriately. A property object can also communicate with other property objects within the same game object to produce cooperative behaviors. For example, when the `Health` property detects and responds to an attack, it could possibly send a message to the `AnimatedSkeleton` property, thereby allowing the game object to play a suitable hit reaction animation. Similarly, when the `Health` property detects that the game object is about to die or be destroyed, it can talk to the `RigidBodyDynamics` property to activate a physics-driven explosion or a “rag doll” dead body simulation.

16.2.2.2 Implementing Behavior via Script

Another option is to store the property values as raw data in one or more database-like tables and use script code to implement a game object’s behaviors. Every game object could have a special property called something like `ScriptId`, which, if present, specifies the block of script code (script function, or script object if the scripting language is itself object-oriented) that will manage the object’s behavior. Script code could also be used to allow a game object to respond to events that occur within the game world. See Section 16.8 for more details on event systems and Section 16.9 for a discussion of game scripting languages.

In some property-centric engines, a core set of hard-coded property classes is provided by the engineers, but a facility is provided allowing game designers and programmers to implement new property types entirely in script. This approach was used successfully on the *Dungeon Siege* project, for example.

16.2.2.3 Properties versus Components

It’s important to note that many of the authors cited in Section 16.2.2.5 use the term “component” to refer to what I call a “property object” here. In Section 16.2.1.4, I used the term “component” to refer to a subobject in an object-centric design, which isn’t quite the same as a property object.

然而，属性对象在很多方面与组件密切相关。在这两种设计中，单个逻辑游戏对象都由多个子对象组成。主要区别在于子对象的角色。在以属性为中心的设计中，每个子对象定义游戏对象本身的特定属性（例如，生命值、视觉呈现、物品栏、特定魔法力量等）；而在基于组件（以对象为中心）的设计中，子对象通常表示与特定低级引擎子系统（渲染器、动画、碰撞和动态等）的链接。这种区别非常微妙，在许多情况下几乎无关紧要。您可以根据自己的喜好将您的设计称为纯组件模型（第 16.2.1.6 节）或以属性为中心的设计，但最终，您将获得基本相同的结果——一个由子对象集合组成并从中派生其行为的逻辑游戏对象。

16.2.2.4 以属性为中心的设计的优缺点

以属性为中心的方法有很多潜在的好处。它往往更节省内存，因为我们只需要存储实际使用的属性数据（即，游戏对象中永远不会有未使用的数据成员）。以数据驱动的方式构建这样的模型也更容易——设计师可以轻松定义新属性，而无需重新编译游戏，因为无需更改游戏对象类的定义。只有当需要添加全新类型的属性时（假设该属性无法通过脚本添加），程序员才需要参与。

以属性为中心的设计也比以对象为中心的模型更有利于缓存，因为相同类型的数据在内存中是连续存储的。这是现代游戏硬件中常见的优化技术，因为访问内存的成本远高于执行指令和计算的成本。（例如，在 PlayStation 3 上，一次缓存未命中的成本相当于执行数千条 CPU 指令的成本。）通过在 RAM 中连续存储数据，我们可以减少或消除缓存未命中，因为当我们访问数据数组中的一个元素时，其大量相邻元素会被加载到同一个缓存行中。这种数据设计方法有时被称为“结构体数组”技术，与更传统的“结构体数组”方法相对。下面的代码片段说明了这两种内存布局之间的差异。（请注意，我们不会真正以这种方式实现游戏对象模型 - 这个例子只是为了说明以属性为中心的设计倾向于生成许多连续的同类类型数据数组，而不是单个复杂对象数组。）

```
static const U32 MAX_GAME_OBJECTS = 1024;
```


However, property objects are very closely related to components in many ways. In both designs, a single logical game object is made up of multiple subobjects. The main distinction lies in the roles of the subobjects. In a property-centric design, each subobject defines a particular attribute of the game object itself (e.g., health, visual representation, inventory, a particular magic power, etc.), whereas in a component-based (object-centric) design, the subobjects often represent linkages to particular low-level engine subsystems (renderer, animation, collision and dynamics, etc.) This distinction is so subtle as to be virtually irrelevant in many cases. You can call your design a *pure component model* (Section 16.2.1.6) or a *property-centric design* as you see fit, but at the end of the day, you'll have essentially the same result—a logical game object that is comprised of, and derives its behavior from, a collection of subobjects.

16.2.2.4 Pros and Cons of Property-Centric Designs

There are a number of potential benefits to an attribute-centric approach. It tends to be more memory-efficient, because we need only store attribute data that is actually in use (i.e., there are never game objects with unused data members). It is also easier to construct such a model in a data-driven manner—designers can define new attributes easily, without recompiling the game, because there are no game object class definitions to be changed. Programmers need only get involved when entirely new types of properties need to be added (presuming the property cannot be added via script).

A property-centric design can also be more cache-friendly than an object-centric model, because data of the same type is stored *contiguously* in memory. This is a commonplace optimization technique on modern gaming hardware, where the cost of accessing memory is far higher than the cost of executing instructions and performing calculations. (For example, on the PlayStation 3, the cost of a single cache miss is equivalent to the cost of executing literally thousands of CPU instructions.) By storing data contiguously in RAM, we can reduce or eliminate cache misses, because when we access one element of a data array, a large number of its neighboring elements are loaded into the same cache line. This approach to data design is sometimes called the *struct of arrays* technique, in contrast to the more-traditional *array of structs* approach. The differences between these two memory layouts are illustrated by the code snippet below. (Note that we wouldn't really implement a game object model in exactly this way—this example is meant only to illustrate the way in which a property-centric design tends to produce many contiguous arrays of like-typed data, rather than a single array of complex objects.)

```
static const U32 MAX_GAME_OBJECTS = 1024;
```

```
// Traditional array-of-structs approach.

struct GameObject
{
    U32         m_uniqueId;
    Vector      m_pos;
    Quaternion  m_rot;
    float       m_health;

    // ...
};

GameObject g_aAllGameObjects[MAX_GAME_OBJECTS];

// Cache-friendlier struct-of-arrays approach.

struct AllGameObjects
{
    U32         m_aUniqueId[MAX_GAME_OBJECTS];
    Vector      m_aPos[MAX_GAME_OBJECTS];
    Quaternion  m_aRot[MAX_GAME_OBJECTS];
    float       m_aHealth[MAX_GAME_OBJECTS];

    // ...
};

AllGameObjects g_allGameObjects;
```

以属性为中心的模型也存在一些问题。例如，当一个游戏对象只是一堆属性的混合体时，强化这些属性之间的关系就会变得更加困难。仅仅将一组属性对象的细粒度行为拼凑在一起，很难实现所需的大规模行为。调试这样的系统也更加棘手，因为程序员无法将游戏对象直接放入调试器的监视窗口来一次性检查其所有属性。

16.2.2.5 进一步阅读

游戏行业的杰出工程师在各种游戏开发会议上就以属性为中心的架构这一主题进行了多次有趣的 PowerPoint 演示。

- Rob Fermier，《创建数据驱动引擎》，游戏开发者大会，2002 年。
- Scott Bilas，《数据驱动的游戏对象系统》，游戏开发者大会，2002 年。


```
// Traditional array-of-structs approach.

struct GameObject
{
    U32         m_uniqueId;
    Vector      m_pos;
    Quaternion  m_rot;
    float       m_health;

    // ...
};

GameObject g_aAllGameObjects[MAX_GAME_OBJECTS];

// Cache-friendlier struct-of-arrays approach.

struct AllGameObjects
{
    U32         m_aUniqueId[MAX_GAME_OBJECTS];
    Vector      m_aPos[MAX_GAME_OBJECTS];
    Quaternion  m_aRot[MAX_GAME_OBJECTS];
    float       m_aHealth[MAX_GAME_OBJECTS];

    // ...
};

AllGameObjects g_allGameObjects;
```


Attribute-centric models have their share of problems as well. For example, when a game object is just a grab bag of properties, it becomes much more difficult to enforce relationships between those properties. It can be hard to implement a desired large-scale behavior merely by cobbling together the fine-grained behaviors of a group of property objects. It's also much trickier to debug such systems, as the programmer cannot slap a game object into the watch window in the debugger in order to inspect all of its properties at once.

16.2.2.5 Further Reading

A number of interesting PowerPoint presentations on the topic of property-centric architectures have been given by prominent engineers in the game industry at various game development conferences.

- Rob Fermier, "Creating a Data Driven Engine," Game Developer's Conference, 2002.
- Scott Bilas, "A Data-Driven Game Object System," Game Developer's Conference, 2002.

• Alex Duran，《构建对象系统：特性、权衡和陷阱》，游戏开发者大会，2003 年。

• Jeremy Chatelaine，《通过现有工具实现数据驱动的调整》，游戏开发者大会，2003 年。

• Doug Church，《对象系统》，在 2003 年韩国首尔的游戏开发会议上发表；会议由 Chris Hecker、Casey Muratori、Jon Blow 和 Doug Church 组织。http://chrishecker.com/images/6/6f/ObjSys.ppt。

16.3 世界块数据格式

正如我们所见，世界块通常包含静态和动态世界元素。静态几何体可以由一个大的三角形网格表示，也可以由许多较小的网格组成。每个网格可能会被多次实例化 - 例如，单个门网格可能会被重复用于块中的所有门口。静态数据通常包括存储为三角形汤的碰撞信息，凸形的集合和/或其他更简单的几何形状，如平面，盒子，胶囊或球体。其他静态元素包括可用于检测事件或描绘游戏世界内区域的体积区域，AI导航网格，一组描绘背景几何体内边缘的线段，玩家角色可以抓取这些线段等等。我们不会在这里详细介绍这些数据格式，因为我们已经在前面的部分中讨论了其中的大部分。

世界块的动态部分包含该块内游戏对象的某种表示。游戏对象由其属性和行为定义，而对象的行为则直接或间接地由其类型决定。在以对象为中心的设计中，对象的类型直接决定了在运行时需要实例化哪些类来表示该对象。在以属性为中心的设计中，游戏对象的行为由其属性行为的组合决定，但类型仍然决定了对象应该具有哪些属性（或者可以说对象的属性定义了它的类型）。因此，对于每个游戏对象，世界块数据文件通常包含：

• 对象属性的初始值。世界块定义了每个游戏对象在游戏世界中首次生成时的状态。对象的属性数据可以以多种不同的格式存储。我们将在下文探讨几种常用的格式。

- Alex Duran, "Building Object Systems: Features, Tradeoffs, and Pitfalls," Game Developer's Conference, 2003.
- Jeremy Chatelaine, "Enabling Data Driven Tuning via Existing Tools," Game Developer's Conference, 2003.
- Doug Church, "Object Systems," presented at a game development conference in Seoul, Korea, 2003; conference organized by Chris Hecker, Casey Muratori, Jon Blow and Doug Church. http://chrishecker.com/images/6/6f/ObjSys.ppt.

16.3 World Chunk Data Formats

As we've seen, a world chunk generally contains both *static* and *dynamic* world elements. The static geometry might be represented by one big triangle mesh, or it might be comprised of many smaller meshes. Each mesh might be *instanced* multiple times—for example, a single door mesh might be reused for all of the doorways in the chunk. The static data usually includes collision information stored as a triangle soup, a collection of convex shapes and/or other simpler geometric shapes like planes, boxes, capsules or spheres. Other static elements include volumetric *regions* that can be used to detect events or delineate areas within the game world, an AI *navigation mesh*, a set of line segments delineating *edges* within the background geometry that can be grabbed by the player character and so on. We won't get into the details of these data formats here, because we've already discussed most of them in previous sections.

The dynamic portion of the world chunk contains some kind of representation of the game objects within that chunk. A game object is defined by its *attributes* and its *behaviors*, and an object's behaviors are determined either directly or indirectly by its *type*. In an object-centric design, the object's type directly determines which class(es) to instantiate in order to represent the object at runtime. In a property-centric design, a game object's behavior is determined by the amalgamation of the behaviors of its properties, but the type still determines which properties the object should have (or one might say that an object's properties define its type). So, for each game object, a world chunk data file generally contains:

- *The initial values of the objects' attributes*. The world chunk defines the state of each game object as it should exist when first spawned into the game world. An object's attribute data can be stored in a number of different formats. We'll explore a few popular formats below.

• 对象类型的某种规范。在以对象为中心的引擎中，这可能是一个字符串、散列字符串 ID 或其他一些唯一的类型 ID。在以属性为中心的设计中，类型可能被显式存储，也可能由构成对象的属性/特性集合隐式定义。

16.3.1 二进制对象图像

将游戏对象集合存储到磁盘文件中的一种方法是将每个对象的二进制图像写入文件，使其与运行时在内存中的样子完全相同。这使得生成游戏对象变得非常简单。一旦游戏世界块加载到内存中，我们就有了所有对象的现成图像，因此我们只需让它们自由移动即可。

嗯，不完全是。存储"实时" C++ 类实例的二进制映像存在诸多问题，包括需要以特殊方式处理指针和虚拟表，以及可能需要在每个类实例内交换数据的字节序。（这些技术将在 7.2.2.9 节中详细介绍。）此外，二进制对象映像缺乏灵活性，且在发生更改时不够稳健。游戏玩法是任何游戏项目中最动态、最不稳定的方面之一，因此选择一种支持快速开发且能够应对频繁更改的数据格式是明智之举。因此，二进制对象映像格式通常不是存储游戏对象数据的理想选择（尽管这种格式可能适用于更稳定的数据结构，例如网格数据或碰撞几何体）。

16.3.2 序列化游戏对象描述

序列化是将游戏对象内部状态表示存储到磁盘文件的另一种方法。与二进制对象映像技术相比，这种方法更易于移植且更易于实现。要将对象序列化到磁盘，需要该对象生成一个包含足够详细信息的数据流，以便稍后重建原始对象。当对象从磁盘序列化回内存时，会创建相应类的实例，然后读取属性数据流以初始化新对象的内部状态。如果原始序列化数据流完整，则新对象在所有方面都应该与原始对象完全相同。

一些编程语言原生支持序列化。例如，C# 和 Java 都提供了将对象实例序列化为 XML 文本格式以及从 XML 文本格式序列化的标准化机制。遗憾的是，C++ 语言并没有提供标准化的序列化功能。然而，许多 C++ 序列化系统已经成功构建，包括内部和外部的。

- *Some kind of specification of the object's type.* In an object-centric engine, this might be a string, a hashed string id or some other unique type id. In a property-centric design, the type might be stored explicitly, or it might be defined implicitly by the collection of properties/attributes of which the object is comprised.

16.3.1 Binary Object Images

One way to store a collection of game objects into a disk file is to write a binary image of each object into the file, exactly as it looks in memory at runtime. This makes spawning game objects trivial. Once the game world chunk has been loaded into memory, we have ready-made images of all our objects, so we simply let them fly.

Well, not quite. Storing binary images of "live" C++ class instances is problematic for a number of reasons, including the need to handle *pointers* and *virtual tables* in a special way, and the possibility of having to *endian-swap* the data within each class instance. (These techniques are described in detail in Section 7.2.2.9.) Moreover, binary object images are inflexible and not robust to making changes. Gameplay is one of the most dynamic and unstable aspects of any game project, so it is wise to select a data format that supports rapid development and is robust to frequent changes. As such, the binary object image format is not usually a good choice for storing game object data (although this format can be suitable for more stable data structures, like mesh data or collision geometry).

16.3.2 Serialized Game Object Descriptions

Serialization is another means of storing a representation of a game object's internal state to a disk file. This approach tends to be more portable and simpler to implement than the binary object image technique. To serialize an object out to disk, the object is asked to produce a stream of data that contains enough detail to permit the original object to be reconstructed later. When an object is serialized back into memory from disk, an instance of the appropriate class is created, and then the stream of attribute data is read in order to initialize the new object's internal state. If the original serialized data stream was complete, the new object should be identical to the original for all intents and purposes.

Serialization is supported natively by some programming languages. For example, C# and Java both provide standardized mechanisms for serializing object instances to and from an XML text format. The C++ language unfortunately does not provide a standardized serialization facility. However, many C++ serialization systems have been successfully built, both inside and out-

游戏行业。我们不会在这里详细介绍如何编写 C++ 对象序列化系统，但我们会描述数据格式以及为了使序列化在 C++ 中正常工作而需要编写的一些主要系统。

序列化数据并非对象的二进制映像。相反，它通常以更便捷、更易于移植的格式存储。XML 是一种流行的对象序列化格式，因为它支持良好且标准化，具有一定的可读性，并且对层级数据结构有出色的支持，而层级数据结构在序列化相互关联的游戏对象集合时经常出现。然而，XML 的解析速度非常慢，这会增加世界区块的加载时间。因此，一些游戏引擎使用专有的二进制格式，这种格式比 XML 文本解析速度更快、更紧凑。

许多游戏引擎（以及非游戏对象序列化系统）已转向使用基于文本的 JSON 数据格式 (http://www.json.org) 来替代 XML。JSON 也广泛用于万维网上的数据通信。例如，Facebook API 就完全使用 JSON 进行通信。

将对象序列化到磁盘和从磁盘序列化的机制通常以两种基本方式之一实现：

- 我们可以在基类中引入一对称为 SerializeOut() 和 SerializeIn() 之类的虚拟函数，并安排每个派生类提供它们的自定义实现，以“知道”如何序列化该特定类的属性。

- 我们可以为 C++ 类实现一个反射系统。然后，我们可以编写一个通用系统，自动序列化任何具有反射信息的 C++ 对象。

反射是 C# 等语言中使用的术语。简而言之，反射数据是对类内容的运行时描述。它存储了类的名称、包含的数据成员、每个数据成员的类型以及每个成员在对象内存映像中的偏移量等信息，还包含类所有成员函数的信息。给定任意 C++ 类的反射信息，我们就可以很容易地编写一个通用的对象序列化系统。

C++ 反射系统最棘手的部分是为所有相关类生成反射数据。这可以通过将类的数据成员封装在 #define 宏中来实现，这些宏通过提供一个虚函数来提取相关的反射信息，该虚函数可以被每个派生类重写。

side the game industry. We won't get into all the details of how to write a C++ object serialization system here, but we'll describe the data format and a few of the main systems that need to be written in order to get serialization to work in C++.

Serialization data isn't a binary image of the object. Instead, it is usually stored in a more-convenient and more-portable format. XML is a popular format for object serialization because it is well-supported and standardized, it is somewhat human-readable and it has excellent support for hierarchical data structures, which arise frequently when serializing collections of interrelated game objects. Unfortunately, XML is notoriously slow to parse, which can increase world chunk load times. For this reason, some game engines use a proprietary binary format that is faster to parse and more compact than XML text.

Many game engines (and non-game object serialization systems) have turned to the text-based JSON data format (http://www.json.org) as an alternative to XML. JSON is also used ubiquitously for data communication over the World Wide Web. For example, the Facebook API communicates exclusively using JSON.

The mechanics of serializing an object to and from disk are usually implemented in one of two basic ways:

- We can introduce a pair of virtual functions called something like `SerializeOut()` and `SerializeIn()` in our base class and arrange for each derived class to provide custom implementations of them that "know" how to serialize the attributes of that particular class.
- We can implement a *reflection* system for our C++ classes. We can then write a generic system that can automatically serialize any C++ object for which reflection information is available.

Reflection is a term used by the C# language, among others. In a nutshell, reflection data is a runtime description of the contents of a class. It stores information about the name of the class, what data members it contains, the types of each data member and the offset of each member within the object's memory image, and it also contains information about all of the class's member functions. Given reflection information for an arbitrary C++ class, we could quite easily write a general-purpose object serialization system.

The tricky part of a C++ reflection system is generating the reflection data for all of the relevant classes. This can be done by encapsulating a class's data members in `#define` macros that extract relevant reflection information by providing a virtual function that can be overridden by each derived class in

为了返回该类的适当反射数据，通过为每个类手动编码反射数据结构，或通过其他一些创造性的方法。

除了属性信息之外，序列化数据流通常包含每个对象的类或类型的名称或唯一 ID。类 ID 用于在对象从磁盘序列化到内存时实例化相应的类。类 ID 可以存储为字符串、散列字符串 ID 或其他类型的唯一 ID。

遗憾的是，C++ 无法仅通过字符串或 ID 来实例化一个类。类名必须在编译时已知，因此必须由程序员进行硬编码（例如，new ConcreteClass）。为了突破语言的这一限制，C++ 对象序列化系统总是包含某种类型的类工厂。工厂可以通过多种方式实现，但最简单的方法是创建一个数据表，将每个类名/ID 映射到某种函数或函子对象，这些函数或函子对象已被硬编码用于实例化该特定类。给定类名或 ID，我们只需在表中查找相应的函数或函子并调用它来实例化该类。

16.3.3 生成器和类型模式

二进制对象图像和序列化格式都存在致命弱点。它们都由所存储游戏对象类型的运行时实现定义，因此都要求世界编辑器对游戏引擎的运行时实现有深入的了解。例如，为了让世界编辑器输出异构游戏对象集合的二进制图像，它必须直接链接到运行时游戏引擎代码，或者必须精心手工编码以生成与游戏对象在运行时的数据布局完全匹配的字节块。序列化数据与游戏对象的实现耦合度较低，但同样，世界编辑器要么需要链接到运行时游戏对象代码才能访问类的 SerializeIn() 和 SerializeOut() 函数，要么需要以某种方式访问□□类的反射信息。

通过以独立于实现的方式抽象游戏对象的描述，可以打破游戏世界编辑器和运行时引擎代码之间的耦合。对于世界块数据文件中的每个游戏对象，我们存储一小块数据，通常称为 生成器 。生成器是游戏对象的轻量级、纯数据表示，可用于在运行时实例化和初始化该游戏对象。它包含游戏对象工具端类型的 id 。它还包含一个描述游戏对象初始属性的简单键值对表。这些属性

order to return appropriate reflection data for that class, by hand-coding a reflection data structure for each class, or via some other inventive approach.

In addition to attribute information, the serialization data stream invariably includes the name or unique id of each object's *class* or *type*. The class id is used to instantiate the appropriate class when the object is serialized into memory from disk. A class id can be stored as a string, a hashed string id, or some other kind of unique id.

Unfortunately, C++ provides no way to instantiate a class given only its name as a string or id. The class name must be known at compile time, and so it must be hard-coded by a programmer (e.g., `new ConcreteClass`). To work around this limitation of the language, C++ object serialization systems invariably include a *class factory* of some kind. A factory can be implemented in any number of ways, but the simplest approach is to create a data table that maps each class name/id to some kind of function or functor object that has been hard-coded to instantiate that particular class. Given a class name or id, we simply look up the corresponding function or functor in the table and call it to instantiate the class.

16.3.3 Spawners and Type Schemas

Both binary object images and serialization formats have an Achilles heel. They are both defined by the runtime implementation of the game object types they store, and hence they both require the world editor to contain intimate knowledge of the game engine's runtime implementation. For example, in order for the world editor to write out a binary image of a heterogeneous collection of game objects, it must either link directly with the runtime game engine code, or it must be painstakingly hand-coded to produce blocks of bytes that exactly match the data layout of the game objects at runtime. Serialization data is less-tightly coupled to the game object's implementation, but again, the world editor either needs to link with runtime game object code in order to gain access to the classes' `SerializeIn()` and `SerializeOut()` functions, or it needs access to the classes' reflection information in some way.

The coupling between the game world editor and the runtime engine code can be broken by abstracting the descriptions of our game objects in an implementation-independent way. For each game object in a world chunk data file, we store a little block of data, often called a *spawner*. A spawner is a lightweight, data-only representation of a game object that can be used to instantiate and initialize that game object at runtime. It contains the id of the game object's tool-side *type*. It also contains a table of simple key-value pairs that describe the initial attributes of the game object. These attributes

通常包含模型到世界的变换，因为大多数游戏对象在世界空间中具有不同的位置、方向和比例。当游戏对象被生成时，会根据生成器的类型实例化相应的类。然后，这些运行时对象可以查询键值对字典，以便正确地初始化其数据成员。

生成器可以配置为在加载后立即生成其游戏对象，也可以使其处于休眠状态，直到游戏后期被要求生成。生成器可以实现为一级对象，因此它们可以拥有便捷的函数式接口，并且除了对象属性之外，还可以存储有用的元数据。生成器甚至可以用于生成游戏对象以外的其他用途。例如，在顽皮狗引擎中，设计师使用生成器来定义游戏世界中的重要点或坐标轴。这些生成器被称为位置生成器或定位器生成器。定位器在游戏中有很多用途，例如：

- 定义 AI 角色的兴趣点，
- 定义一组坐标轴，相对于该坐标轴，可以完美同步播放一组动画，
- 定义粒子效果或音频效果的起源位置，
- 定义赛道沿线的航点，

等等。

16.3.3.1 对象类型模式

游戏对象的属性和行为由其类型定义。在采用基于生成器的设计的游戏世界编辑器中，游戏对象类型可以用数据驱动的模式表示，该模式定义了用户在创建或编辑该类型对象时应看到的属性集合。在运行时，工具端对象类型可以以硬编码或数据驱动的方式映射到一个类或类集合，这些类或类必须实例化才能生成给定类型的游戏对象。

类型模式可以存储在一个简单的文本文件中，供世界编辑器使用，并供用户检查和编辑。例如，一个模式文件可能如下所示：

```
enum LightType
{
    Ambient, Directional, Point, Spot
}
```


often include a model-to-world transform, since most game objects have a distinct position, orientation and scale in world space. When the game object is spawned, the appropriate class or classes are instantiated, as determined by the spawner's type. These runtime objects can then consult the dictionary of key-value pairs in order to initialize their data members appropriately.

A spawner can be configured to spawn its game object immediately upon being loaded, or it can lie dormant until asked to spawn at some later time during the game. Spawners can be implemented as first-class objects, so they can have a convenient functional interface and can store useful metadata in addition to object attributes. A spawner can even be used for purposes other than spawning game objects. For example, in the Naughty Dog engine, designers used spawners to define important points or coordinate axes in the game world. These were called *position spawners* or *locator spawners*. Locators have many uses in a game, such as:

- defining points of interest for an AI character,
- defining a set of coordinate axes relative to which a set of animations can be played in perfect synchronization,
- defining the location at which a particle effect or audio effect should originate,
- defining waypoints along a race track,

and the list goes on.

16.3.3.1 Object Type Schemas

A game object's attributes and behaviors are defined by its type. In a game world editor that employs a spawner-based design, a game object type can be represented by a data-driven *schema* that defines the collection of attributes that should be visible to the user when creating or editing an object of that type. At runtime, the tool-side object type can be mapped in either a hard-coded or data-driven way to a class or collection of classes that must be instantiated in order to spawn a game object of the given type.

Type schemas can be stored in a simple text file for consumption by the world editor and for inspection and editing by its users. For example, a schema file might look something like this:

```
enum LightType
{
    Ambient, Directional, Point, Spot
}
```

```
type Light
{
    String      UniqueId;
    LightType   Type;
    Vector      Pos;
    Quaternion  Rot;
    Float       Intensity : min(0.0), max(1.0);
    ColorARGB   DiffuseColor;
    ColorARGB   SpecularColor;
    // ...
}

type Vehicle
{
    String         UniqueId;
    Vector         Pos;
    Quaternion     Rot;
    MeshReference  Mesh;
    Int            NumWheels : min(2), max(4);
    Float          TurnRadius;
    Float          TopSpeed : min(0.0);
    // ...
}

//...
```

上面的例子揭示了一些重要的细节。你会注意到，除了名称之外，每个属性的数据类型都已定义。这些数据类型可以是简单类型，例如字符串、整数和浮点值，也可以是特殊类型，例如向量、四元数、ARGB 颜色，或者对特殊资源类型（例如网格、碰撞数据等）的引用。在这个例子中，我们甚至提供了一种定义枚举类型的机制，例如 LightType 。另一个微妙之处是，对象类型模式为世界编辑器提供了额外的信息，例如编辑属性时要使用的 GUI 元素类型。有时，属性的 GUI 要求由其数据类型暗示——字符串通常使用文本字段编辑，布尔值通过复选框编辑，向量通过三个分别代表 x、y 和 z 坐标的文本字段编辑，或者可能通过专门为在 3D 环境中操作向量而设计的 GUI 元素编辑。该模式还可以指定 GUI 使用的元信息，例如整数和浮点属性的最小和最大允许值、下拉组合框的可用选项列表等等。

一些游戏引擎允许继承对象类型模式，就像


```
type Light
{
    String      UniqueId;
    LightType   Type;
    Vector      Pos;
    Quaternion  Rot;
    Float       Intensity : min(0.0), max(1.0);
    ColorARGB   DiffuseColor;
    ColorARGB   SpecularColor;
    // ...
}

type Vehicle
{
    String         UniqueId;
    Vector         Pos;
    Quaternion     Rot;
    MeshReference  Mesh;
    Int            NumWheels : min(2), max(4);
    Float          TurnRadius;
    Float          TopSpeed : min(0.0);
    // ...
}

//...
```

The above example brings a few important details to light. You'll notice that the *data types* of each attribute are defined, in addition to their names. These can be simple types like strings, integers and floating-point values, or they can be specialized types like vectors, quaternions, ARGB colors, or references to special asset types like meshes, collision data and so on. In this example, we've even provided a mechanism for defining enumerated types, like `LightType`. Another subtle point is that the object type schema provides additional information to the world editor, such as what type of GUI element to use when editing the attribute. Sometimes an attribute's GUI requirements are implied by its data type—strings are generally edited with a text field, Booleans via a check box, vectors via three text fields for the *x*-, *y*- and *z*-coordinates or perhaps via a specialized GUI element designed for manipulating vectors in 3D. The schema can also specify meta-information for use by the GUI, such as minimum and maximum allowable values for integer and floating-point attributes, lists of available choices for drop-down combo boxes and so on.

Some game engines permit object type schemas to be inherited, much like

类。例如，每个游戏对象都需要知道自己的类型，并且必须有一个唯一的 ID，以便在运行时将其与所有其他游戏对象区分开来。这些属性可以在顶级架构中指定，所有其他架构都从该架构派生而来。

16.3.3.2 默认属性值

可以想象，典型的游戏对象架构中的属性数量可能会非常庞大□□。这意味着游戏设计师必须为其放置在游戏世界中的每种游戏对象类型的每个实例指定大量数据。在架构中为许多属性定义默认值将非常有帮助。这使得游戏设计师可以轻松地放置游戏对象类型的“原始”实例，同时仍然允许其根据需要微调特定实例的属性值。

当特定属性的默认值发生变化时，默认值的一个固有问题就出现了。例如，我们的游戏设计师最初可能希望兽人拥有 20 点生命值。经过数月的制作，设计师可能会决定兽人应该更强大，默认拥有 30 点生命值。除非另有说明，否则任何新放入游戏世界的兽人现在都将拥有 30 点生命值。但是，在更改之前放入游戏世界区块的所有兽人该怎么办呢？我们是否需要找到所有这些之前创建的兽人，并手动将其生命值更改为 30 点？

理想情况下，我们希望设计我们的生成器系统，使默认值的更改能够自动传播到所有尚未明确覆盖其默认值的现有实例。实现此功能的一种简单方法是，直接省略与默认值相同的属性的键值对。当生成器缺少某个属性时，可以使用相应的默认值。（这假设游戏引擎可以访问对象类型架构文件，以便读取属性的默认值。要么是引擎本身，要么是工具本身——在这种情况下，传播新的默认值只需简单地重建所有受更改影响的世界区块即可。）在我们的示例中，大多数现有的兽人生成器根本没有 HitPoints 键值对（当然，除非生成器的生命值已被手动更改为默认值）。因此，当默认值从 20 更改为 30 时，这些兽人将自动使用新值。

一些引擎允许在派生对象类型中覆盖默认值。例如，名为 Vehicle 的类型的架构可能将默认 TopSpeed 定义为 80 英里/小时。派生的 Motorcycle 类型架构可以将此 TopSpeed 覆盖为 100 英里/小时。

classes. For example, every game object needs to know its *type* and must have a *unique id* so that it can be distinguished from all the other game objects at runtime. These attributes could be specified in a top-level schema, from which all other schemas are derived.

16.3.3.2 Default Attribute Values

As you can well imagine, the number of attributes in a typical game object schema can grow quite large. This translates into a lot of data that must be specified by the game designer for each instance of each game object type he or she places into the game world. It can be extremely helpful to define *default values* in the schema for many of the attributes. This permits game designers to place “vanilla” instances of a game object type with little effort but still permits him or her to fine-tune the attribute values on specific instances as needed.

One inherent problem with default values arises when the default value of a particular attribute changes. For example, our game designers might have originally wanted Orcs to have 20 hit points. After many months of production, the designers might decide that Orcs should be more powerful and have 30 hit points by default. Any new Orcs placed into a game world will now have 30 hit points unless otherwise specified. But what about all the Orcs that were placed into game world chunks prior to the change? Do we need to find all of these previously created Orcs and manually change their hit points to 30?

Ideally, we’d like to design our spawner system so that changes in default values automatically propagate to all preexisting instances that have not had their default values overridden explicitly. One easy way to implement this feature is to simply omit key-value pairs for attributes whose value does not differ from the default value. Whenever an attribute is missing from the spawner, the appropriate default can be used. (This presumes that the game engine has access to the object type schema file, so that it can read in the attributes’ default values. Either that or the tool can do it—in which case, propagating new default values requires a simple rebuild of all world chunk(s) affected by the change.) In our example, most of the preexisting Orc spawners would have had no `HitPoints` key-value pair at all (unless of course one of the spawner’s hit points had been changed from the default value manually). So when the default value changes from 20 to 30, these Orcs will automatically use the new value.

Some engines allow default values to be overridden in derived object types. For example, the schema for a type called `Vehicle` might define a default `TopSpeed` of 80 miles per hour. A derived `Motorcycle` type schema could override this `TopSpeed` to be 100 miles per hour.

16.3.3.3 生成器和类型模式的一些好处

将生成器与游戏对象实现分离的主要好处是简单、灵活和健壮。从数据管理的角度来看，处理键值对表比管理带有指针修复或自定义序列化对象格式的二进制对象图像要简单得多。键值对方法还使数据格式非常灵活且易于更改。如果游戏对象遇到不希望看到的键值对，它可以简单地忽略它们。同样，如果游戏对象无法找到所需的键值对，它可以选择使用默认值。这使得键值对数据格式对于设计人员和程序员所做的更改都非常稳健。

生成器还简化了游戏世界编辑器的设计和实现，因为它只需要知道如何管理键值对列表和对象类型模式。它不需要以任何方式与运行时游戏引擎共享代码，并且与引擎的实现细节的耦合度非常低。

生成器和原型赋予游戏设计师和程序员极大的灵活性和强大功能。设计师可以在世界编辑器中定义新的游戏对象类型模式，几乎无需程序员干预。程序员可以在时间允许的情况下，随时实现这些新对象类型的运行时实现。程序员无需在添加每个新对象类型时立即提供其实现，以免破坏游戏。无论是否带有运行时实现，新对象数据都可以安全地存在于世界区块文件中，并且运行时实现也可以存在于世界区块文件中，无论是否带有相应的数据。

16.4 加载和流式传输游戏世界

为了弥合离线世界编辑器和运行时游戏对象模型之间的差距，我们需要一种方法将世界块加载到内存中，并在不再需要它们时卸载它们。游戏世界加载系统主要负责两件事：管理将游戏世界块和其他所需资源从磁盘加载到内存所需的文件 I/O，以及管理这些资源的内存分配和释放。引擎还需要管理游戏对象在游戏中的生成和销毁，包括为对象分配和释放内存，以及确保为每个游戏对象实例化正确的类。在接下来的章节中，我们将探讨游戏世界是如何

16.3.3.3 Some Beneifts of Spawners and Type Schemas

The key benefits of separating the spawner from the implementation of the game object are *simplicity*, *flexibility* and *robustness*. From a data management point of view, it is much simpler to deal with a table of key-value pairs than it is to manage a binary object image with pointer fix-ups or a custom serialized object format. The key-value pairs approach also makes the data format extremely flexible and robust to changes. If a game object encounters key-value pairs that it is not expecting to see, it can simply ignore them. Likewise, if the game object is unable to find a key-value pair that it needs, it has the option of using a default value instead. This makes a key-value pair data format extremely robust to changes made by both the designers and the programmers.

Spawners also simplify the design and implementation of the game world editor, because it only needs to know how to manage lists of key-value pairs and object type schemas. It doesn't need to share code with the runtime game engine in any way, and it is only very loosely coupled to the engine's implementation details.

Spawners and archetypes give game designers and programmers a great deal of flexibility and power. Designers can define new game object type schemas within the world editor with little or no programmer intervention. The programmer can implement the runtime implementation of these new object types whenever his or her schedule allows it. The programmer does not need to immediately provide an implementation of each new object type as it is added in order to avoid breaking the game. New object data can exist safely in the world chunk files with or without a runtime implementation, and runtime implementations can exist with or without corresponding data in the world chunk file.

16.4 Loading and Streaming Game Worlds

To bridge the gap between the offline world editor and our runtime game object model, we need a way to load world chunks into memory and unload them when they are no longer needed. The game world loading system has two main responsibilities: to manage the file I/O necessary to load game world chunks and other needed assets from disk into memory and to manage the allocation and deallocation of memory for these resources. The engine also needs to manage the *spawning* and *destruction* of game objects as they come and go in the game, both in terms of allocating and deallocating memory for the objects and ensuring that the proper classes are instantiated for each game object. In the following sections, we'll investigate how game worlds

已加载，并查看对象生成系统通常如何工作。

16.4.1 简单关卡加载

最直接的游戏世界加载方式，也是所有早期游戏都采用的方式，是每次只允许加载一个游戏世界区块（也就是关卡）。游戏首次启动时，以及在两个关卡之间，玩家在等待关卡加载时，会看到一个静态或简单的二维动画加载画面。

这种设计中的内存管理非常简单。正如我们在 7.2.2.7 节中提到的，基于堆栈的分配器非常适合一次加载一个关卡的世界加载设计。游戏首次运行时，所有游戏关卡所需的资源数据都会加载到堆栈底部。为了便于讨论，我们将这些资源数据称为“加载并驻留资源”（LSR）。堆栈指针的位置会在 LSR 资源完全加载后记录下来。每个游戏世界数据块及其相关的网格、纹理、音频、动画和其他资源数据都会加载到堆栈中 LSR 资源的顶部。当玩家完成关卡后，只需将堆栈指针重置到 LSR 资源块的顶部即可释放所有内存。此时，可以在其位置加载新的关卡。如图 16.11 所示。

虽然这种设计非常简单，但它也存在一些缺点。首先，玩家只能以离散的区块形式看到游戏世界——这种技术无法实现一个广阔、连续、无缝的世界。另一个问题是，在加载关卡资源数据时，内存中没有游戏世界。因此，玩家只能看到某种二维的加载画面。

16.4.2 走向无缝装载：气锁

避免无聊的关卡加载画面的最佳方法是允许玩家在加载下一个世界区块及其相关资源数据时继续游戏。一种简单的方法是将我们为游戏世界资源预留的内存分成两个大小相等的块。我们可以将关卡 A 加载到一个内存块中，允许玩家开始玩关卡 A，然后使用流文件 I/O 库将关卡 B 加载到另一个内存块中（即，加载代码将在单独的线程中运行）。这种技术的最大问题是，与一次加载一个关卡的方法相比，它将每个关卡的大小减少了一半。

are loaded and also have a look at how object spawning systems typically work.

16.4.1 Simple Level Loading

The most straightforward game world loading approach, and the one used by all of the earliest games, is to allow one and only one game world chunk (a.k.a. level) to be loaded at a time. When the game is first started, and between pairs of levels, the player sees a static or simply animated two-dimensional loading screen while he or she waits for the level to load.

Memory management in this kind of design is quite straightforward. As we mentioned in Section 7.2.2.7, a stack-based allocator is very well-suited to a one-level-at-a-time world loading design. When the game first runs, any resource data that is required across all game levels is loaded at the bottom of the stack. We'll call these *load and stay resident* assets (LSR) for the purposes of this discussion. The location of the stack pointer is recorded after the LSR assets have been fully loaded. Each game world chunk, along with its associated mesh, texture, audio, animation and other resource data, is loaded on top of the LSR assets on the stack. When the level has been completed by the player, all of its memory can be freed by simply resetting the stack pointer to the top of the LSR asset block. At this point, a new level can be loaded in its place. This is illustrated in Figure 16.11.

While this design is very simple, it has a number of drawbacks. For one thing, the player only sees the game world in discrete chunks—there is no way to implement a vast, contiguous, seamless world using this technique. Another problem is that during the time the level's resource data is being loaded, there is no game world in memory. So, the player is forced to watch a two-dimensional loading screen of some sort.

16.4.2 Toward Seamless Loading: Air Locks

The best way to avoid boring level-loading screens is to permit the player to continue playing the game while the next world chunk and its associated resource data are being loaded. One simple approach would be to divide the memory that we've set aside for game world assets into two equally sized blocks. We could load level A into one memory block, allow the player to start playing level A and then load level B into the other block using a streaming file I/O library (i.e., the loading code would run in a separate thread). The big problem with this technique is that it cuts the size of each level in half relative to what would be possible with a one-level-at-a-time approach.

图 16.11。基于堆栈的内存分配器非常适合一次一个级别的世界加载系统。

我们可以通过将游戏世界内存划分成两个大小不等的区块来实现类似的效果——一个大区块可以容纳一个“完整的”游戏世界区块，另一个小区块只能容纳一个很小的世界区块。小区块有时被称为“气锁”。

游戏开始时，会加载一个“完整”块和一个“气锁”块。玩家穿过完整的区块并进入气闸舱，此时某种门或其他障碍物会确保玩家既无法看到之前的完整世界区域，也无法返回。之后，完整的区块可以被卸载，并加载一个新的完整大小的世界区块。在加载过程中，玩家需要在气闸舱内执行一些任务。这些任务可能很简单，比如从走廊的一端走到另一端，也可能更具吸引力，比如解谜或与敌人战斗。

Figure 16.11. A stack-based memory allocator is extremely well-suited to a one-level-at-a-time world loading system.

We can achieve a similar effect by dividing the game world memory into two unequally sized blocks—a large block that can contain a “full” game world chunk and a small block that is only large enough to contain a tiny world chunk. The small chunk is sometimes known as an “air lock.”

When the game starts, a “full” chunk and an “air lock” chunk are loaded. The player progresses through the full chunk and into the air lock, at which point some kind of gate or other impediment ensures that the player can neither see the previous full world area nor return to it. The full chunk can then be un-loaded, and a new full-sized world chunk can be loaded. During the load, the player is kept busy doing some task within the air lock. The task might be as simple as walking from one end of a hallway to the other, or it could be something more engaging, like solving a puzzle or fighting some enemies.

异步文件 I/O 使得玩家在气闸区域游玩时，能够加载完整的世界区块。更多详情请参阅第 7.1.3 节。需要注意的是，气闸系统并不能免除我们在每次启动新游戏时显示加载画面的麻烦，因为在初始加载期间，内存中没有可供游玩的游戏世界。然而，一旦玩家进入游戏世界，由于气闸和异步数据加载，他或她就无需再看到加载画面。

Xbox 版《光环》采用了类似的技术。广阔的世界区域总是由更小、更封闭的区域连接。在玩《光环》时，要注意那些阻碍你回溯的封闭区域——大约每 5-10 分钟你就会发现一个。PlayStation 2 版《杰克 2》也使用了气锁技术。游戏世界的结构是一个枢纽区域（主城），以及一些分支区域，每个分支区域都通过一个小型的封闭气锁区域与枢纽区域相连。

16.4.3 游戏世界流媒体

许多游戏设计都希望玩家感觉自己置身于一个巨大、连贯、无缝的世界。理想情况下，玩家不应该被周期性地限制在狭小的气闸区域——最好是世界尽可能自然、可信地在玩家面前展开。

现代游戏引擎通过使用一种名为“流式传输”的技术来支持这种无缝世界。世界流式传输可以通过多种方式实现。其主要目标始终是：(a) 在玩家进行常规游戏任务时加载数据；(b) 以消除碎片的方式管理内存，同时允许在玩家在游戏世界中前进时根据需要加载和卸载数据。

最近的主机和 PC 比之前的主机拥有更大的内存，因此现在可以同时在内存中保存多个世界区块。我们可以设想将内存空间划分成三个大小相等的缓冲区。首先，我们将世界区块 A、B 和 C 加载到这三个缓冲区中，并允许玩家从区块 A 开始游戏。当玩家进入区块 B 并且已经深入到无法再看到区块 A 时，我们可以卸载区块 A，并开始将新的区块 D 加载到第一个缓冲区中。当无法再看到区块 B 时，我们可以将其转储并加载区块 E。这种缓冲区的循环利用可以持续到玩家到达连续游戏世界的尽头。

粗粒度的世界流方法的问题在于，它对世界块的大小施加了严格的限制。整个游戏中的所有块必须大致相同——大到足以填满我们三个内存缓冲区中的大部分，但不能更大。

Asynchronous file I/O is what enables the full world chunk to be loaded while the player is simultaneously playing in the air lock region. See Section 7.1.3 for more details. It's important to note that an air lock system does *not* free us from displaying a loading screen whenever a new game is started, because during the initial load there is no game world in memory in which to play. However, once the player is in the game world, he or she needn't see a loading screen ever again, thanks to air locks and asynchronous data loading.

Halo for the Xbox used a technique similar to this. The large world areas were invariably connected by smaller, more confined areas. As you play *Halo*, watch for confined areas that prevent you from back-tracking—you'll find one roughly every 5–10 minutes of gameplay. *Jak 2* for the PlayStation 2 used the air lock technique as well. The game world was structured as a hub area (the main city) with a number of offshoot areas, each of which was connected to the hub via a small, confined air lock region.

16.4.3 Game World Streaming

Many game designs call for the player to feel like he or she is playing in a huge, contiguous, seamless world. Ideally, the player should not be confined to small air lock regions periodically—it would be best if the world simply unfolded in front of the player as naturally and believably as possible.

Modern game engines support this kind of seamless world by using a technique known as *streaming*. World streaming can be accomplished in various ways. The main goals are always (a) to load data while the player is engaged in regular gameplay tasks and (b) to manage the memory in such a way as to eliminate *fragmentation* while permitting data to be loaded and unloaded as needed as the player progresses through the game world.

Recent consoles and PCs have a lot more memory than their predecessors, so it is now possible to keep multiple world chunks in memory simultaneously. We could imagine dividing our memory space into, say, three equally sized buffers. At first, we load world chunks A, B and C into these three buffers and allow the player to start playing through chunk A. When he or she enters chunk B and is far enough along that chunk A can no longer be seen, we can unload chunk A and start loading a new chunk D into the first buffer. When B can no longer be seen, it can be dumped and chunk E loaded. This recycling of buffers can continue until the player has reached the end of the contiguous game world.

The problem with a coarse-grained approach to world streaming is that it places onerous restrictions on the size of a world chunk. All chunks in the entire game must be roughly the same size—large enough to fill up the majority of one of our three memory buffers but never any larger.

解决这个问题的一种方法是采用更细粒度的内存细分。与其将相对较大的世界数据块流式传输，不如将每个游戏资源（从游戏世界数据块到前景网格、纹理到动画库）划分成大小相等的数据块。然后，我们可以使用基于池的块状内存分配系统（如第 7.2.2.7 节中所述）来根据需要加载和卸载资源数据，而无需担心内存碎片。这本质上就是顽皮狗引擎所采用的技术。（尽管顽皮狗的实现也采用了一些复杂的技术来利用未满数据块末尾原本未使用的空间。）

16.4.3.1 确定要加载哪些资源

使用细粒度块状内存分配器进行世界流传输时，一个问题是引擎如何知道在游戏过程中的任何特定时刻需要加载哪些资源。在顽皮狗引擎中，我们使用一个相对简单的关卡加载区域系统来控制资源的加载和卸载。

所有《神秘海域》和《最后生还者》系列游戏都设定在多个地理位置不同、连续的游戏世界中。例如，《神秘海域：德雷克的宝藏》的故事发生在丛林和岛屿上。每个世界都存在于一个单一、一致的世界空间中，但它们被划分为许多地理上相邻的区块。一个简单的凸形体积（称为区域）包围着每个区块；这些区域之间会有所重叠。每个区域都包含一个世界区块列表，当玩家位于该区域时，这些区块应该被保存在内存中。

在任何特定时刻，玩家都位于一个或多个这样的区域内。为了确定应该在内存中的世界区块集合，我们只需取包含 Nathan Drake 角色的每个区域的区块列表的并集即可。关卡加载系统会定期检查此主区块列表，并将其与当前内存中的世界区块集合进行比较。如果某个区块从主列表中消失，则会将其卸载，从而释放其占用的所有分配块。如果列表中出现了新的区块，则会将其加载到任何可以找到的空闲分配块中。关卡加载区域和世界区块的设计方式确保玩家在卸载时不会看到区块消失，并且从区块开始加载到玩家首次看到其内容之间有足够的时间，以允许该区块完全加载到内存中。图 16.12 演示了此技术。

One way around this problem is to employ a much finer-grained subdivision of memory. Rather than streaming relatively large chunks of the world, we can divide every game asset, from game world chunks to foreground meshes to textures to animation banks, into equally sized blocks of data. We can then use a chunky, pool-based memory allocation system like the one described in Section 7.2.2.7 to load and unload resource data as needed without having to worry about memory fragmentation. This is essentially the technique employed by Naughty Dog's engine. (Although Naughty Dog's implementation also employs some sophisticated techniques for making use of what would otherwise be unused space at the ends of under-full chunks.)

16.4.3.1 Determining Which Resources to Load

One question that arises when using a fine-grained chunky memory allocator for world streaming is how the engine will know what resources to load at any given moment during gameplay. In the Naughty Dog engine, we use a relatively simple system of *level load regions* to control the loading and unloading of assets.

All of the *Uncharted* and *The Last of Us* games are set in multiple, geographically distinct, contiguous game worlds. For example, *Uncharted: Drake's Fortune* takes place in a jungle and on an island. Each of these worlds exists in a single, consistent world space, but they are divided up into numerous geographically adjacent chunks. A simple convex volume known as a *region* encompasses each of the chunks; the regions overlap each other somewhat. Each region contains a list of the world chunks that should be in memory when the player is in that region.

At any given moment, the player is within one or more of these regions. To determine the set of world chunks that should be in memory, we simply take the *union* of the chunk lists from each of the regions enclosing the Nathan Drake character. The level loading system periodically checks this master chunk list and compares it against the set of world chunks that are currently in memory. If a chunk disappears from the master list, it is unloaded, thereby freeing up all of the allocation blocks it occupied. If a new chunk appears in the list, it is loaded into any free allocation blocks that can be found. The level load regions and world chunks are designed in such a way as to ensure that the player never sees a chunk disappear when it is unloaded and that there's enough time between the moment at which a chunk starts loading and the moment its contents are first seen by the player to permit the chunk to be fully streamed into memory. This technique is illustrated in Figure 16.12.

图 16.12。游戏世界被划分为多个区块。每个关卡加载区域都有一个请求的区块列表，这些区块的排列方式可以保证玩家不会看到任何区块弹出或消失。

16.4.3.2 PlayStation 4 上的 PlayGo

PlayStation 4 新增了一项名为 PlayGo 的功能，让下载游戏（而非购买蓝光光碟）的过程比以往轻松许多。PlayGo 的工作原理是，只下载游玩游戏第一部分所需的最小数据子集。PS4 会在后台下载剩余的游戏内容，同时玩家可以继续畅玩游戏，不会受到任何干扰。为了实现此功能，游戏必须支持无缝关卡串流，正如我们上文所述。

16.4.4 对象生成的内存管理

一旦游戏世界加载到内存中，我们就需要管理动态游戏对象的生成过程。大多数游戏引擎都具有某种游戏对象生成系统，用于管理构成每个游戏对象的类的实例化，并在不再需要游戏对象时处理它们的销毁。任何对象生成系统的核心工作之一就是管理新生成的游戏对象的动态内存分配。动态分配可能很慢，因此必须采取措施确保分配尽可能高效。而且，由于游戏对象的大小各不相同，动态分配它们可能会导致内存碎片化，从而导致过早出现内存不足的情况。游戏对象内存管理有许多不同的方法。我们将在以下章节中探讨一些常见的方法。

16.4.4.1 对象生成的离线内存分配

一些游戏引擎以一种相当严苛的方式解决了分配速度和内存碎片化的问题，即在游戏过程中完全禁止动态内存分配。这类引擎允许游戏世界

Figure 16.12. A game world divided into chunks. Level load regions, each with a requested chunk list, are arranged in such a way as to guarantee that the player never sees a chunk pop in or out of view.

16.4.3.2 PlayGo on the PlayStation 4

The PlayStation 4 includes a new feature called PlayGo that makes the process of downloading a game (as opposed to buying it on Blu-ray) a lot less painful than it has traditionally been. PlayGo works by downloading only the minimum subset of data required in order to play the first section of the game. The PS4 downloads the rest of the game's content in the background, while the player continues to experience the game without interruption. In order for this to work well, the game must of course support seamless level streaming, as we've described above.

16.4.4 Memory Management for Object Spawning

Once a game world has been loaded into memory, we need to manage the process of *spawning* the dynamic game objects in the world. Most game engines have some kind of game object spawning system that manages the instantiation of the class or classes that make up each game object and handles destruction of game objects when they are no longer needed. One of the central jobs of any object spawning system is to manage the dynamic allocation of memory for newly spawned game objects. Dynamic allocation can be slow, so steps must be taken to ensure allocations are as efficient as possible. And because game objects come in a wide variety of sizes, dynamically allocating them can cause memory to become fragmented, leading to premature out-of-memory conditions. There are a number of different approaches to game object memory management. We'll explore a few common ones in the following sections.

16.4.4.1 OffLine Memory Allocation for Object Spawning

Some game engines solve the problems of allocation speed and memory fragmentation in a rather draconian way, by simply disallowing dynamic memory allocation during gameplay altogether. Such engines permit game world

区块需要动态加载和卸载，但它们会在加载区块后立即在所有动态游戏对象中生成。此后，任何游戏对象都无法创建或销毁。你可以将此技术视为遵循“游戏对象守恒定律”。一旦世界区块加载完毕，就不会再创建或销毁任何游戏对象。

这种技术可以避免内存碎片化，因为世界区块中所有游戏对象的内存需求 (a) 是预先已知的，并且 (b) 是有界限的。这意味着游戏对象的内存可以由世界编辑器离线分配，并作为世界区块数据本身的一部分。因此，所有游戏对象都从用于加载游戏世界及其资源的同一块内存中分配，并且它们不会像任何其他已加载的资源数据那样更容易出现碎片化。这种方法的另一个好处是，使游戏的内存使用模式高度可预测。大量游戏对象意外生成到世界中并导致游戏内存耗尽的情况绝对不会发生。

不利的一面是，这种方法对于游戏设计师来说可能有很大的限制。动态对象生成可以通过在世界编辑器中分配游戏对象来模拟，但在世界首次加载时将其设置为不可见和休眠状态。之后，该对象可以通过简单地激活自身并使其可见来“生成”。但是，游戏设计师必须在世界编辑器中首次创建游戏世界时预测所需的每种类型的游戏对象总数。如果他们想为玩家提供无限量的医疗包、武器、敌人或其他类型的游戏对象，他们要么需要找到一种回收游戏对象的方法，要么就只能等着运气了。

16.4.4.2 对象生成的动态内存管理

游戏设计师可能更喜欢使用支持真正动态对象生成的游戏引擎。虽然这比静态游戏对象生成方法更难实现，但它可以通过多种不同的方式实现。

再次强调，主要问题是内存碎片。由于不同类型的游戏对象（有时甚至是同一类型对象的不同实例）占用的内存量不同，我们无法使用我们最喜欢的无碎片分配器——池分配器。而且由于游戏对象的销毁顺序通常与它们生成的顺序不同，我们也无法使用基于堆栈的分配器。我们唯一的选择似乎是容易产生碎片的堆分配器。值得庆幸的是，有很多方法可以解决碎片问题。我们将在以下章节中探讨一些常见的方法。

chunks to be loaded and unloaded dynamically, but they spawn in all dynamic game objects immediately upon loading a chunk. Thereafter, no game objects can be created or destroyed. You can think of this technique as obeying a “law of conservation of game objects.” No game objects are created or destroyed once a world chunk has been loaded.

This technique avoids memory fragmentation because the memory requirements of all the game objects in a world chunk are (a) known a priori and (b) bounded. This means that the memory for the game objects can be allocated offline by the world editor and included as part of the world chunk data itself. All game objects are therefore allocated out of the same memory used to load the game world and its resources, and they are no more prone to fragmentation than any other loaded resource data. This approach also has the benefit of making the game’s memory usage patterns highly predictable. There’s no chance that a large group of game objects is going to spawn into the world unexpectedly, and cause the game to run out of memory.

On the downside, this approach can be quite limiting for game designers. Dynamic object spawning can be simulated by allocating a game object in the world editor but instructing it to be invisible and dormant when the world is first loaded. Later, the object can “spawn” by simply activating itself and making itself visible. But the game designers have to predict the total number of game objects of each type that they’ll need when the game world is first created in the world editor. If they want to provide the player with an infinite supply of health packs, weapons, enemies or some other kind of game object, they either need to work out a way to recycle their game objects, or they’re out of luck.

16.4.4.2 Dynamic Memory Management for Object Spawning

Game designers would probably prefer to work with a game engine that supports true dynamic object spawning. Although this is more difficult to implement than a static game object spawning approach, it can be implemented in a number of different ways.

Again, the primary problem is memory fragmentation. Because different types of game objects (and sometimes even different instances of the same type of object) occupy different amounts of memory, we cannot use our favorite fragmentation-free allocator—the pool allocator. And because game objects are generally destroyed in a different order than that in which they were spawned, we cannot use a stack-based allocator either. Our only choice appears to be a fragmentation-prone heap allocator. Thankfully, there are many ways to deal with the fragmentation problem. We’ll investigate a few common ones in the following sections.

每个对象类型一个内存池

如果保证每种游戏对象类型的实例占用相同的内存量，我们可以考虑为每种对象类型使用单独的内存池。实际上，每种大小的游戏对象只需要一个内存池，因此相同大小的对象类型可以共享一个内存池。

这样做可以完全避免内存碎片，但这种方法的一个局限性在于我们需要维护大量独立的池。我们还需要对每种类型的对象数量进行合理的估算。如果池中的元素过多，最终会浪费内存；如果池中的元素过少，我们将无法在运行时满足所有生成请求，游戏对象将无法生成。

小内存分配器

我们可以将“每种游戏对象类型一个内存池”的思路，转变为更可行的方案，允许游戏对象从元素大小大于对象本身的内存池中分配。这可以显著减少我们需要的独立内存池数量，但代价是每个内存池中可能会浪费一些内存。

例如，我们可以创建一组池分配器，每个分配器的元素大小都是其前一个分配器的两倍——可能是 8、16、32、64、128、256 或 512 字节。我们也可以使用符合其他合适模式的元素大小序列，或者根据从正在运行的游戏中收集的分配统计数据来创建大小列表。

每当我们尝试分配游戏对象时，我们都会搜索元素大于或等于待分配对象大小的最小池。我们接受某些对象会浪费空间。作为回报，我们缓解了所有内存碎片问题——这是一个相当公平的权衡。如果我们遇到大于最大池大小的内存分配请求，我们始终可以将其交给通用堆分配器，因为我们知道大内存块的碎片远不如小内存块的碎片那么严重。

这种类型的分配器有时被称为小型内存分配器。它可以消除碎片（对于适合池的分配）。它还可以显著加快小块数据的内存分配速度，因为池分配只需两次指针操作即可从空闲元素链表中移除元素——这比通用堆分配操作的开销要小得多。

One Memory Pool per Object Type

If the individual instances of each game object type are all guaranteed to occupy the same amount of memory, we could consider using a separate memory pool for each object type. Actually, we only need one pool per *unique game object size*, so object types of the same size can share a single pool.

Doing this allows us to completely avoid memory fragmentation, but one limitation of this approach is that we need to maintain lots of separate pools. We also need to make educated guesses about how many of each type of object we'll need. If a pool has too many elements, we end up wasting memory; if it has too few, we won't be able to satisfy all of the spawn requests at runtime, and game objects will fail to spawn.

Small Memory Allocators

We can transform the idea of one pool per game object type into something more workable by allowing a game object to be allocated out of a pool whose elements are larger than the object itself. This can reduce the number of unique memory pools we need significantly, at the cost of some potentially wasted memory in each pool.

For example, we might create a set of pool allocators, each one with elements that are twice as large as those of its predecessor—perhaps 8, 16, 32, 64, 128, 256 and 512 bytes. We can also use a sequence of element sizes that conforms to some other suitable pattern or base the list of sizes on allocation statistics collected from the running game.

Whenever we try to allocate a game object, we search for the smallest pool whose elements are larger than or equal to the size of the object we're allocating. We accept that for some objects, we'll be wasting space. In return, we alleviate all of our memory fragmentation problems—a reasonably fair trade. If we ever encounter a memory allocation request that is larger than our largest pool, we can always turn it over to the general-purpose heap allocator, knowing that fragmentation of large memory blocks is not nearly as problematic as fragmentation involving tiny blocks.

This type of allocator is sometimes called a *small memory allocator*. It can eliminate fragmentation (for allocations that fit into one of the pools). It can also speed up memory allocations significantly for small chunks of data, because a pool allocation involves two pointer manipulations to remove the element from the linked list of free elements—a much less-expensive operation than a general-purpose heap allocation.

内存重定位

消除碎片的另一种方法是直接解决问题。这种方法称为内存重定位。它涉及将已分配的内存块向下移动到相邻的空闲空间以消除碎片。移动内存很容易，但由于我们移动的是“活跃的”已分配对象，因此需要非常小心地修复指向所移动内存块的任何指针。有关更多详细信息，请参阅第 6.2.2.2 节。

16.4.5 已保存的游戏

许多游戏允许玩家保存进度，退出游戏后稍后再以离开游戏时的状态重新加载游戏。保存游戏系统与世界区块加载系统类似，因为它能够从磁盘文件或存储卡加载游戏世界的状态。但该系统的要求与世界加载系统的要求略有不同，因此两者通常有所不同（或仅部分重叠）。

为了理解这两个系统需求之间的差异，让我们简要比较一下世界区块和游戏存档文件。世界区块指定了世界中所有动态对象的初始条件，同时也包含所有静态世界元素的完整描述。许多静态信息（例如背景网格和碰撞数据）往往会占用大量磁盘空间。因此，世界区块有时由多个磁盘文件组成，并且与世界区块相关的数据总量通常很大。

保存的游戏文件还必须存储世界中游戏对象的当前状态信息。但是，它无需存储任何可通过读取世界区块数据确定的信息的副本。例如，无需在保存的游戏文件中保存静态几何图形。保存的游戏也无需存储每个对象状态的所有细节。一些对游戏玩法没有影响的对象可以完全省略。对于其他游戏对象，我们可能只需要存储部分状态信息。只要玩家无法区分游戏世界在保存和重新加载前后的状态差异（或者这些差异与玩家无关），那么我们就拥有一个成功的保存游戏系统。因此，保存的游戏文件通常比世界区块文件小得多，并且可能更注重数据压缩和数据省略。当大量保存的游戏文件必须装入老式游戏机使用的微型存储卡时，小文件大小尤为重要。但即使在今天，有了配备大容量硬盘并连接到云保存系统的控制台，将保存的游戏文件的大小保持在尽可能小的范围内仍然是一个好主意

Memory Relocation

Another way to eliminate fragmentation is to attack the problem directly. This approach is known as *memory relocation*. It involves shifting allocated memory blocks down into adjacent free holes to remove fragmentation. Moving the memory is easy, but because we are moving “live” allocated objects, we need to be very careful about fixing up any pointers into the memory blocks we move. See Section 6.2.2.2 for more details.

16.4.5 Saved Games

Many games allow the player to save his or her progress, quit the game and then load up the game at a later time in exactly the state he or she left it. A saved game system is similar to the world chunk loading system in that it is capable of loading the state of the game world from a disk file or memory card. But the requirements of this system differ somewhat from those of a world loading system, so the two are usually distinct (or overlap only partially).

To understand the differences between the requirements of these two systems, let’s briefly compare world chunks to saved game files. World chunks specify the initial conditions of all dynamic objects in the world, but they also contain a full description of all static world elements. Much of the static information, such as background meshes and collision data, tends to take up a lot of disk space. As such, world chunks are sometimes comprised of multiple disk files, and the total amount of data associated with a world chunk is usually large.

A saved game file must also store the current state information of the game objects in the world. However, it does not need to store a duplicate copy of any information that can be determined by reading the world chunk data. For example, there’s no need to save out the static geometry in a saved game file. A saved game need not store every detail of every object’s state either. Some objects that have no impact on gameplay can be omitted altogether. For the other game objects, we may only need to store partial state information. As long as the player can’t tell the difference between the state of the game world before and after it has been saved and reloaded (or if the differences are irrelevant to the player), then we have a successful saved game system. As such, saved game files tend to be much smaller than world chunk files and may place more of an emphasis on data compression and omission. Small file sizes are especially important when numerous saved game files must fit onto the tiny memory cards that were used on older consoles. But even today, with consoles that are equippedwith large hard drives and linked to a cloud save system, it’s still a good idea to keep the size of a saved game file as small as

可能的。

16.4.5.1 检查点

保存游戏的一种方法是将保存限制在游戏中的特定点，称为检查点。这种方法的好处是，关于游戏状态的大部分知识都保存在每个检查点附近的当前世界块中。无论哪个玩家在玩游戏，这些数据始终完全相同，因此不需要存储在保存的游戏中。因此，基于检查点的游戏保存文件可以非常小。我们可能只需要存储最后到达的检查点的名称，再加上一些关于玩家角色当前状态的信息，比如玩家的健康状况、剩余生命数、他的库存中有哪些物品、他拥有哪些武器以及每个武器包含多少弹药。一些基于检查点的游戏甚至不存储这些信息——它们让玩家在每个检查点都处于已□□知状态。当然，基于检查点的游戏的缺点是可能会让用户感到沮丧，尤其是在检查点很少的情况下。

16.4.5.2 随处保存

有些游戏支持“随处保存”功能。顾名思义，这类游戏允许在游戏过程中的任意时间点保存游戏状态。为了实现此功能，游戏保存数据文件的大小必须显著增加。所有与游戏玩法相关的游戏对象的当前位置和内部状态都必须保存，并在稍后再次加载游戏时恢复。

在“随处保存”的设计中，游戏存档数据文件包含的信息与世界区块基本相同，只是少了世界的静态组件。虽然两个系统可以采用相同的数据格式，但可能存在一些因素使其不可行。例如，世界区块数据格式可能设计得更具灵活性，而游戏存档格式则可能经过压缩，以最小化每个游戏存档的大小。

正如我们之前提到的，减少游戏存档中所需数据量的一种方法是省略某些不相关的游戏对象，并省略其他一些不相关的细节。例如，我们不需要记住当前正在播放的每个动画的精确时间索引，也不需要记住每个物理模拟刚体的精确动量和速度。我们可以依靠人类玩家不完美的记忆，只保存游戏状态的粗略近似值。

possible.

16.4.5.1 Checkpoints

One approach to save games is to limit saves to specific points in the game, known as *checkpoints*. The benefit of this approach is that most of the knowledge about the state of the game is saved in the current world chunk(s) in the vicinity of each checkpoint. This data is always exactly the same, no matter which player is playing the game, so it needn't be stored in the saved game. As a result, saved game files based on checkpoints can be extremely small. We might need to store only the name of the last checkpoint reached, plus perhaps some information about the current state of the player character, such as the player's health, number of lives remaining, what items he has in his inventory, which weapon(s) he has and how much ammo each one contains. Some games based on checkpoints don't even store this information—they start the player off in a known state at each checkpoint. Of course, the downside of a game based on checkpoints is the possibility of user frustration, especially if checkpoints are few and far between.

16.4.5.2 Save Anywhere

Some games support a feature known as *save anywhere*. As the name implies, such games permit the state of the game to be saved at literally any point during play. To implement this feature, the size of the saved game data file must increase significantly. The current locations and internal states of every game object whose state is relevant to gameplay must be saved and then restored when the game is loaded again later.

In a save-anywhere design, a saved game data file contains basically the same information as a world chunk, minus the world's static components. It is possible to utilize the same data format for both systems, although there may be factors that make this infeasible. For example, the world chunk data format might be designed for flexibility, but the saved game format might be compressed to minimize the size of each saved game.

As we've mentioned, one way to reduce the amount of data that needs to be stored in a saved game file is to omit certain irrelevant game objects and to omit some irrelevant details of others. For example, we needn't remember the exact time index within every animation that is currently playing or the exact momentums and velocities of every physically simulated rigid body. We can rely on the imperfect memories of human gamers and save only a rough approximation to the game's state.

16.5 对象引用和世界查询

每个游戏对象通常都需要某种唯一的 ID，以便与游戏中的其他对象区分开来，在运行时被找到，并作为对象间通信的目标等等。唯一的对象 ID 在工具方面同样有用，因为它们可以用来在世界编辑器中识别和查找游戏对象。

在运行时，我们总是需要各种方法来查找游戏对象。我们可能希望通过对象的唯一 ID、类型或一组任意条件来查找对象。我们经常需要执行基于邻近度的查询，例如查找玩家角色周围 10 米半径范围内的所有敌方外星人。

通过查询找到游戏对象后，我们需要某种方式来引用它。在 C 或 C++ 等语言中，对象引用可能实现为指针，或者我们可能使用更复杂的方式，例如句柄或智能指针。对象引用的生命周期可能差异很大，从单个函数调用的范围到几分钟的时间。在接下来的章节中，我们将首先研究实现对象引用的各种方法。然后，我们将探讨在实现游戏玩法时经常需要的查询类型以及如何实现这些查询。

16.5.1 指针

在 C 或 C++ 中，实现对象引用最直接的方式是通过指针（在 C++ 中称为引用）。指针功能强大，而且极其简单直观。然而，指针也存在一些问题：

- 孤立对象。理想情况下，每个对象都应该有一个所有者——另一个负责管理其生命周期的对象——创建它，并在不再需要时删除它。但指针并不能帮助程序员执行这条规则。结果可能是出现一个孤立对象——一个仍然占用内存，但不再被系统中任何其他对象需要或引用的对象。

- 过时的指针。如果删除了一个对象，理想情况下我们应该将指向该对象的所有指针清零。但是，如果我们忘记这样做，我们最终会得到一个过时的指针——一个指向曾经被有效对象占用但现在是空闲内存的内存块的指针。如果有人试图通过过时的指针读取或写入数据，结果可能会导致崩溃或程序行为不正确。过时的指针很难追踪，因为它们在对象删除后可能还会继续工作一段时间。只有在很久以后，当在过时的内存块上分配一个新对象时，数据才会真正发生变化并导致崩溃。

16.5 Object References and World Queries

Every game object generally requires some kind of unique id so that it can be distinguished from the other objects in the game, found at runtime, serve as a target of inter-object communication and so on. Unique object ids are equally helpful on the tool side, as they can be used to identify and find game objects within the world editor.

At runtime, we invariably need various ways to find game objects. We might want to find an object by its unique id, by its type, or by a set of arbitrary criteria. We often need to perform proximity-based queries, for example finding all enemy aliens within a 10 m radius of the player character.

Once a game object has been found via a query, we need some way to refer to it. In a language like C or C++, object references might be implemented as pointers, or we might use something more sophisticated, like handles or smart pointers. The lifetime of an object reference can vary widely, from the scope of a single function call to a period of many minutes. In the following sections, we'll first investigate various ways to implement object references. Then we'll explore the kinds of queries we often require when implementing gameplay and how those queries might be implemented.

16.5.1 Pointers

In C or C++, the most straightforward way to implement an object reference is via a pointer (or a reference in C++). Pointers are powerful and are just about as simple and intuitive as you can get. However, pointers suffer from a number of problems:

- *Orphaned objects*. Ideally, every object should have an *owner*—another object that is responsible for managing its lifetime—creating it and then deleting it when it is no longer needed. But pointers don't give the programmer any help in enforcing this rule. The result can be an *orphaned* object—an object that still occupies memory but is no longer needed or referenced by any other object in the system.
- *Stale pointers*. If an object is deleted, ideally we should null-out any and all pointers to that object. If we forget to do so, however, we end up with a stale pointer—a pointer to a block of memory that used to be occupied by a valid object but is now free memory. If anyone tries to read or write data through a stale pointer, the result can be a crash or incorrect program behavior. Stale pointers can be difficult to track down because they may continue to work for some time after the object has deleted. Only much later, when a new object is allocated on top of the stale memory block, does the data actually change and cause a crash.

• 无效指针。程序员可以随意在指针中存储任何地址，包括完全无效的地址。一个常见问题是取消引用空指针。可以使用断言宏来防止这些问题，在取消引用指针之前检查指针是否不为空。更糟糕的是，如果一段数据被误解为指针，取消引用它可能会导致程序读取或写入一个本质上随机的内存地址。这通常会导致崩溃或其他重大问题，并且很难调试。

许多游戏引擎大量使用指针，因为它们是迄今为止实现对象引用最快、最高效、最易用的方式。然而，经验丰富的程序员总是对指针保持警惕，一些游戏团队转而使用更复杂的对象引用，这要么是出于更安全的编程实践，要么是出于必要。例如，如果游戏引擎在运行时重定位已分配的数据块以消除内存碎片（参见第 6.2.2.2 节），则不能使用简单的指针。我们要么需要使用一种对内存重定位具有鲁棒性的对象引用类型，要么需要在移动每个重定位内存块时手动修复指向该内存块的指针。

16.5.2 智能指针

智能指针是一个小型对象，在大多数情况下，它的行为类似于指针，但避免了原生 C/C++ 指针固有的大多数问题。简单来说，智能指针包含一个原生指针作为数据成员，并提供一组重载运算符，使其在大多数情况下表现得像指针一样。指针可以被解引用，因此 * 和 -> 运算符被重载，分别返回指向被引用对象的引用和指针，正如您所期望的那样。指针可以进行□□算术运算，因此 + 、、 ++ 和

-operators 也适当地超载。

由于智能指针本身就是一个对象，它可以包含额外的元数据，并/或执行常规指针无法实现的额外步骤。例如，智能指针可能包含一些信息，使其能够识别其指向的对象是否已被删除，并在删除后返回空地址。

智能指针还可以通过相互协作来确定对特定对象的引用数量，从而帮助管理对象的生命周期。这被称为引用计数。当引用特定对象的智能指针数量降至零时，我们就知道该对象不再需要，因此可以自动删除它。这可以让程序员不必再担心对象所有权和孤立对象。

- *Invalid pointers*. A programmer is free to store any address in a pointer, including a totally invalid address. A common problem is dereferencing a null pointer. These problems can be guarded against by using assertion macros to check that pointers are never null prior to dereferencing them. Even worse, if a piece of data is misinterpreted as a pointer, dereferencing it can cause the program to read or write an essentially random memory address. This usually results in a crash or other major problem that can be very tough to debug.

Many game engines make heavy use of pointers, because they are by far the fastest, most efficient and easiest-to-work-with way to implement object references. However, experienced programmers are always wary of pointers, and some game teams turn to more sophisticated kinds of object references, either out of a desire to use safer programming practices or out of necessity. For example, if a game engine relocates allocated data blocks at runtime to eliminate memory fragmentation (see Section 6.2.2.2), simple pointers cannot be used. We either need to use a type of object reference that is robust to memory relocation, or we need to manually fix up any pointers into every relocated memory block at the time it is moved.

16.5.2 Smart Pointers

A *smart pointer* is a small object that acts like a pointer for most intents and purposes but avoids most of the problems inherent with native C/C++ pointers. At its simplest, a smart pointer contains a native pointer as a data member and provides a set of overloaded operators that make it act like a pointer in most ways. Pointers can be dereferenced, so the * and -> operators are overloaded to return a reference and a pointer to the referenced object, respectively, as you'd expect. Pointers can undergo arithmetic operations, so the +, -, ++ and -- operators are also overloaded appropriately.

Because a smart pointer is an object, it can contain additional metadata and/or take additional steps not possible with a regular pointer. For example, a smart pointer might contain information that allows it to recognize when the object to which it points has been deleted and start returning a null address if so.

Smart pointers can also help with object lifetime management by cooperating with one another to determine the number of references to a particular object. This is called *reference counting*. When the number of smart pointers that reference a particular object drops to zero, we know that the object is no longer needed, so it can be automatically deleted. This can free the programmer from having to worry about object ownership and orphaned objects.

引用计数通常也是 Java 和 Python 等现代编程语言中"垃圾收集"系统的核心。

智能指针也存在一些问题。首先，它们实现起来相对容易，但要正确使用却很难。需要处理的情况非常多，而原始 C++ 标准库提供的 std::auto_ptr 类被广泛认为在很多情况下都不够用。值得庆幸的是，C++11 引入了三个智能指针类： std::shared_ptr 、 std::weak_ptr 和 std::unique_ptr ，解决了大部分这些问题。

C++11 智能指针类模仿了 Boost C++ 模板库提供的丰富的智能指针功能。它定义了六种不同类型的智能指针：

- boost::scoped_ptr . 指向具有一个所有者的单个对象的指针。
- boost::scoped_array . 指向具有单一所有者的对象数组的指针。
- boost::shared_ptr . 指向一个对象的指针，该对象的生命周期由多个所有者共享。
- boost::shared_array . 指向对象数组的指针，该数组的生命周期由多个所有者共享。
- boost::weak_ptr . 不拥有或自动销毁其引用对象的指针（其生命周期假定由 shared_ptr 管理）。
- boost::intrusive_ptr 。一种通过假设指向的对象将自行维护引用计数来实现引用计数的指针。侵入式指针非常有用，因为它们的大小与原生 C++ 指针相同（因为不需要引用计数机制），并且可以直接从原生指针构造。

正确实现智能指针类可能是一项艰巨的任务。您可以浏览一下 Boost 智能指针文档 (http://www.boost.org/doc/libs/1_36_0/libs/smart_ptr/smart_ptr.htm) 来理解我的意思。各种各样的问题都会出现，包括：

- 智能指针的类型安全，
- 智能指针能够用于不完整类型，
- 发生异常时纠正智能指针行为，以及
- 运行成本可能很高。

Reference counting usually also lies at the core of the "garbage collection" systems found in modern programming languages like Java and Python.

Smart pointers have their share of problems. For one thing, they are relatively easy to implement, but they are tough to get right. There are a great many cases to handle, and the `std::auto_ptr` class provided by the original C++ standard library is widely recognized to be inadequate in many situations. Thankfully most of these issues were resolved in C++11 with the introduction of three smart pointer classes: `std::shared_ptr`, `std::weak_ptr` and `std::unique_ptr`.

The C++11 smart pointer classes were modeled after the rich smart pointer facilities provided by the Boost C++ template library. It defines six different varieties of smart pointers:

- `boost::scoped_ptr`. A pointer to a single object with one owner.
- `boost::scoped_array`. A pointer to an array of objects with a single owner.
- `boost::shared_ptr`. A pointer to an object whose lifetime is shared by multiple owners.
- `boost::shared_array`. A pointer to an array of objects whose lifetimes are shared by multiple owners.
- `boost::weak_ptr`. A pointer that does not own or automatically destroy the object it references (whose lifetime is assumed to be managed by a `shared_ptr`).
- `boost::intrusive_ptr`. A pointer that implements reference counting by assuming that the pointed-to object will maintain the reference count itself. Intrusive pointers are useful because they are the same size as a native C++ pointer (because no reference-counting apparatus is required) and because they can be constructed directly from native pointers.

Properly implementing a smart pointer class can be a daunting task. Have a glance at the Boost smart pointer documentation (http://www.boost.org/doc/libs/1_36_0/libs/smart_ptr/smart_ptr.htm) to see what I mean. All sorts of issues come up, including:

- type safety of smart pointers,
- the ability for a smart pointer to be used with an incomplete type,
- correct smart pointer behavior when an exception occurs, and
- runtime costs, which can be high.

图 16.13。句柄表包含原始对象指针。句柄只是该表的索引。

我曾经参与过一个尝试实现自己的智能指针的项目，直到项目结束，我们一直在修复各种棘手的 bug。我个人建议远离智能指针；如果必须使用，请使用成熟的实现，例如 C++11 标准库或 Boost，而不是自己动手实现。

16.5.3 句柄

句柄在很多方面类似于智能指针，但实现起来更简单，而且不容易出错。句柄本质上是全局句柄表的整数索引。句柄表又包含指向句柄所引用对象的指针。要创建句柄，我们只需在句柄表中搜索相关对象的地址，并将其索引存储在句柄中。要取消引用句柄，调用代码只需索引句柄表中的相应槽位，然后取消引用在那里找到的指针。如图 16.13 所示。

由于句柄表提供的简单间接层，句柄比原始指针更安全、更灵活。如果删除某个对象，只需将其在句柄表中的条目清空即可。这会导致该对象的所有现有句柄立即自动转换为空引用。句柄还支持内存重定位。当对象在内存中重定位时，可以在句柄表中找到其地址并进行相应更新。同样，该对象的所有现有句柄也会因此自动更新。

句柄可以实现为原始整数。然而，句柄表索引通常被包装在一个简单的类中，以便提供创建和取消引用句柄的便捷接口。

Figure 16.13. A handle table contains raw object pointers. A handle is simply an index into this table.

I worked on a project that attempted to implement its own smart pointers, and we were fixing all sorts of nasty bugs with them up until the very end of the project. My personal recommendation is to stay away from smart pointers; if you must use them, use a mature implementation such as the C++11 standard library or Boost, rather than rolling your own.

16.5.3 Handles

A *handle* acts like a smart pointer in many ways, but it is simpler to implement and tends to be less prone to problems. A handle is basically an integer index into a global *handle table*. The handle table, in turn, contains pointers to the objects to which the handles refer. To create a handle, we simply search the handle table for the address of the object in question and store its index in the handle. To dereference a handle, the calling code simply indexes the appropriate slot in the handle table and dereferences the pointer it finds there. This is illustrated in Figure 16.13.

Because of the simple level of indirection afforded by the handle table, handles are much safer and more flexible than raw pointers. If an object is deleted, it can simply null out its entry in the handle table. This causes all existing handles to the object to be immediately and automatically converted to null references. Handles also support memory relocation. When an object is relocated in memory, its address can be found in the handle table and updated appropriately. Again, all existing handles to the object are automatically updated as a result.

A handle can be implemented as a raw integer. However, the handle table index is usually wrapped in a simple class so that a convenient interface for creating and dereferencing the handle can be provided.

句柄很容易引用过时的对象。例如，假设我们创建了对象 A 的句柄，它占用句柄表中的第 17 个槽位。后来，对象 A 被删除，第 17 个槽位被清空。之后，又创建了一个新的对象 B，它恰好占用了句柄表中的第 17 个槽位。如果在创建对象 B 时仍有对象 A 的句柄，那么这些句柄就会突然开始引用对象 B（而不是空值）。这当然不是我们想要的行为。

解决句柄失效问题的一个简单方法是在每个句柄中包含一个唯一的对象 ID。这样，当创建对象 A 的句柄时，它不仅包含槽位索引 17，还包含对象 ID“A”。当对象 B 在句柄表中取代 A 的位置时，任何剩余的 A 句柄将在句柄索引上保持一致，但在对象 ID 上不一致。这使得失效的对象 A 句柄在取消引用时可以继续返回 null，而不是意外返回指向对象 B 的指针。

以下代码片段展示了如何实现一个简单的句柄类。请注意，我们还在 Game Object 类本身中包含了句柄索引——这使我们能够快速地为 GameObject 创建新的句柄，而无需在句柄表中搜索其地址来确定其句柄索引。

```
// Within the GameObject class, we store a unique id,
// and also the object's handle index, for efficient
// creation of new handles.

class GameObject
{
private:
    // ...

    GameObjectId m_uniqueId;       // object's unique id
    U32          m_handleIndex;    // speedier handle
                                   // creation

    friend class GameObjectHandle;  // access to id and
                                    // index

    // ...

public:
    GameObject() // constructor
    {
        // The unique id might come from the world editor,
        // or it might be assigned dynamically at runtime.
        m_uniqueId = AssignUniqueObjectId();
```


Handles are prone to the possibility of referencing a stale object. For example, let's say we create a handle to object A, which occupies slot 17 in the handle table. Later, object A is deleted, and slot 17 is nulled out. Later still, a new object B is created, and it just happens to occupy slot 17 in the handle table. If there are still any handles to object A lying around when object B is created, they will suddenly start referring to object B (instead of null). This is certainly not desirable behavior.

One simple solution to the stale handle problem is to include a unique object id in each handle. That way, when a handle to object A is created, it contains not only slot index 17, but the object id "A." When object B takes A's place in the handle table, any leftover handles to A will agree on the handle index but disagree on the object id. This allows stale object A handles to continue to return null when dereferenced rather than returning a pointer to object B unexpectedly.

The following code snippet shows how a simple handle class might be implemented. Notice that we've also included the handle index in the `GameObject` class itself—this allows us to create new handles to a `GameObject` very quickly without having to search the handle table for its address to determine its handle index.


```
// Within the GameObject class, we store a unique id,
// and also the object's handle index, for efficient
// creation of new handles.

class GameObject
{
private:
    // ...

    GameObjectId m_uniqueId;       // object's unique id
    U32          m_handleIndex;    // speedier handle
                                   // creation

    friend class GameObjectHandle;  // access to id and
                                    // index

    // ...

public:
    GameObject() // constructor
    {
        // The unique id might come from the world editor,
        // or it might be assigned dynamically at runtime.
        m_uniqueId = AssignUniqueObjectId();
```

```
        // The handle index is assigned by finding the
        // first free slot in the handle table.
        m_handleIndex = FindFreeSlotInHandleTable();

        // ...
    }
    // ...

};

// This constant defines the size of the handle table,
// and hence the maximum number of game objects that can
// exist at any one time.
static const U32 MAX_GAME_OBJECTS = 2048;

// This is the global handle table -- a simple array of
// pointers to GameObjects.
static GameObject* g_apGameObject[MAX_GAME_OBJECTS];

// This is our simple game object handle class.
class GameObjectHandle
{
private:
    U32 m_handleIndex;        // index into the handle
                              // table
    GameObjectId m_uniqueId;  // unique id avoids stale
                              // handles

public:
    explicit GameObjectHandle(GameObject& object) :
        m_handleIndex(object.m_handleIndex),
        m_uniqueId(object.m_uniqueId)
    {
    }

    // This function dereferences the handle.
    GameObject* ToObject() const
    {
        GameObject* pObject
         = g_apGameObject[m_handleIndex];

        if (pObject != nullptr
        &&  pObject->m_uniqueId == m_uniqueId)
        {
            return pObject;
        }
```



```
        // The handle index is assigned by finding the
        // first free slot in the handle table.
        m_handleIndex = FindFreeSlotInHandleTable();

        // ...
    }
    // ...

};

// This constant defines the size of the handle table,
// and hence the maximum number of game objects that can
// exist at any one time.
static const U32 MAX_GAME_OBJECTS = 2048;

// This is the global handle table -- a simple array of
// pointers to GameObjects.
static GameObject* g_apGameObject[MAX_GAME_OBJECTS];

// This is our simple game object handle class.
class GameObjectHandle
{
private:
    U32 m_handleIndex;        // index into the handle
                              // table
    GameObjectId m_uniqueId;  // unique id avoids stale
                              // handles

public:
    explicit GameObjectHandle(GameObject& object) :
        m_handleIndex(object.m_handleIndex),
        m_uniqueId(object.m_uniqueId)
    {
    }

    // This function dereferences the handle.
    GameObject* ToObject() const
    {
        GameObject* pObject
         = g_apGameObject[m_handleIndex];

        if (pObject != nullptr
        &&  pObject->m_uniqueId == m_uniqueId)
        {
            return pObject;
        }
```

```
        return nullptr;
    }
};
```

此示例功能齐全，但尚不完善。我们可能希望实现复制语义，提供额外的构造函数变体等等。全局句柄表中的条目可能包含其他信息，而不仅仅是指向每个游戏对象的原始指针。当然，像这样的固定大小句柄表实现并非唯一可行的设计；不同引擎的句柄系统略有不同。

值得注意的是，全局句柄表的一个好处是，它为我们提供了系统中所有活动游戏对象的现成列表。例如，全局句柄表可以用来快速高效地遍历世界中的所有游戏对象。在某些情况下，它还可以使其他类型的查询更容易实现。

16.5.4 游戏对象查询

每个游戏引擎都至少提供了几种在运行时查找游戏对象的方法。我们将这些搜索称为游戏对象查询。最简单的查询类型是通过唯一 ID 查找特定的游戏对象。然而，真正的游戏引擎会进行许多其他类型的游戏对象查询。以下是游戏开发者可能想要进行的查询类型的几个示例：

- 找到所有与玩家处于视线范围内的敌方角色。
- 遍历特定类型的所有游戏对象。
- 找到所有生命值超过 80% 的可破坏游戏物体。
- 将伤害传递到爆炸半径内的所有游戏物体。
- 按照从近到远的顺序，迭代子弹或其他射弹路径上的所有物体。

这个列表可以有很多页，当然它的内容高度依赖于正在制作的特定游戏的设计。

为了最大限度地提高游戏对象查询的灵活性，我们可以设想一个通用的游戏对象数据库，它能够使用任意搜索条件生成任意查询。理想情况下，我们的游戏对象数据库能够极其高效、快速地执行所有这些查询，最大限度地利用现有的硬件和软件资源。


```
        return nullptr;
    }
};
```

This example is functional but incomplete. We might want to implement copy semantics, provide additional constructor variants and so on. The entries in the global handle table might contain additional information, not just a raw pointer to each game object. And of course, a fixed size handle table implementation like this one isn't the only possible design; handle systems vary somewhat from engine to engine.

We should note that one fortunate side benefit of a global handle table is that it gives us a ready-made list of all active game objects in the system. The global handle table can be used to quickly and efficiently iterate over all game objects in the world, for example. It can also make implementing other kinds of queries easier in some cases.

16.5.4 Game Object Queries

Every game engine provides at least a few ways to find game objects at runtime. We'll call these searches *game object queries*. The simplest type of query is to find a particular game object by its unique id. However, a real game engine makes many other types of game object queries. Here are just a few examples of the kinds of queries a game developer might want to make:

- Find all enemy characters with line of sight to the player.
- Iterate over all game objects of a certain type.
- Find all destructible game objects with more than 80% health.
- Transmit damage to all game objects within the blast radius of an explosion.
- Iterate over all objects in the path of a bullet or other projectile, in nearest-to-farthest order.

This list could go on for many pages, and of course its contents are highly dependent upon the design of the particular game being made.

For maximum flexibility in performing game object queries, we could imagine a general-purpose game object database, complete with the ability to formulate arbitrary queries using arbitrary search criteria. Ideally, our game object database would perform all of these queries extremely efficiently and rapidly, making maximum use of whatever hardware and software resources are available.

实际上，这种灵活性和极快速度的理想结合通常难以实现。游戏团队通常会在游戏开发过程中确定最有可能需要的查询类型，并实现专门的数据结构来加速这些特定类型的查询。当需要新的查询时，工程师要么利用现有的数据结构来实现，要么在速度不够快的情况下自行设计新的数据结构。以下是一些可以加速特定类型游戏对象查询的专用数据结构示例：

- 通过唯一 ID 查找游戏对象。指向游戏对象的指针或句柄可以存储在以唯一 ID 为键的哈希表或二叉搜索树中。
- 遍历所有符合特定条件的对象。游戏对象可以根据各种条件预先排序到链表中（假设这些条件是预先知道的）。例如，我们可以构建一个包含特定类型所有游戏对象的列表，维护一个包含玩家特定半径范围内所有对象的列表，等等。
- 查找所有位于抛射物路径上或与某个目标点视线范围内的物体。碰撞系统通常用于执行此类游戏对象查询。大多数碰撞系统提供快速射线投射，有些系统还提供将其他形状（例如球体或任意凸体）投射到世界中以确定其撞击目标的能力。

  （参见第 13.3.7 节。）
- 查找给定区域或半径内的所有对象。我们可以考虑将游戏对象存储在某种空间哈希数据结构中。这可以简单到在整个游戏世界中放置一个水平网格，也可以更复杂，例如四叉树、八叉树、KD树或其他编码空间邻近度的数据结构。

16.6 实时更新游戏对象

每个游戏引擎，从最简单到最复杂，都需要某种方法来随时更新每个游戏对象的内部状态。游戏对象的状态可以定义为其所有属性（有时称为属性，在 C++ 语言中称为数据成员）的值。例如，Pong 游戏中球的状态由其在屏幕上的 (x, y) 位置和速度（行进的速度和方向）描述。由于游戏是动态的、基于时间的模拟，因此游戏对象的状态描述的是其在特定时刻的配置。换句话说，游戏对象的时间概念是离散的，而不是连续的。（然而，正如我们将看到的，将对象视为

In reality, such an ideal combination of flexibility and blinding speed is generally not possible. Instead, game teams usually determine which types of queries are most likely to be needed during development of the game, and specialized data structures are implemented to accelerate those particular types of queries. As new queries become necessary, the engineers either leverage preexisting data structures to implement them, or they invent new ones if sufficient speed cannot be obtained. Here are a few examples of specialized data structures that can accelerate specific types of game object queries:

- *Finding game objects by unique id*. Pointers or handles to the game objects could be stored in a hash table or binary search tree keyed by unique id.
- *Iterating over all objects that meet a particular criterion*. The game objects could be presorted into linked lists based on various criteria (presuming the criteria are known a priori). For example, we might construct a list of all game objects of a particular type, maintain a list of all objects within a particular radius of the player, etc.
- *Finding all objects in the path of a projectile or with line of sight to some target point*. The collision system is usually leveraged to perform these kinds of game object queries. Most collision systems provide fast ray casts, and some also provide the ability to cast other shapes such as spheres or arbitrary convex volumes into the world to determine what they hit. (See Section 13.3.7.)
- *Finding all objects within a given region or radius*. We might consider storing our game objects in some kind of spatial hash data structure. This could be as simple as a horizontal grid placed over the entire game world or something more sophisticated, such as a quadtree, octree, *k*d-tree or other data structure that encodes spatial proximity.

16.6 Updating Game Objects in Real Time

Every game engine, from the simplest to the most complex, requires some means of updating the internal state of every game object over time. The *state* of a game object can be defined as the values of all its *attributes* (sometimes called its *properties*, and called *data members* in the C++ language). For example, the state of the ball in *Pong* is described by its (x, y) position on the screen and its velocity (speed and direction of travel). Because games are dynamic, time-based simulations, a game object's state describes its configuration at *one specific instant in time*. In other words, a game object's notion of time is *discrete* rather than *continuous*. (However, as we'll see, it's helpful to think of the ob-

（我们将对象的状态视为连续变化，然后由引擎离散采样，因为它可以帮助您避免一些常见的陷阱。）在接下来的讨论中，我们将使用符号 S i（t）来表示对象 i 在任意时间 t 的状态。这里使用向量符号在数学上并不严格，但它提醒我们，游戏对象的状态就像一个异构的 n 维向量，包含各种数据类型的信息。我们应该注意，术语“状态”的这种用法与有限状态机中的状态不同。游戏对象很可能用一个或多个有限状态机来实现，但在这种情况下，每个 FSM 的当前状态的规范仅仅是游戏对象的整体状态向量 S（t）的一部分。

大多数底层引擎子系统（渲染、动画、碰撞、物理、音频等等）都需要定期更新，游戏对象系统也不例外。正如我们在第 8 章中看到的，更新通常通过一个称为游戏循环的主循环完成。几乎所有游戏引擎都将游戏对象状态更新作为其主游戏循环的一部分——换句话说，它们将游戏对象模型视为另一个需要定期维护的引擎子系统。

因此，游戏对象更新可以被认为是根据前一时刻 S i（t − Δ t）的状态，确定当前时刻 S i（t）中每个对象状态的过程。所有对象状态更新完成后，当前时间 t 将成为新的前一时刻（t − Δ t），并且此过程会在游戏运行期间重复。通常，引擎会维护一个或多个时钟——一个精确跟踪实时时间，其他时钟可能与实时时间一致，也可能不一致。这些时钟为引擎提供绝对时间 t 和/或游戏循环每次迭代之间的时间变化 Δ t。驱动游戏对象状态更新的时钟通常允许与实时时间不同。这允许游戏对象的行为暂停、减速、加速甚至反向运行——无论需要什么，以满足游戏设计的需求。这些功能对于游戏的调试和开发也至关重要。

正如我们在第一章中了解到的，游戏对象更新系统是计算机科学中所谓的动态、实时、基于代理的计算机模拟的一个例子。游戏对象更新系统也展现了离散事件模拟的某些方面（有关事件的更多详细信息，请参阅第 16.8 节）。这些是计算机科学中研究较为深入的领域，并且在互动娱乐领域之外有着广泛的应用。游戏是基于代理的模拟中较为复杂的一种——正如我们将看到的，在动态、交互式虚拟环境中，随着时间的推移更新游戏对象的状态可能出奇地难以做到正确。游戏程序员可以学到很多关于游戏

jects' states as changing continuously and then being sampled discretely by the engine, because it helps you to avoid some common pitfalls.)

In the following discussions, we'll use the symbol $\mathbf{S}_i(t)$ to denote the state of object i at an arbitrary time t. The use of vector notation here is not strictly mathematically correct, but it reminds us that a game object's state acts like a heterogeneous n-dimensional vector, containing all sorts of information of various data types. We should note that this usage of the term "state" is not the same as the states in a *finite state machine*. A game object may very well be implemented in terms of one—or many—finite state machines, but in that case, a specification of the current state of each FSM would merely be a part of the game object's overall state vector $\mathbf{S}(t)$.

Most low-level engine subsystems (rendering, animation, collision, physics, audio and so on) require periodic updating, and the game object system is no exception. As we saw in Chapter 8, updating is usually done via a single master loop called the *game loop*. Virtually all game engines update game object states as part of their main game loop—in other words, they treat the game object model as just another engine subsystem that requires periodic servicing.

Game object updating can therefore be thought of as the process of determining the state of each object at the current time $\mathbf{S}_i(t)$ given its state at a previous time $\mathbf{S}_i(t - \Delta t)$. Once all object states have been updated, the current time t becomes the new previous time $(t - \Delta t)$, and this process repeats for as long as the game is running. Usually, one or more *clocks* are maintained by the engine—one that tracks real time exactly and possibly others that may or may not correspond to real time. These clocks provide the engine with the absolute time t and/or with the change in time Δt from iteration to iteration of the game loop. The clock that drives the updating of game object states is usually permitted to diverge from real time. This allows the behaviors of the game objects to be paused, slowed down, sped up or even run in reverse—whatever is required in order to suit the needs of the game design. These features are also invaluable for debugging and development of the game.

As we learned in Chapter 1, a game object updating system is an example of what is known as a *dynamic, real-time, agent-based computer simulation* in computer science. Game object updating systems also exhibit some aspects of *discrete event simulations* (see Section 16.8 for more details on events). These are well-researched areas of computer science, and they have many applications outside the field of interactive entertainment. Games are one of the more complex kinds of agent-based simulation—as we'll see, updating game object states over time in a dynamic, interactive virtual environment can be surprisingly difficult to get right. Game programmers can learn a lot about game

通过研究基于代理和离散事件模拟的更广泛领域，可以实现对象更新。这些领域的研究人员或许也能从游戏引擎设计中学到一些东西！

与所有高级游戏引擎系统一样，每个引擎都采用略微（有时甚至是截然不同）不同的方法。然而，与之前一样，大多数游戏团队都会遇到一系列共同的问题，并且某些设计模式几乎在每个引擎中都会反复出现。在本节中，我们将探讨这些常见问题及其一些常见的解决方案。请记住，某些游戏引擎可能采用与本文所述截然不同的解决方案，并且某些游戏设计面临着我们无法在此涵盖的独特问题。

16.6.1 一种简单的方法（但不起作用）

更新游戏对象集合状态的最简单方法是遍历该集合，并依次对每个对象调用一个类似于 Update() 的虚函数。这通常在主游戏循环的每次迭代中执行一次（即每帧一次）。游戏对象类可以提供 Update() 函数的自定义实现，以便执行将该类型对象的状态推进到下一个离散时间索引所需的任何任务。可以将与上一帧的时间增量传递给更新函数，以便对象能够正确考虑时间的流逝。那么，最简□□单的 Update() 函数签名可能如下所示：

```
virtual void Update(float dt);
```

为了便于后续讨论，我们假设我们的引擎采用单体对象层级结构，其中每个游戏对象都由单个类的单个实例表示。然而，我们可以轻松地将此处的思路扩展到几乎任何以对象为中心的设计。例如，要更新基于组件的对象模型，我们可以对构成每个游戏对象的每个组件调用 Update()，或者对“中心”对象调用 Update()，让它根据自身情况更新其关联组件。我们还可以将以上思路扩展到以属性为中心的设计，即每帧对每个属性实例调用某种 Update() 函数。

俗话说细节决定成败，所以让我们来研究一下两个重要的细节。首先，我们应该如何维护所有游戏对象的集合？其次，Update() 函数应该负责哪些事情？

object updating by studying the wider field of agent-based and discrete event simulations. And researchers in those fields can probably learn a thing or two from game engine design as well!

As with all high-level game engine systems, every engine takes a slightly (or sometimes radically) different approach. However, as before, most game teams encounter a common set of problems, and certain design patterns tend to crop up again and again in virtually every engine. In this section, we'll investigate these common problems and some common solutions to them. Please bear in mind that game engines may exist that employ very different solutions to the ones described here, and some game designs face unique problems that we can't possibly cover here.

16.6.1 A Simple Approach (That Doesn't Work)

The simplest way to update the states of a collection of game objects is to iterate over the collection and call a virtual function, named something like `Update()`, on each object in turn. This is typically done once during each iteration of the main game loop (i.e., once per *frame*). Game object classes can provide custom implementations of the `Update()` function in order to perform whatever tasks are required to advance the state of that type of object to the next discrete time index. The time delta from the previous frame can be passed to the update function so that objects can take proper account of the passage of time. At its simplest, then, our `Update()` function's signature might look something like this:

```
virtual void Update(float dt);
```

For the purposes of the following discussions, we'll assume that our engine employs a monolithic object hierarchy, in which each game object is represented by a single instance of a single class. However, we can easily extend the ideas here to virtually any object-centric design. For example, to update a component-based object model, we could call `Update()` on every component that makes up each game object, or we could call `Update()` on the "hub" object and let it update its associated components as it sees fit. We can also extend these ideas to property-centric designs, by calling some sort of `Update()` function on each property instance every frame.

They say that the devil is in the details, so let's investigate two important details here. First, how should we maintain the collection of all game objects? And second, what kinds of things should the `Update()` function be responsible for doing?

16.6.1.1 维护活动游戏对象集合

活动游戏对象集合通常由一个单例管理器类维护，该类的名称可能类似于 GameWorld 或 GameObject Manager 。游戏对象集合通常需要动态变化，因为游戏对象会在游戏进行过程中生成和销毁。因此，使用指向游戏对象的指针、智能指针或句柄的链表是一种简单有效的方法。（某些游戏引擎不允许动态生成和销毁游戏对象；这类引擎可以使用静态大小的游戏对象指针、智能指针或句柄数组，而不是链表。）正如我们将在下面看到的，大多数引擎使用更复杂的数据结构来跟踪其游戏对象，而不仅仅是简单的扁平链表。但目前，为了简单起见，我们可以将数据结构可视化为链表。

16.6.1.2 Update()函数的职责

游戏对象的 Update() 函数主要负责根据先前状态 $S_i(t-\Delta t)$ 确定该游戏对象在当前离散时间索引 $S_i(t)$ 的状态。这可能涉及对对象应用刚体动力学模拟、采样预先编写的动画、对当前时间步长内发生的事件做出反应等等。

大多数游戏对象都会与一个或多个引擎子系统交互。它们可能需要动画、渲染、发射粒子效果、播放音频、与其他对象和静态几何体碰撞等等。每个系统都具有内部状态，并且必须随时间更新，通常每帧更新一次或几次。直接从游戏对象的 Update() 函数中更新所有这些子系统似乎是合理且直观的。例如，考虑以下假设的 Tank 对象更新函数：

```
virtual void Tank::Update(float dt)
{
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Now update low-level engine subsystems on behalf
    // of this tank. (NOT a good idea... see below!)
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->draw();
```


16.6.1.1 Maintaining a Collection of Active Game Objects

The collection of active game objects is often maintained by a singleton manager class, perhaps named something like `GameWorld` or `GameObjectManager`. The collection of game objects generally needs to be dynamic, because game objects are spawned and destroyed as the game is played. Hence a *linked list* of pointers, smart pointers or handles to game objects is one simple and effective approach. (Some game engines disallow dynamic spawning and destroying of game objects; such engines can use a statically sized *array* of game object pointers, smart pointers or handles rather than a linked list.) As we'll see below, most engines use more complex data structures to keep track of their game objects rather than just a simple, flat linked list. But for the time being, we can visualize the data structure as a linked list for simplicity.

16.6.1.2 Responsibilities of the `Update()` Function

A game object's `Update()` function is primarily responsible for determining the state of that game object at the current discrete time index $\mathbf{S}_i(t)$ given its previous state $\mathbf{S}_i(t-\Delta t)$. Doing this may involve applying a rigid body dynamics simulation to the object, sampling a preauthored animation, reacting to events that have occurred during the current time step and so on.

Most game objects interact with one or more engine subsystems. They may need to animate, be rendered, emit particle effects, play audio, collide with other objects and static geometry and so on. Each of these systems has an internal state that must also be updated over time, usually once or a few times per frame. It might seem reasonable and intuitive to simply update all of these subsystems directly from within the game object's `Update()` function. For example, consider the following hypothetical update function for a `Tank` object:

```
virtual void Tank::Update(float dt)
{
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Now update low-level engine subsystems on behalf
    // of this tank. (NOT a good idea... see below!)
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->draw();
```

```
}
```

鉴于我们的 Update() 函数结构如此，游戏循环几乎可以完全由游戏对象的更新来驱动，如下所示：

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        // This hypothetical Update() function updates
        // all engine subsystems!
        gameObject.Update(dt);
    }

    g_renderingEngine.SwapBuffers();
}
```

尽管上面展示的简单的对象更新方法看起来很有吸引力，但它通常在商业级游戏引擎中并不可行。在接下来的章节中，我们将探讨这种简单方法的一些问题，并研究每个问题的常用解决方法。

16.6.2 性能约束和批量更新

大多数底层引擎系统都具有极其严格的性能约束。它们需要处理大量数据，并且必须尽可能快地在每一帧中执行大量计算。因此，大多数引擎系统都受益于批量更新。例如，批量更新大量动画通常比与其他不相关的操作（例如碰撞检测、物理模拟和渲染）交错更新每个对象的动画效率高得多。

在大多数商业游戏引擎中，每个引擎子系统都由主游戏循环直接或间接更新，而不是在每个游戏对象的 Update() 函数中逐个更新。如果某个游戏对象需要某个引擎子系统的服务，它会请求该子系统为其分配一些特定于子系统的状态信息。例如，一个希望通过三角形网格进行渲染的游戏对象可能会请求渲染子系统为其分配一个网格实例。（如第 11.1.1.5 节所述，一个网格实例代表一个


```
}
```

Given that our `Update()` functions are structured like this, the game loop could be driven almost entirely by the updating of the game objects, like this:

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        // This hypothetical Update() function updates
        // all engine subsystems!
        gameObject.Update(dt);
    }

    g_renderingEngine.SwapBuffers();
}
```

However attractive the simple approach to object updating shown above may seem, it is usually not viable in a commercial-grade game engine. In the following sections, we'll explore some of the problems with this simplistic approach and investigate common ways in which each problem can be solved.

16.6.2 Performance Constraints and Batched Updates

Most low-level engine systems have extremely stringent performance constraints. They operate on a large quantity of data, and they must do a large number of calculations every frame as quickly as possible. As a result, most engine systems benefit from *batched updating*. For example, it is usually far more efficient to update a large number of animations in one batch than it is to update each object's animation interleaved with other unrelated operations, such as collision detection, physical simulation and rendering.

In most commercial game engines, each engine subsystem is updated directly or indirectly by the main game loop rather than being updated on a per-game object basis from within each object's `Update()` function. If a game object requires the services of a particular engine subsystem, it asks that subsystem to allocate some subsystem-specific state information on its behalf. For example, a game object that wishes to be rendered via a triangle mesh might request the rendering subsystem to allocate a *mesh instance* for its use. (As described in Section 11.1.1.5, a mesh instance represents a single instance of a

三角形网格——它会跟踪实例在世界空间中的位置、方向和比例（无论它是否可见）、每个实例的材质数据以及任何其他可能相关的每个实例的信息。渲染引擎在内部维护一个网格实例集合。它可以按照自己认为合适的方式管理网格实例，以最大限度地提高自身的运行时性能。游戏对象通过操作网格实例对象的属性来控制自身的渲染方式，但游戏对象并不直接控制网格实例的渲染。相反，在所有游戏对象都有机会更新自身之后，渲染引擎会通过一次高效的批量更新来绘制所有可见的网格实例。

通过批量更新，特定游戏对象的 Update() 函数（例如我们假设的坦克对象的函数）可能看起来更像这样：

```
virtual void Tank::Update(float dt)
{
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Control the properties of my various engine
    // subsystem components, but do NOT update
    // them here...

    if (justExploded)
    {
        m_pAnimationComponent->PlayAnimation("explode");
    }

    if (isVisible)
    {
        m_pCollisionComponent->Activate();
        m_pRenderingComponent->Show();
    }
    else
    {
        m_pCollisionComponent->Deactivate();
        m_pRenderingComponent->Hide();
    }

    // etc.
}
```

游戏循环最终看起来更像这样：

```
while (true)
{
```


triangle mesh—it keeps track of the position, orientation and scale of the instance in world space whether or not it is visible, per-instance material data and any other per-instance information that may be relevant.) The rendering engine maintains a collection of mesh instances internally. It can manage the mesh instances however it sees fit in order to maximize its own runtime performance. The game object controls *how* it is rendered by manipulating the properties of the mesh instance object, but the game object does not control the rendering of the mesh instance directly. Instead, after all game objects have had a chance to update themselves, the rendering engine draws all visible mesh instances in one efficient batch update.

With batched updating, a particular game object's `Update()` function, such as that of our hypothetical tank object, might look more like this:

```
virtual void Tank::Update(float dt)
{
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Control the properties of my various engine
    // subsystem components, but do NOT update
    // them here...

    if (justExploded)
    {
        m_pAnimationComponent->PlayAnimation("explode");
    }

    if (isVisible)
    {
        m_pCollisionComponent->Activate();
        m_pRenderingComponent->Show();
    }
    else
    {
        m_pCollisionComponent->Deactivate();
        m_pRenderingComponent->Hide();
    }

    // etc.
}
```

The game loop then ends up looking more like this:

```
while (true)
{
```

```
        PollJoypad();

        float dt = g_gameClock.CalculateDeltaTime();

        for (each gameObject)
        {
            gameObject.Update(dt);
        }

        g_animationEngine.Update(dt);
        g_physicsEngine.Simulate(dt);
        g_collisionEngine.DetectAndResolveCollisions(dt);
        g_audioEngine.Update(dt);
        g_renderingEngine.RenderFrameAndSwapBuffers();
    }
```

批量更新提供了许多性能优势，包括但不限于：

- 最大程度的缓存一致性。批量更新允许引擎子系统实现最大程度的缓存一致性，因为其每个对象的数据都在内部维护，并且可以排列在单个连续的 RAM 区域中。
- 最小化重复计算。全局计算只需进行一次，即可重复用于多个游戏对象，而不必为每个对象重复进行计算。
- 减少资源重新分配。引擎子系统通常需要在更新期间分配和管理内存和/或其他资源。如果某个子系统的更新与其他引擎子系统的更新交错进行，则必须为每个正在处理的游戏对象释放并重新分配这些资源。但如果更新是批量进行的，则可以每帧分配一次资源，并重复用于该批次中的所有对象。
- 高效的流水线处理。许多引擎子系统对游戏世界中的每个对象执行几乎相同的计算。当更新被批量处理时，可以采用分散/聚集方法将大量工作负载分配到多个 CPU 核心上。单独处理每个对象时，无法实现这种并行性。

性能优势并非选择批量更新方法的唯一原因。有些引擎子系统在逐个对象更新时根本无法正常工作。例如，如果我们尝试解决由多个动态刚体组成的系统内的碰撞问题，就无法找到令人满意的解决方案。


```
        PollJoypad();

        float dt = g_gameClock.CalculateDeltaTime();

        for (each gameObject)
        {
            gameObject.Update(dt);
        }

        g_animationEngine.Update(dt);
        g_physicsEngine.Simulate(dt);
        g_collisionEngine.DetectAndResolveCollisions(dt);
        g_audioEngine.Update(dt);
        g_renderingEngine.RenderFrameAndSwapBuffers();
    }
```

Batched updating provides many performance benefits, including but not limited to:

- *Maximal cache coherency*. Batched updating allows an engine subsystem to achieve maximum cache coherency because its per-object data is maintained internally and can be arranged in a single, contiguous region of RAM.
- *Minimal duplication of computations*. Global calculations can be done once and reused for many game objects rather than being redone for each object.
- *Reduced reallocation of resources*. Engine subsystems often need to allocate and manage memory and/or other resources during their updates. If the update of a particular subsystem is interleaved with those of other engine subsystems, these resources must be freed and reallocated for each game object that is processed. But if the updates are batched, the resources can be allocated once per frame and reused for all objects in the batch.
- *Efficient pipelining*. Many engine subsystems perform a virtually identical set of calculations on each and every object in the game world. When updates are batched, a scatter/gather approach can be employed to divide large workloads across multiple CPU cores. This kind of parallelism cannot be achieved when processing each object in isolation.

Performance benefits aren't the only reason to favor a batch updating approach. Some engine subsystems simply don't work at all when updated on a per-object basis. For example, if we are trying to resolve collisions within a system of multiple dynamic rigid bodies, a satisfactory solution cannot be

通过孤立地考虑每个对象，可以发现一般的相互渗透。这些对象之间的相互渗透必须作为一个整体来解决，可以通过迭代方法或求解线性系统来解决。

16.6.3 对象和子系统相互依赖性

即使我们不关心性能，当游戏对象相互依赖时，简单的逐个对象更新方法也会失效。例如，一个人类角色可能怀里抱着一只猫。为了计算猫骨骼的世界空间姿势，我们首先需要计算人类的世界空间姿势。这意味着对象的更新顺序对于游戏的正常运行至关重要。

当引擎子系统相互依赖时，还会出现另一个相关问题。例如，布娃娃物理模拟必须与动画引擎协同更新。通常，动画系统会生成一个中间的局部空间骨骼姿势。这些关节变换会被转换到世界空间，并应用于物理系统内连接刚体系统，这些刚体系统近似于物理系统中的骨架。物理系统会按时间向前模拟刚体，然后将关节的最终静止位置应用回骨架中对应的关节。最后，动画系统计算世界空间姿势和蒙皮矩阵调色板。因此，动画和物理系统的更新必须按照特定顺序进行，才能产生正确的结果。这类子系统间的依赖关系在游戏引擎设计中很常见。

16.6.3.1 分阶段更新

为了解决子系统间的依赖关系，我们可以在主游戏循环中以正确的顺序显式编写引擎子系统的更新代码。例如，为了处理动画系统和布娃娃物理系统之间的相互作用，我们可以编写如下代码：

```
while (true) // main game loop
{
    // ...

    g_animationEngine.CalculateIntermediatePoses(dt);
    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // runs ragdolls too
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();
```


found in general by considering each object in isolation. The interpenetrations between these objects must be resolved as a group, either via an iterative approach or by solving a linear system.

16.6.3 Object and Subsystem Interdependencies

Even if we didn't care about performance, a simplistic per-object updating approach breaks down when game objects *depend* on one another. For example, a human character might be holding a cat in her arms. In order to calculate the world-space pose of the cat's skeleton, we first need to calculate the world-space pose of the human. This implies that the *order* in which objects are updated is important to the proper functioning of the game.

Another related problem arises when engine *subsystems* depend on one another. For example, a rag doll physics simulation must be updated in concert with the animation engine. Typically, the animation system produces an intermediate, local-space skeletal pose. These joint transforms are converted to world space and applied to a system of connected rigid bodies that approximate the skeleton within the physics system. The rigid bodies are simulated forward in time by the physics system, and then the final resting places of the joints are applied back to their corresponding joints in the skeleton. Finally, the animation system calculates the world-space pose and skinning matrix palette. So once again, the updating of the animation and physics systems must occur in a particular order in order to produce correct results. These kinds of inter-subsystem dependencies are commonplace in game engine design.

16.6.3.1 Phased Updates

To account for inter-subsystem dependencies, we can explicitly code our engine subsystem updates in the proper order within the main game loop. For example, to handle the interplay between the animation system and rag doll physics, we might write something like this:

```
while (true) // main game loop
{
    // ...

    g_animationEngine.CalculateIntermediatePoses(dt);
    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // runs ragdolls too
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();
```

```
    // ...
}
```

我们必须谨慎地在游戏循环的正确时间更新游戏对象的状态。这通常不像每个游戏对象每帧调用一个 Update() 函数那么简单。游戏对象可能依赖于各种引擎子系统执行计算的中间结果。例如，一个游戏对象可能要求在动画系统运行其更新之前播放动画。然而，同一个对象可能还希望在动画系统生成的中间姿势被布娃娃物理系统使用之前和/或生成最终姿势和矩阵调色板之前，程序性地调整该姿势。这意味着对象必须更新两次：一次在动画计算其中间姿势之前，一次在之后。

许多游戏引擎允许其游戏对象在帧的多个时间点运行更新逻辑。例如，顽皮狗引擎（驱动《神秘海域》和《最后生还者》系列游戏的引擎）会更新游戏对象三次：一次在动画混合之前，一次在动画混合之后但在最终姿势生成之前，一次在最终姿势生成之后。这可以通过为每个游戏对象类提供三个充当“钩子”的虚函数来实现。在这样的系统中，游戏循环最终看起来像这样：

```
while (true) // main game loop
{
    // ...

    for (each gameObject)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses(dt);

    for (each gameObject)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // runs ragdolls too
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
```



```
    // ...
}
```

We must be careful to update the states of our game objects at the right time during the game loop. This is often not as simple as calling a single `Update()` function per game object per frame. Game objects may depend upon the intermediate results of calculations performed by various engine subsystems. For example, a game object might request that animations be played prior to the animation system running its update. However, that same object may also want to procedurally adjust the intermediate pose generated by the animation system prior to that pose being used by the rag doll physics system and/or the final pose and matrix palette being generated. This implies that the object must be updated twice, once before the animation calculates its intermediate poses and once afterward.

Many game engines allow their game objects to run update logic at multiple points during the frame. For example, the Naughty Dog engine (the engine that drives the *Uncharted* and *The Last of Us* game series) updates game objects three times—once before animation blending, once after animation blending but prior to final pose generation and once after final pose generation. This can be accomplished by providing each game object class with three virtual functions that act as “hooks.” In such a system, the game loop ends up looking something like this:

```
while (true) // main game loop
{
    // ...

    for (each gameObject)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses(dt);

    for (each gameObject)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // runs ragdolls too
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
```

```
        g_animationEngine.FinalizePoseAndMatrixPalette();

        for (each gameObject)
        {
            gameObject.FinalUpdate(dt);
        }

        // ...
    }
```

我们可以为游戏对象提供任意数量的更新阶段。但必须谨慎，因为遍历所有游戏对象并在每个对象上调用虚函数可能会非常耗时。此外，并非所有游戏对象都需要所有更新阶段——遍历不需要特定阶段的对象纯粹是浪费 CPU 带宽。

实际上，上面的例子并不完全符合实际。直接遍历所有游戏对象来调用它们的 PreAnimUpdate() 、 PostAnimUpdate() 和 FinalUpdate() 钩子函数效率极低，因为实际上可能只有一小部分对象需要在每个钩子中执行逻辑。此外，这种设计也不够灵活，因为只支持游戏对象——如果我们想在动画后期更新粒子系统，那就没办法了。最后，这样的设计会导致底层引擎系统和游戏对象系统之间不必要的耦合。

通用回调机制将是更好的设计选择。在这样的设计中，动画系统将提供一种便利，任何客户端代码（游戏对象或任何其他引擎系统）都可以为三个更新阶段（动画前、动画后和最终）分别注册一个回调函数。动画系统将遍历所有已注册的回调并进行调用，而无需了解游戏对象本身。这种设计最大限度地提高了性能，因为只有真正需要更新的客户端才会注册回调并在每一帧中被调用。它还最大限度地提高了灵活性，并消除了游戏对象系统与其他引擎子系统之间不必要的耦合，因为任何客户端都可以注册回调，而不仅仅是游戏对象。

16.6.3.2 分桶更新

在存在对象间依赖关系的情况下，上述分阶段更新技术必须进行一些调整。这是因为对象间依赖关系可能导致控制更新顺序的规则发生冲突。例如，假设对象 B 被对象 A 持有。进一步假设，只有在 A 完全更新后才能更新对象 B，包括


```
        g_animationEngine.FinalizePoseAndMatrixPalette();

        for (each gameObject)
        {
            gameObject.FinalUpdate(dt);
        }

        // ...
    }
```

We can provide our game objects with as many update phases as we see fit. But we must be careful, because iterating over all game objects and calling a virtual function on each one can be expensive. Also, not all game objects require all update phases—iterating over objects that don't require a particular phase is a pure waste of CPU bandwidth.

Actually, the above example isn't completely realistic. Iterating directly over all game objects to call their `PreAnimUpdate()`, `PostAnimUpdate()` and `FinalUpdate()` hook functions would be highly inefficient, because only a small percentage of the objects might actually need to perform any logic in each hook. It's also an inflexible design, because only game objects are supported—if we wanted to update a particle system during the post-animation phase, we'd be out of luck. Finally, such a design would lead to unnecessary *coupling* between the low-level engine systems and the game object system.

A generic callback mechanism would be a much better design choice. In such a design, the animation system would provide a facility by which any client code (game objects or any other engine system) could *register a callback function* for each of the three update phases (pre-animation, post-animation and final). The animation system would iterate through all registered callbacks and call them, without any "knowledge" of game objects *per se*. This design maximizes performance, because only those clients that actually *need* updates register callbacks and are called each frame. It also maximizes flexibility and eliminates unnecessary coupling between the game object system and other engine subsystems, because any client is allowed to register a callback, not just game objects.

16.6.3.2 Bucketed Updates

In the presence of *inter-object* dependencies, the phased updates technique described above must be adjusted a little. This is because inter-object dependencies can lead to conflicting rules governing the order of updating. For example, let's imagine that object B is being held by object A. Further, let's assume that we can only update object B after A has been *fully* updated, including the

图 16.14 对象间更新顺序依赖关系可以看作是一个依赖关系森林树木。

计算其最终的世界空间姿势和矩阵调色板。这与批量更新所有游戏对象的动画的需求相冲突，这样动画系统才能实现最大吞吐量。

对象间的依赖关系可以被视为依赖树的森林。没有父级（不依赖任何其他对象）的游戏对象代表森林的根。直接依赖于这些根对象的对象位于森林中某棵树的第一层子对象中。依赖于第一层子对象的对象将成为第二层子对象，依此类推。如图 16.14 所示。

解决更新顺序冲突问题的一个方法是将对象收集到独立的组中，由于没有更好的名称，我们在此将其称为“桶”。第一个桶包含森林中的所有根对象。第二个桶包含所有第一层级的子对象。第三个桶包含所有第二层级的子对象，依此类推。对于每个桶，我们会对游戏对象和引擎系统进行一次完整的更新，包括所有更新阶段。然后，我们会对每个桶重复整个过程，直到没有其他桶为止。

理论上，我们的依赖森林中的树的深度是无限的。然而，在实践中，它们通常相当浅显。例如，我们可能有手持武器的角色，这些角色可能骑着也可能不骑着移动平台或车辆。为了实现这一点，我们的依赖关系森林只需要三层，因此只需要三个存储桶：一个用于

Figure 16.14. Inter-object update order dependencies can be viewed as a forest of dependency trees.

calculation of its final world-space pose and matrix palette. This conflicts with the need to batch animation updates of all game objects together in order to allow the animation system to achieve maximum throughput.

Inter-object dependencies can be visualized as a forest of dependency trees. The game objects with no parents (no dependencies on any other object) represent the roots of the forest. An object that depends directly on one of these root objects resides in the first tier of children in one of the trees in the forest. An object that depends on a first-tier child becomes a second-tier child and so on. This is illustrated in Figure 16.14.

One solution to the problem of conflicting update order requirements is to collect objects into independent groups, which we'll call *buckets* here for lack of a better name. The first bucket consists of all root objects in the forest. The second bucket is comprised of all first-tier children. The third bucket contains all second-tier children and so on. For each bucket, we run a complete update of the game objects and the engine systems, complete with all update phases. Then we repeat the entire process for each bucket until there are no more buckets.

In theory, the depths of the trees in our dependency forest are unbounded. However, in practice, they are usually quite shallow. For example, we might have characters holding weapons, and those characters might or might not be riding on a moving platform or a vehicle. To implement this, we only need three tiers in our dependency forest, and hence only three buckets: one for

平台/载具，一个用于角色，一个用于角色手中的武器。许多游戏引擎明确限制了依赖森林的深度，以便使用固定数量的桶（假设它们完全使用桶的方法——当然，还有许多其他方法来构建游戏循环）。

以下是分阶段、分批更新循环可能的样子：

```
enum Bucket
{
    kBucketVehiclesPlatforms,
    kBucketCharacters,
    kBucketAttachedObjects,
    kBucketCount
};

void UpdateBucket(Bucket bucket)
{
    // ...

    for (each gameObject in bucket)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses
     (bucket, dt);

    for (each gameObject in bucket)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls(bucket);
    g_physicsEngine.Simulate(bucket, dt); // ragdolls etc.
    g_collisionEngine.DetectAndResolveCollisions
     (bucket, dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons(bucket);
    g_animationEngine.FinalizePoseAndMatrixPalette
     (bucket);

    for (each gameObject in bucket)
    {
        gameObject.FinalUpdate(dt);
    }
```


platforms/vehicles, one for characters and one for the weapons in the characters' hands. Many game engines explicitly limit the depth of their dependency forest so that they can use a fixed number of buckets (presuming they use a bucketed approach at all—there are of course many other ways to architect a game loop).

Here's what a bucketed, phased, batched update loop might look like:

```
enum Bucket
{
    kBucketVehiclesPlatforms,
    kBucketCharacters,
    kBucketAttachedObjects,
    kBucketCount
};

void UpdateBucket(Bucket bucket)
{
    // ...

    for (each gameObject in bucket)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses
     (bucket, dt);

    for (each gameObject in bucket)
    {
        gameObject.PostAnimUpdate(dt);
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls(bucket);
    g_physicsEngine.Simulate(bucket, dt); // ragdolls etc.
    g_collisionEngine.DetectAndResolveCollisions
     (bucket, dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons(bucket);
    g_animationEngine.FinalizePoseAndMatrixPalette
     (bucket);

    for (each gameObject in bucket)
    {
        gameObject.FinalUpdate(dt);
    }
```

```
    // ...
}

void RunGameLoop()
{
    while (true)
    {
        // ...

        UpdateBucket(kBucketVehiclesAndPlatforms);
        UpdateBucket(kBucketCharacters);
        UpdateBucket(kBucketAttachedObjects);

        // ...

        g_renderingEngine.RenderSceneAndSwapBuffers();
    }
}
```

实际情况可能比这更复杂一些。例如，某些引擎子系统（例如物理引擎）可能不支持“存储桶”的概念，可能是因为它们是第三方 SDK，或者因为它们实际上无法以“存储桶”的方式进行更新。然而，顽皮狗基本上就是用这种“存储桶”更新来实现《神秘海域》系列以及《最后生还者》所有游戏的。因此，这种方法已被证明是实用且相当高效的。

16.6.3.3 对象状态不一致和一帧延迟

让我们重新讨论游戏对象的更新，但这次从每个对象的局部时间概念的角度来思考。我们在 16.6 节中提到，游戏对象 i 在时间 t 的状态可以用状态向量 S i（t）表示。当我们更新一个游戏对象时，我们会将其先前的状态向量 S i（t 1）转换为新的当前状态向量 S i（t 2）（其中 t 2 = t 1 + Δ t）。

理论上，所有游戏对象的状态都会从时间 t 1 到时间 t 2 即时并行更新，如图 1 6.15 所示。但是，假设我们的游戏更新循环是单线程的，我们实际上是逐个更新对象——我们循环遍历每个游戏对象，并依次对每个对象调用某种更新函数。如果我们在这个更新循环中途停止程序，那么一半的游戏对象的状态将更新为 S i（t 2），而另一半仍将保持其先前的状态 S i（t 1）。这意味着，如果我们在更新循环期间询问两个游戏对象当前时间是多少，它们可能会或可能不会一致！而且，根据我们中断更新循环的具体位置，对象


```
    // ...
}

void RunGameLoop()
{
    while (true)
    {
        // ...

        UpdateBucket(kBucketVehiclesAndPlatforms);
        UpdateBucket(kBucketCharacters);
        UpdateBucket(kBucketAttachedObjects);

        // ...

        g_renderingEngine.RenderSceneAndSwapBuffers();
    }
}
```

In practice, things might be a bit more complex than this. For example, some engine subsystems like the physics engine might not support the concept of buckets, perhaps because they are third-party SDKs or because they cannot be practically updated in a bucketed manner. However, this bucketed update is essentially what we used at Naughty Dog to implement all of the games in the *Uncharted* series as well as *The Last of Us*. So it's a method that has proven to be practical and reasonably efficient.

16.6.3.3 Object State Inconsistencies and One-Frame-Off Lag

Let's revisit game object updating, but this time thinking in terms of each object's local notion of time. We said in Section 16.6 that the state of game object i at time t can be denoted by a state vector $\mathbf{S}_i(t)$. When we update a game object, we are converting its previous state vector $\mathbf{S}_i(t_1)$ into a new current state vector $\mathbf{S}_i(t_2)$ (where $t_2 = t_1 + \Delta t$).

In theory, the states of all game objects are updated from time t_1 to time t_2 instantaneously and in parallel, as depicted in Figure 16.15. However, presuming that our game update loop is single-threaded, we actually update the objects one by one—we loop over each game object and call some kind of update function on each one in turn. If we were to stop the program halfway through this update loop, half of our game objects' states would have been updated to $\mathbf{S}_i(t_2)$, while the remaining half would still be in their previous states $\mathbf{S}_i(t_1)$. This implies that if we were to ask two of our game objects what the current time is during the update loop, they may or may not agree! What's more, depending on where exactly we interrupt the update loop, the objects

图 16.15。理论上，所有游戏对象的状态都会在游戏循环的每次迭代期间即时并行更新。

可能全部处于部分更新状态。例如，动画姿势混合可能已经运行，但物理和碰撞解决可能尚未应用。这引出了以下规则：

> 所有游戏对象的状态在更新循环前后都是一致的，但在更新循环期间可能不一致。

如图 16.16 所示。

更新循环期间游戏对象状态的不一致是造成混乱和错误的主要原因，甚至对于游戏行业的专业人士来说也是如此。

图 16.16。实际上，游戏对象的状态是逐个更新的。这意味着在更新循环中的某个任意时刻，一些对象会认为当前时间是 t 2 ，而其他对象则认为当前时间仍为 t 1 。有些对象可能只进行了部分更新，因此它们的状态在内部会不一致。实际上，此类对象的状态位于 t 1 和 t 2 之间的某个点。

Figure 16.15. In theory, the states of all game objects are updated instantaneously and in parallel during each iteration of the game loop.

may all be in a partially updated state. For example, animation pose blending may have been run, but physics and collision resolution may not yet have been applied. This leads us to the following rule:

> The states of all game objects are consistent *before* and *after* the update loop, but they may be inconsistent *during* it.

This is illustrated in Figure 16.16.

The inconsistency of game object states during the update loop is a major source of confusion and bugs, even among professionals within the game in-

Figure 16.16. In practice, the states of the game objects are updated one by one. This means that at some arbitrary moment during the update loop, some objects will think the current time is t_2 while others think it is still t_1. Some objects may be only partially updated, so their states will be internally inconsistent. In effect, the state of such an object lies at a point between t_1 and t_2.

行业。当游戏对象在更新循环期间相互查询状态信息时（这意味着它们之间存在依赖关系），问题最常出现。例如，如果对象 B 查看对象 A 的速度以确定其在时间 t 的速度，则程序员必须清楚他或她是否想要读取对象 A 的先前状态 SA（t 1），还是新状态 SA（t 2）。如果需要新状态但对象 A 尚未更新，则会出现更新顺序问题，这会导致一类称为一帧滞后延迟的错误。在这种类型的错误中，一个对象的状态比其对等体的状态滞后一帧，这在屏幕上表现为游戏对象之间缺乏同步。

16.6.3.4 对象状态缓存

如上所述，解决此问题的一种方法是将游戏对象分组到存储桶中（第 16.6.3.2 节）。简单的分桶更新方法的一个问题是，它对游戏对象相互查询状态信息的方式施加了一些任意的限制。如果游戏对象 A 想要另一个对象 B 的更新状态向量 SB（t 2），那么对象 B 必须位于先前更新的存储桶中。同样，如果对象 A 想要对象 B 的先前状态向量 SB（t 1），那么对象 B 必须位于尚未更新的存储桶中。对象 A 永远不应该请求其自己存储桶中对象的状态向量，因为正如我们在上面的规则中所述，这些状态向量可能只是部分更新。或者充其量，对于您是在时间 t 1 还是时间 t 2 访问另一个对象的状态存在一些不确定性。

提高一致性的一种方法是安排每个游戏对象在计算其新状态向量 S i（t 2）时缓存其先前的状态向量 S i（t 1），而不是在更新期间就地覆盖它。这有两个直接的好处。首先，它允许任何对象安全地查询任何其他对象的先前状态向量，而无需考虑更新顺序。其次，它保证即使在更新新状态向量期间，也始终有一个完全一致的状态向量（S i（t 1））可用。据我所知，这种技术没有标准术语，所以由于缺乏更好的名称，我将其称为状态缓存。

状态缓存的另一个好处是，我们可以在前一个状态和下一个状态之间进行线性插值，从而近似估算出物体在这两个时间点之间任意时刻的状态。Havok 物理引擎正是为此目的而维护了模拟中每个刚体的先前状态和当前状态。

状态缓存的缺点是，它消耗的内存是就地更新方法的两倍。它也只能解决一半的问题，因为

dustry. The problem rears its head most often when game objects query one another for state information during the update loop (which implies that there is a dependency between them). For example, if object B looks at the velocity of object A in order to determine its own velocity at time t, then the programmer must be clear about whether he or she wants to read the *previous* state of object A, $\mathbf{S}_A(t_1)$, or the *new* state, $\mathbf{S}_A(t_2)$. If the new state is needed but object A has not yet been updated, then we have an update order problem that can lead to a class of bugs known as *one-frame-off lags*. In this type of bug, the state of one object lags one frame behind the states of its peers, which manifests itself on-screen as a lack of synchronization between game objects.

16.6.3.4 Object State Caching

As described above, one solution to this problem is to group the game objects into buckets (Section 16.6.3.2). One problem with a simple bucketed update approach is that it imposes somewhat arbitrary limitations on the way in which game objects are permitted to query one another for state information. If a game object A wants the *updated* state vector $\mathbf{S}_B(t_2)$ of another object B, then object B must reside in a *previously updated* bucket. Likewise, if object A wants the *previous* state vector $\mathbf{S}_B(t_1)$ of object B, then object B must reside in a *yet-to-be-updated* bucket. Object A should never ask for the state vector of an object within its own bucket, because, as we stated in the rule above, those state vectors may be only partially updated. Or at best, there's some uncertainty as to whether you're accessing the other object's state at time t_1 or time t_2.

One way to improve consistency is to arrange for each game object to *cache* its previous state vector $\mathbf{S}_i(t_1)$ while it is calculating its new state vector $\mathbf{S}_i(t_2)$ rather than *overwriting* it in-place during its update. This has two immediate benefits. First, it allows any object to safely query the previous state vector of any other object without regard to update order. Second, it guarantees that a totally consistent state vector ($\mathbf{S}_i(t_1)$) will always be available, even during the update of the new state vector. To my knowledge there is no standard terminology for this technique, so I'll call it *state caching* for lack of a better name.

Another benefit of state caching is that we can linearly interpolate between the previous and next states in order to approximate the state of an object at any moment between these two points in time. The Havok physics engine maintains the previous and current state of every rigid body in the simulation for just this purpose.

The downside of state caching is that it consumes twice the memory of the update-in-place approach. It also only solves half the problem, because while

尽管 t 1 时刻的先前状态完全一致，但 t 2 时刻的新状态仍然存在潜在的不一致性。尽管如此，如果运用得当，该技术仍然非常有用。

这种技术深受纯函数式编程设计原则的影响（参见第 16.9.2 节）。在纯函数式编程语言中，所有操作都由具有明确输入和输出且没有副作用的函数执行。所有数据都被视为常量且不可变——它不会就地改变输入数据，而是始终生成一个全新的数据。

16.6.3.5 时间戳

提高游戏对象状态一致性的一个简单且低成本的方法是为其添加时间戳。这样一来，判断游戏对象的状态向量是否与其先前时间或当前时间的配置相对应就变得轻而易举了。任何在更新循环中查询其他游戏对象状态的代码都可以断言或显式检查时间戳，以确保获取正确的状态信息。

时间戳并不能解决存储桶更新过程中状态不一致的问题。但是，我们可以设置一个全局或静态变量来反映当前正在更新哪个存储桶。假设每个游戏对象都“知道”自己位于哪个存储桶中。因此，我们可以将查询的游戏对象的存储桶与当前正在更新的存储桶进行比对，并断言它们不相等，以防止出现不一致的状态查询。

16.7 将并发应用于游戏对象更新

在第 4 章中，我们探讨了硬件并行性以及如何利用显式并行计算硬件（这已成为近年来游戏硬件的标配）。这可以通过并发编程技术来实现。在第 8.6 节中，我们介绍了一些允许游戏引擎利用并行处理资源的方法。在本节中，我们将探讨如何将并发性和并行性应用于更新游戏对象状态的问题。

16.7.1 并发引擎子系统

显然，我们引擎中最关键的性能部分——例如渲染、动画、音频和物理——将从并行处理中受益最多。因此，无论我们的游戏对象模型是否在

the previous states at time t_1 are fully consistent, the new states at time t_2 still suffer from potential inconsistency. Nonetheless, the technique can be useful when applied judiciously.

This technique is influenced strongly by the design principles of pure *functional programming* (see Section 16.9.2). In a pure functional programming language, all operations are performed by functions with a clear input and output, and no side-effects. All data is considered constant and immutable—rather than mutating an input datum in place, a brand new datum is always produced.

16.6.3.5 Time-Stamping

One easy and low-cost way to improve the consistency of game object states is to time-stamp them. It is then a trivial matter to determine whether a game object's state vector corresponds to its configuration at a previous time or the current time. Any code that queries the state of another game object during the update loop can assert or explicitly check the time stamp to ensure that the proper state information is being obtained.

Time-stamping does not address the inconsistency of states during the update of a bucket. However, we can set a global or static variable to reflect which bucket is currently being updated. Presumably every game object "knows" in which bucket it resides. So we can check the bucket of a queried game object against the currently updating bucket and assert that they are not equal in order to guard against inconsistent state queries.

16.7 Applying Concurrency to Game Object Updates

In Chapter 4 we explored *hardware parallelism* and how to take advantage of the explicitly parallel computing hardware that has become the norm in recent gaming hardware. This is done via *concurrent programming* techniques. In Section 8.6, we introduced a number of approaches that allow a game engine to take advantage of the parallel processing resources. In this section, we'll take a look at ways in which concurrency and parallelism can be applied to the problem of updating the states of our game objects.

16.7.1 Concurrent Engine Subsystems

Clearly, the most performance-critical parts of our engine—such as rendering, animation, audio and physics—are the ones that will benefit most from parallel processing. So, whether or not our game object model is being updated in a

单线程或跨多核，它需要能够与几乎肯定是多线程的低级引擎系统交互。

如果我们的引擎支持通用作业系统（参见第 8.6.4 节），我们可以使用该作业系统使引擎子系统并发执行。在这种情况下，每个子系统的帧更新可能每帧都作为单个作业启动。但是，每个子系统每帧启动多个作业来执行其工作可能是一个更好的主意。例如，动画系统可能会为游戏世界中每个需要动画混合的对象启动一个作业。在帧的后期，当动画系统执行其世界矩阵和蒙皮矩阵计算时，它可能会采用分散/聚集方法将这项工作分配到可用的核心上。

当我们的底层引擎系统并发更新时，我们需要确保它们的接口是线程安全的。我们希望确保任何外部代码都不会陷入数据竞争的情况，无论是与其他外部客户端争用，还是与子系统本身的内部工作机制争用。这通常需要对每个子系统的所有外部调用使用锁。

如果我们的作业系统使用用户级线程（协程或纤程），那么我们需要使用自旋锁来确保子系统线程安全。但是，如果我们的子系统由操作系统线程更新，那么互斥锁也是一个可行的选择。

如果我们可以确定某个引擎子系统仅在游戏循环的某个特定阶段运行，我们或许可以使用 lock-notneeded 断言，而不必实际锁定子系统的某些部分。有关这些有用断言的更多信息，请参阅第 4.9.7.5 节。

当然，另一种选择是尝试使用无锁数据结构来实现我们引擎子系统的关键（共享）数据。无锁数据结构的实现比较棘手，而且有些数据结构至今还没有已知的无锁实现。因此，如果您选择无锁方法，最好将工作范围限制在性能要求最严格的引擎子系统上。

16.7.2 异步程序设计

当与并发引擎子系统交互时（例如，在游戏对象更新过程中），我们必须开始异步思考。因此，当要执行耗时操作时，我们应该避免调用阻塞函数——该函数直接在调用线程的上下文中执行工作，从而阻塞该线程或作业，直到工作完成。相反，只要有可能，就应该通过调用非阻塞函数来请求大型或高开销的作业——该函数将请求发送到

single thread or across multiple cores, it needs to be able to *interface* with low-level engine systems that are almost certainly multithreaded.

If our engine supports a general-purpose *job system* (see Section 8.6.4), we can use that job system to make our engine subsystems execute concurrently. In this scenario, each subsystem's frame update might be kicked off as a single job each frame. However, it's probably a better idea for each subsystem to kick off multiple jobs to perform its work each frame. For example, an animation system might kick off one job for each object in the game world that requires animation blending. Later in the frame, when the animation system performs its world matrix and skinning matrix computations, it could potentially employ a scatter/gather approach to divide this work across the available cores.

When making our low-level engine systems update concurrently, we need to ensure that their *interfaces* are *thread-safe*. We want to be sure that no external code can get into a data race situation, either in contention with other external clients, or in contention with the inner workings of the subsystem itself. This typically involves using locks on all external calls into each subsystem.

If our job system employs *user-level threads* (coroutines or fibers), then we will need to use spin locks to make our subsystems thread-safe. However, if our subsystems are updated by OS threads, then mutexes are also a viable option for this purpose.

If we can be certain that a particular engine subsystem only ever runs during one particular phase of the game loop, we may be able to employ *lock-not-needed assertions* rather than actually having to lock certain parts of the subsystem. See Section 4.9.7.5 for more information on these useful assertions.

Of course, another option is to attempt to use *lock-free* data structures to implement our engine subsystems critical (shared) data. Lock-free data structures are tricky to implement, and some data structures still have no known lock-free implementation. As such, if you opt for a lock-free approach, it's probably wise to restrict your efforts only to engine subsystems with the most stringent performance requirements.

16.7.2 Asynchronous Program Design

When interfacing with concurrent engine subsystems (for example from within a game object update), we must begin thinking *asynchronously*. When a time-consuming operation is to be performed, we should therefore avoid calling a *blocking function*—a function that does its work directly in the context of the calling thread, thereby blocking that thread or job until the work has been completed. Instead, whenever possible, large or expensive jobs should be requested by calling a *non-blocking function*—a function that sends the request

由另一个线程、核心或处理器执行，然后立即将控制权返回给调用函数。在我们等待请求结果的同时，调用线程或作业可以继续执行其他不相关的工作，例如更新其他游戏对象。稍后在同一帧或下一帧中，我们可以获取请求的结果并加以利用。

例如，游戏可能会请求将射线投射到场景中，以确定玩家是否在敌方角色的视线范围内。在同步设计中，射线投射会立即响应请求，并且当射线投射函数返回时，结果即可获取，如下所示。

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r = castRay(playerPos, enemyPos);

    // Now process the results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

在异步设计中，光线投射请求可以通过调用一个函数来发出，该函数只需设置并加入光线投射作业，然后立即返回。调用线程或作业可以在其他 CPU 或核心处理该作业时继续执行其他不相关的工作。之后，一旦光线投射作业完成，调用线程或作业就可以获取光线投射查询的结果并进行处理：

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r;
    requestRayCast(playerPos, enemyPos, &r);
```


to be executed by another thread, core or processor and then immediately returns control to the calling function. The calling thread or job can proceed with other unrelated work, perhaps including updating other game objects, while we wait for the results of our request. Later in the same frame, or in next frame, we can pick up the results of the request and make use of them.

For example, a game might request that a ray be cast into the world in order to determine whether the player has line-of-sight to an enemy character. In a synchronous design, the ray cast would be done immediately in response to the request, and when the ray casting function returned, the results would be available, as shown below.

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r = castRay(playerPos, enemyPos);

    // Now process the results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

In an asynchronous design, a ray cast request would be made by calling a function that simply sets up and enqueues a ray cast job, and then returns immediately. The calling thread or job can continue doing other unrelated work while the job is being processed by another CPU or core. Later, once the ray cast job has been completed, the calling thread or job can pick up the results of the ray cast query and process them:

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r;
    requestRayCast(playerPos, enemyPos, &r);
```

```
    // Do other unrelated work while we wait for the
    // other CPU to perform the ray cast for us.

    // ...

    // OK, we can't do any more useful work. Wait for the
    // results of our ray cast job. If the job is
    // complete, this function will return immediately.
    // Otherwise, the main thread will idle until the
    // results are ready...
    waitForRayCastResults(&r);

    // Process results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

在许多情况下，异步代码可以在某一帧发起请求，然后在下一帧获取结果。在这种情况下，您可能会看到如下代码：

```
RayCastResult r;
bool rayJobPending = false;

SomeGameObject::Update()
{
    // ...

    // Wait for the results of last frame's ray cast job.
    if (rayJobPending)
    {
        waitForRayCastResults(&r);

        // Process results...
        if (r.hitSomething() && isEnemy(r.getHitObject()))
        {
            // Player can see the enemy.
            // ...
        }
    }
```



```
    // Do other unrelated work while we wait for the
    // other CPU to perform the ray cast for us.

    // ...

    // OK, we can't do any more useful work. Wait for the
    // results of our ray cast job. If the job is
    // complete, this function will return immediately.
    // Otherwise, the main thread will idle until the
    // results are ready...
    waitForRayCastResults(&r);

    // Process results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

In many instances, asynchronous code can kick off a request on one frame, and pick up the results on the next. In this case, you may see code that looks like this:

```
RayCastResult r;
bool rayJobPending = false;

SomeGameObject::Update()
{
    // ...

    // Wait for the results of last frame's ray cast job.
    if (rayJobPending)
    {
        waitForRayCastResults(&r);

        // Process results...
        if (r.hitSomething() && isEnemy(r.getHitObject()))
        {
            // Player can see the enemy.
            // ...
        }
    }
```

```
        // Cast a new ray for next frame.
        rayJobPending = true;
        requestRayCast(playerPos, enemyPos, &r);

        // Do other work...
        // ...
    }
```

16.7.2.1 何时发出异步请求

将同步、非批处理代码转换为异步、批处理方法的一个特别棘手的方面是确定游戏循环中的何时 (a) 启动请求以及 (b) 等待并利用结果。在执行此操作时，问自己以下问题通常会有所帮助：

- 我们最早什么时候可以启动这个请求？我们越早发出请求，它就越有可能在我们真正需要结果时完成——这有助于确保主线程永远不会闲置等待异步请求完成，从而最大限度地提高 CPU 利用率。因此，对于任何给定的请求，我们应该确定帧中最早拥有足够信息来启动它的点，并在那里启动它。

- 我们需要等待多久才能获得此请求的结果？也许我们可以等到更新循环的后期再执行操作的后半部分。也许我们可以容忍一帧的延迟，并使用上一帧的结果来更新本帧的对象状态。（某些子系统，例如人工智能，甚至可以容忍更长的延迟时间，因为它们每隔几秒才更新一次。）在许多情况下，使用请求结果的代码实际上可以推迟到帧的后期，只需稍加思考、进行一些代码重构，并可能对中间数据进行一些额外的缓存。

16.7.3 作业依赖性和并行度

为了充分利用并行计算平台，我们希望所有核心始终保持繁忙状态。假设我们使用作业系统来并行化引擎，那么游戏循环的每次迭代都将由数百甚至数千个并发运行的作业组成。然而，这些作业之间的依赖关系可能会导致可用核心的利用率不理想。如果作业 B 的输入是作业 A 生成的数据，那么作业 B 必须在作业 A 完成之前才能开始。这就造成了作业 B 和作业 A 之间的依赖关系。

系统的并行度（DOP），也称为并发度（DOC），衡量的是理论上可以执行的最大作业数。


```
        // Cast a new ray for next frame.
        rayJobPending = true;
        requestRayCast(playerPos, enemyPos, &r);

        // Do other work...
        // ...
    }
```

16.7.2.1 When to Make Asynchronous Requests

One particularly tricky aspect of converting synchronous, unbatched code to use an asynchronous, batched approach is determining *when* during the game loop (a) to kick off the request and (b) to wait for and utilize the results. In doing this, it is often helpful to ask ourselves the following questions:

- *How early can we kick off this request?* The earlier we make the request, the more likely it is to be done when we actually need the results—and this maximizes CPU utilization by helping to ensure that the main thread is never idle waiting for an asynchronous request to complete. So for any given request, we should determine the earliest point during the frame at which we have enough information to kick it off, and kick it there.
- *How long can we wait before we need the results of this request?* Perhaps we can wait until later in the update loop to do the second half of an operation. Perhaps we can tolerate a one-frame lag and use last frame's results to update the object's state this frame. (Some subsystems like AI can tolerate even longer lag times because they update only every few seconds.) In many circumstances, code that uses the results of a request can in fact be deferred until later in the frame, given a little thought, some code refactoring, and possibly some additional caching of intermediate data.

16.7.3 Job Dependencies and Degree of Parallelism

In order to make the best use of a parallel computing platform, we'd like to keep all cores busy at all times. Presuming we're using a job system to parallelize our engine, each iteration of our game loop will be comprised of hundreds or even thousands of jobs running concurrently. However, *dependencies* between these jobs can lead to less-than-ideal utilization of the available cores. If job B takes as its input some data that is produced by job A, then job B cannot begin until job A is done. This creates a *dependency* between job B and job A.

The *degree of parallelism* (DOP) of a system, also known as its *degree of concurrency* (DOC), measures the theoretical maximum number of jobs that can

图 16.17。三棵作业依赖树。树的节点表示作业，箭头表示它们之间的依赖关系。树中叶子节点的数量表示系统的并行度 (DOP)。

在任何给定时刻并行运行。一组作业的并行度可以通过绘制依赖关系图来确定。在这样的图中，作业构成树的节点，父子关系表示依赖关系。树叶的数量告诉我们该组作业的并行度。图 16.17 展示了一些作业依赖关系树及其对应并行度的示例。

为了在显式并行计算机中充分利用 CPU 资源，我们希望系统的 DOP 能够等于或超过可用核心数。如果软件的 DOP 恰好等于核心数，则吞吐量最大。当系统的 DOP 高于核心数时，吞吐量会降低，因为某些作业必须串行运行，但没有核心处于空闲状态。但是，当系统的 DOP 小于核心数时，某些核心将无事可做。

每当一个作业被迫等待其依赖的作业完成任务时，系统就会引入一个同步点（或简称“同步点”）。每个同步点都意味着宝贵的 CPU 资源在作业等待其依赖的作业完成工作时被浪费。如图 16.18 所示。

为了最大限度地利用硬件，我们可以尝试增加

Figure 16.17. Three job dependency trees. The nodes of the tree are jobs, and the arrows represent dependencies between them. The number of leaves in such a tree indicates the system's degree of parallelism (DOP).

be running in parallel at any given moment in time. The degree of parallelism of a group of jobs can be determined by drawing a *dependency graph*. In such a graph, the jobs form the nodes of the tree, and the parent-child relationships represent dependencies. The number of leaves of the tree tells us the degree of parallelism of that collection of jobs. Figure 16.17 illustrates a few examples of job dependency trees and their corresponding degrees of parallelism.

In order to achieve full utilization of CPU resources in an explicitly parallel computer, we would like the DOP of our system to match or exceed the number of available cores. If the DOP of the software exactly equals the number of cores, we get maximum throughput. When the DOP of our system is higher than the number of cores, throughput is reduced because some jobs must run serially, but no core is idle. But when our system's DOP is less than the number of cores, some cores won't have any work to do.

Whenever a job is forced to wait for the jobs on which it depends to complete their tasks, a *synchronization point* (or "sync point" for short) is introduced into the system. Each sync point represents an opportunity for valuable CPU resources to be wasted while a job waits for its dependent jobs to complete their work. This is illustrated in Figure 16.18.

To maximize hardware utilization, we can attempt to increase the DOP of

图 16.18。每当一个作业依赖于一个或多个其他作业的完成时，就会引入一个同步点。这里，作业 D 依赖于作业 C，作业 F 依赖于作业 A、B、D 和 E。

通过减少作业之间的依赖关系来优化我们的系统。我们还可以尝试在空闲期间找一些其他不相关的工作来做。我们还可以通过推迟同步点来减少或消除同步点的影响。例如，假设作业 D 直到作业 A、B 和 C 完成才能开始工作。如果我们试图在 A、B 和 C 全部完成之前安排作业 D，那么它显然必须闲置一段时间。但是，如果我们推迟作业 D，使其在 A、B 和 C 完成后运行，那么我们可以确信 D 永远不必等待。图 16.19 说明了这个想法。Mike Acton 在他的题为“深入并发兔子洞”（http://cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf）的演讲中说：“优化并发设计的秘诀是延迟。”这就是他所说的：推迟同步点作为减少或消除并发系统中空闲时间的一种手段。

16.7.4 并行化游戏对象模型本身

众所周知，游戏对象模型很难并行化，原因如下。游戏对象往往高度相互依赖，并且通常依赖于众多引擎子系统。对象间依赖关系的产生是因为游戏对象在更新过程中会定期相互通信并查询彼此的状态。这些交互往往会在更新循环中多次发生，并且通信模式可能难以预测，并且对人类玩家的输入和游戏世界中发生的事件高度敏感。这使得并发处理游戏对象更新（使用多线程或多 CPU 核心）变得非常困难。

话虽如此，当然可以更新游戏对象模型，

Figure 16.18. A synchronization point is introduced whenever one job is dependent upon the completion of one or more other jobs. Here, job D depends on job C, and job F depends on jobs A, B, D and E.

our system by reducing dependencies between jobs. We could also try to find some other unrelated work to do during idle periods. We can also reduce or eliminate the impact of a sync point by *deferring* it. For example, let's say that job D cannot commence its work until jobs A, B and C are done. If we try to schedule job D before A, B and C are all done, then it will obviously have to sit idle for some time. But if we defer job D so that it runs well after A, B and C are done, then we can be confident that D will never have to wait. This idea is illustrated in Figure 16.19. In his talk entitled "Diving Down the Concurrency Rabbit Hole" (http://cellperformance.beyond3d.com/articles/public/concurrency_rabit_hole.pdf), Mike Acton says, "The secret of optimized concurrent design is *delay*." This is what he's talking about: Deferring sync points as a means of reducing or eliminating idle time in a concurrent system.

16.7.4 Parallelizing the Game Object Model Itself

Game object models are notoriously difficult to parallelize for a few reasons. Game objects tend to be highly interdependent, and are typically dependent on numerous engine subsystems as well. Inter-object dependencies arise because game objects routinely communicate with one another and query one another's states during their updates. These interactions tend to occur multiple times during the update loop, and the pattern of communication can be unpredictable and highly sensitive to the inputs of the human player and the events that are occurring in the game world. This makes it difficult to process game object updates *concurrently* (using multiple threads or multiple CPU cores).

That being said, it certainly is possible to update a game object model con-

图 16.19。作业 D 依赖于作业 A、B 和 C。上图：如果我们尝试在核心 2 上将作业 D 调度到作业 C 之后立即执行，核心 2 将处于空闲状态，等待作业 B 完成。下图：如果我们将作业 D 的调用延迟到作业 A、B 和 C 完成后，核心 2 就可以释放出来运行其他作业，从而避免同步点造成的空闲时间。

目前。举个例子，顽皮狗在我们将《最后生还者：重制版》的引擎从 PS3 移植到 PS4 时，实现了一个并发游戏对象模型。在本节中，我们将探讨并发更新游戏对象时遇到的一些问题，并看看顽皮狗在我们的引擎中使用的一些解决方案。当然，这些技术只是解决问题的一种可能方法——其他引擎可能会使用不同的方法。谁知道呢？也许你会发明一种新方法来解决并发游戏对象模型更新的一些棘手问题！

16.7.4.1 游戏对象更新作为作业

如果我们的游戏引擎有一个作业系统，它很可能会用于并行化引擎中的各种低级子系统，例如动画、音频、碰撞/物理、渲染、文件 I/O 等等。那么，为什么不将游戏对象的更新也变成作业来并行化呢？顽皮狗引擎就采用了这种方法。这种方法行之有效，但要让它正确高效地运行，并非易事。

Figure 16.19. Job D depends on jobs A, B and C. Top: If we attempt to schedule job D immediately after job C on Core 2, the core will sit idle waiting for job B to complete. Bottom: If we delay job D's invocation until well after jobs A, B and C have completed, we free up Core 2 to run other jobs, thereby avoiding the idle time that had been caused by the sync point.

currently. As one example, Naughty Dog implemented a concurrent game object model when we ported our engine from PS3 to PS4 for *The Last of Us: Remastered*. In this section, we'll explore some of the problems one encounters when updating game objects concurrently, and we'll have a look at some of the solutions that Naughty Dog uses in our engine. Of course, these techniques are just one possible way to tackle the problem—other engines may use different approaches. And who knows? Perhaps *you* will invent a novel way of solving some of the vexing problems of concurrent game object model updates!

16.7.4.1 Game Object Updates as Jobs

If our game engine has a job system, it will very likely be used to parallelize the various low-level subsystems in our engine, such as animation, audio, collsion/physics, rendering, file I/O, and so on. So why not parallelize our game object updates by making them jobs, too? This is the approach used in the Naughty Dog engine. It works, but getting it to work correctly, and efficiently, is a non-trivial undertaking.

我们在 16.6.3.2 节中讨论过，由于游戏对象之间存在相互依赖关系，我们需要控制它们的更新顺序。实现此目的的一种方法是将游戏中的所有对象分成 N 个存储桶，使得存储桶 B 中的对象仅依赖于存储桶 0 到 B - 1 中的对象。如果我们采用这种方法，我们可以通过将每个存储桶中的所有游戏对象作为作业启动来更新它们，并让作业系统将它们调度到可用的核心上（并与当时正在运行的其他作业穿插执行）。这大致是顽皮狗引擎中使用的技术。

```
void UpdateBucket(int iBucket)
{
    job::Declaration aDecl[kMaxObjectsPerBucket];
    const int count = GetNumObjectsInBucket(iBucket);
    for (int jObject = 0; jObject < count; ++jObject)
    {
        job::Declaration& decl = aDecl[jObject];
        decl.m_pEntryPoint = UpdateGameObjectJob;
        decl.m_param = reinterpret_cast<uintptr_t>(
            GetGameObjectInBucket(iBucket, jObject));
    }
    job::KickJobsAndWait(aDecl, count);
}
```

处理对象间依赖关系的另一种方法是使其显式化。在这种情况下，我们大概会通过某种方式声明游戏对象 A 依赖于游戏对象 B、C 和 D。这些依赖关系将形成一个简单的有向图。要更新游戏对象，我们首先启动所有不依赖于任何其他游戏对象的游戏对象的更新作业（即，依赖图中没有出边的节点）。每个作业完成后，我们将遍历每个依赖其的游戏对象，并等待其所有依赖对象的更新作业完成。此过程将重复，直到整个游戏对象图都更新完毕。

如果游戏对象依赖关系图包含任何循环，我们可能会遇到麻烦。涉及两个或多个游戏对象的依赖循环表明这些对象无法简单地按依赖顺序更新。为了处理循环，我们要么需要通过改变对象交互的方式来消除它们（将图转换为有向无环图，即 DAG），要么需要隔离这些循环依赖的游戏对象“块”，并在单个核心上串行更新每个块。

We discussed in Section 16.6.3.2 that, due to interdependencies between game objects, we need to control the order in which they update. One way to achieve this is to divide all of the objects in the game into N buckets, such that the objects in bucket B depend only on objects in buckets 0 through $B-1$. If we take this approach, we could update all the game objects in each bucket by kicking them off as jobs, and let the job system schedule them across the available cores (and interspersed with whatever other jobs happen to be running at the time). This is roughly the technique employed in the Naughty Dog engine.

```
void UpdateBucket(int iBucket)
{
    job::Declaration aDecl[kMaxObjectsPerBucket];
    const int count = GetNumObjectsInBucket(iBucket);
    for (int jObject = 0; jObject < count; ++jObject)
    {
        job::Declaration& decl = aDecl[jObject];
        decl.m_pEntryPoint = UpdateGameObjectJob;
        decl.m_param = reinterpret_cast<uintptr_t>(
            GetGameObjectInBucket(iBucket, jObject));
    }
    job::KickJobsAndWait(aDecl, count);
}
```

Another way to deal with inter-object dependencies is to make them explicit. In this case, we would presumably have some way of declaring that game object A depends on game objects B, C and D, for example. These dependencies would form a simple *directed graph*. To update the game objects, we'd start by kicking off update jobs for all of the game objects with no dependencies on any other game object (i.e., those nodes of the dependency graph with no outgoing edges). As each job completes, we would walk to each dependent game object, and wait until all of its dependent objects' update jobs are complete. This process would repeat until the entire graph of game objects has been updated.

We can get into trouble with a graph of game object dependencies if the graph contains any *cycles*. A dependency cycle involving two or more game objects indicates that those objects cannot simply be updated in dependency order. To handle cycles, we either need to eliminate them by changing the way the objects interact (turning our graph into a *directed acyclic graph* or DAG), or we need to isolate these "clumps" of cyclically-dependent game objects and update each clump serially on a single core.

16.7.4.2 异步游戏对象更新

我们在 16.7.1 节中提到，游戏对象更新通常是异步完成的。例如，我们不会调用阻塞函数来投射射线，而是发起一个异步的射线投射请求，碰撞子系统会在帧的某个未来时间处理此请求。在帧的稍后时间，也就是下一帧，我们会获取射线投射的结果并对其进行处理。

当游戏对象本身也在同时更新（跨多个核心）时，这种方法仍然有效。但是，如果我们的作业系统基于用户级线程（协程或纤程），那么阻塞调用也将成为可行的选择。这是可行的，因为协程具有能够屈服（给另一个协程）然后在稍后某个时间（当另一个协程回到它时）继续从上次中断处执行的独特属性。在基于纤程的作业系统（例如顽皮狗引擎中使用的系统）中，作业本身并不是协程，但它们具有相同的属性：基于纤程的作业能够在执行过程中“进入睡眠状态”，然后被“唤醒”以继续从上次中断处执行。

下面是在基于协程或纤程的作业系统中使用阻塞调用实现的射线投射示例：

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r = castRayAndWait(playerPos, enemyPos);

    // zzz...

    // Wake up when ray cast result is ready!

    // Now process the results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

请注意，这个实现看起来与我们在 16.7.2 节中展示的“不该做什么”的例子几乎完全相同！得益于用户级线程，

16.7.4.2 Asynchronous Game Object Updates

We said in Section 16.7.1 that game object updates are usually done asynchronously. For example, rather than calling a *blocking function* to cast a ray, we would kick off an asynchronous *request* for a ray cast, and the collision subsystem would process this request at some future time during the frame. Later in the frame, or next frame, we would pick up the results of the ray cast, and act on them.

This approach continues to work well when the game objects themselves are also updating concurrently (across multiple cores). However, if our job system is based on *user-level threads* (coroutines or fibers), then *blocking calls* become a viable option, too. This works because a coroutine has the unique property of being able to *yield* (to another coroutine) and then continue where it left off at some later time (when another coroutine yields back to it). In a fiber-based job system (like the one used in the Naughty Dog engine), jobs aren't coroutines *per se*, but they do have this same property: A fiber-based job is able to "go to sleep" midway through its execution, and later to be "woken up" to continue execution where it left off.

Here's an example ray cast implemented using a blocking call, in a coroutine- or fiber-based job system:

```
SomeGameObject::Update()
{
    // ...

    // Cast a ray to see if the player has line of sight
    // to the enemy.
    RayCastResult r = castRayAndWait(playerPos, enemyPos);

    // zzz...

    // Wake up when ray cast result is ready!

    // Now process the results...
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // Player can see the enemy.
        // ...
    }

    // ...
}
```

Notice that this implementation looks nearly identical to the example of what *not to do* that we presented in Section 16.7.2! Thanks to user-level threads, the

图 16.20 如果我们的作业系统是用用户级线程实现的，则可以在作业执行过程中中断作业以执行异步操作，并在该操作完成后恢复作业。

毕竟，我们的作业执行可以利用阻塞函数调用！图 16.20 说明了正在发生的事情：该作业实际上被分成了两部分——在阻塞射线投射调用之前运行的部分，以及在阻塞射线投射调用之后运行的部分。

这种机制使我们能够实现诸如 WaitForCounter() 和 KickJobsAndWait() 之类的作业系统函数。这些函数会阻塞其作业，使其进入睡眠状态，并允许其他作业在等待相关计数器归零时执行。

16.7.4.3 游戏对象更新期间的锁定

分桶更新对于解决由于对象间依赖性而引起的问题大有帮助。它们确保游戏对象以正确的全局顺序更新（例如，火车车厢在其上的对象之前更新）。并且它们有助于对象间状态查询。桶 B 中的对象可以安全地查询桶 $B - \Delta$ 和桶 $B + \Delta$（其中 $\Delta > 0$）中对象的状态，因为在这两种情况下，我们都知道这些对象不会与桶 B 中的对象同时更新。然而，仍然存在一个一帧不同步的问题：如果在第 N 帧，桶 B 中的对象查询桶 $B - \Delta$ 中的对象，它将看到该对象在第 N 帧的状态；但是，如果它查询桶 $B + \Delta$ 中的对象，它将看到该对象上一帧（在 $N - 1$ 帧）的状态，因为它尚未更新。

因此，有了分桶更新系统，我们可以安全地访问其他桶中的游戏对象，而无需任何锁。但是，当同一个桶中的游戏对象需要相互交互或查询时该怎么办？这时，我们又很容易出现并发竞争条件，所以我们不能什么也不做，只能祈祷好运。

我们可能会尝试在游戏对象系统中引入一个全局锁（互斥锁或自旋锁）。特定存储桶内的每个游戏对象都可以获取此锁，执行其更新（可能与其他对象交互）。

Figure 16.20. If our job system is implemented with user-level threads, a job can be interrupted part-way through its execution in order to perform an asynchronous operation, and resumed once that operation has been completed.

execution of our job can make use of blocking function calls after all! Figure 16.20 illustrates what is happening: The job has effectively been cleaved into two parts—the portion that runs before the blocking ray cast call, and the portion that runs after.

This mechanism is what allows us to implement job system functions like `WaitForCounter()` `KickJobsAndWait()`. These functions *block* their jobs, putting them to sleep and permitting other jobs to execute while they wait for the counter(s) in question to hit zero.

16.7.4.3 Locking During Game Object Updates

Bucketed updates go a long way to solving the problems that arise due to inter-object dependencies. They ensure that the game objects are updated in the correct global order (e.g., the train car updates before the objects sitting on it). And they help with inter-object state queries. An object in bucket B can safely query the state of objects in buckets $B - \Delta$, and those in buckets $B + \Delta$ (where $\Delta > 0$), because in both cases we know that those objects won't be updating concurrently with the objects in bucket B. There is still a one-frame-off issue, however: If on frame N, an object in bucket B queries an object in bucket $B - \Delta$, it will see the state of that object on frame N; however, if it queries an object in bucket $B + \Delta$, it will see the state of that object as it was last frame (on frame $N - 1$) because it won't have updated yet.

So, with a bucketed updating system, we can safely access game objects in *other* buckets without needing any kind of lock. However, what about when game objects in the *same* bucket need to interact or query one another? Here, we are once again prone to concurrent race conditions, so we can't just do nothing and cross our fingers.

We might be tempted to introduce a single, global lock (mutex or spin lock) into the game object system. Every game object within one particular bucket could acquire this lock, perform its update (possibly interacting with other

（例如，在进程中执行游戏对象更新操作），并在完成后释放锁。这当然可以保证对象间通信不会出现数据争用。然而，它也会带来非常不良的影响，即序列化存储桶中所有游戏对象的更新，从而将“并发”游戏对象更新实际上简化为单线程更新！这是因为锁会阻止任何两个游戏对象并发更新，即使这两个对象之间没有任何交互。

有很多其他方法可以解决这个问题。我们早期在顽皮狗尝试过的一种方法是引入一个全局锁定系统，但仅在游戏对象句柄在游戏对象的更新函数中被取消引用时才获取锁。为了支持内存碎片整理，我们引擎中的游戏对象是通过句柄而不是原始指针引用的。因此，要获取指向游戏对象的原始指针，必须先取消引用其句柄。这是一个绝佳的机会来检测一个游戏对象是否打算与另一个游戏对象交互。通过仅在交互可能发生时获取锁，我们能够恢复一定程度的并发性。然而，这个锁定系统很复杂，难以操作，并且在存储桶更新期间仍然会导致 CPU 核心的低效使用。

16.7.4.4 对象快照

通过分析实际游戏引擎中游戏对象之间的相互依赖关系，我们可以观察到：游戏对象在更新过程中的交互绝大多数都是状态查询。换句话说，游戏对象 A 会查询游戏对象 B、C 等的当前状态。当游戏对象 A 执行此操作时，它实际上只需要访问这些其他游戏对象状态的只读副本。它无需与这些对象本身进行交互（在进行此类交互时，这些对象可能正在并发更新，也可能没有）。

鉴于这种情况，为每个游戏对象提供其相关状态信息的快照是合理的——这是一个只读副本，系统中的任何其他游戏对象都可以查询，而无需锁定或担心数据争用。快照实际上只是我们在第 16.6.3.4 节中描述的状态缓存技术的一个例子。顽皮狗在《最后生还者：重制版》《神秘海域 4：盗贼末路》和《神秘海域：失落的遗产》中就使用了这种方法。

顽皮狗引擎中的快照工作原理如下：每次存储桶更新开始时，我们都会要求每个游戏对象更新其快照。这些更新不会查询其他游戏对象的状态，因此它们可以在没有锁的情况下并发运行。所有更新完成后，我们会启动并发作业来更新游戏对象的内部状态。这些作业也同时运行。

game objects in the process), and then release the lock when it's done. This would certainly guarantee that inter-object communication would be free from data races. However, it would also have the highly undesirable effect of *serializing* the updates of all game objects within the bucket, reducing our "concurrent" game object update in effect to a single-threaded update! This occurs because the lock would prevent any two game objects from ever updating concurrently, even when those two objects don't interact in any way.

There are all sorts of other ways to tackle this problem. One approach we tried early on at Naughty Dog was to introduce a global locking system, but to only acquire the lock whenever a game object handle was *dereferenced* within a game object's update function. Game objects in our engine are referenced by handle rather than by raw pointers, to support memory defragmentation. So to get a raw pointer to a game object, one must first dereference its handle. This is a perfect opportunity to detect that one game object intends to interact with another. By only taking locks when interactions are actually likely to happen, we were able to recover some degree of concurrency. However, this locking system was complex, difficult to work with, and still led to inefficient use of the CPU cores during bucket updates.

16.7.4.4 Object Snapshots

Upon analyzing the inter-dependencies between the game objects in a real game engine, we can make an observation: A large majority of the interactions between game objects during their updates are state queries. In other words, game object A reaches out and queries the current state of game objects B, C, and so on. When game object A does this, it really only needs to have access to a *read-only copy* of the states of these other game objects. It needn't interact with the objects themselves (which might or might not be concurrently updating at the moment of such interactions).

Given this state of affairs, it makes sense to arrange for each game object to provide a *snapshot* of its relevant state information—a read-only copy which can be queried, without locks or fear of data races, by any other game object in the system. Snapshots are really just an example of the *state caching* technique we described in Section 16.6.3.4. We used this approach at Naughty Dog for *The Last of Us: Remastered*, *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy*.

Here's how snapshots work in the Naughty Dog engine: At the start of each bucket update, we ask each game object to update its snapshot. These updates never query the state of other game objects, so they can be run concurrently without locks. Once all updates are complete, we kick off concurrent jobs to update the game objects' internal states. These too run concurrently.

每当存储桶 B 中的游戏对象需要查询另一个对象的状态时，它现在有三个选项：

1. 当查询存储桶B-Δ中的对象时，可以直接查询，也可以查询该对象的快照。
2. 当查询存储桶B中的对象时，它会查询快照。
3. 当查询存储桶 B + Δ 中的对象时，它可以再次查询对象本身或其快照。

16.7.4.5 处理对象间变异

快照可以帮助我们在游戏对象读取彼此的游戏状态时避免锁。然而，当一个游戏对象需要修改同一存储桶中另一个对象的状态时，快照无法解决可能发生的数据竞争问题。为了处理这些情况，顽皮狗结合使用了以下经验法则和技术：

- 尽可能减少对象间的变异。
- bucket 之间的对象间变异是安全的，但是
- 必须小心处理存储桶内的对象间变异……
  - 带锁，
  - 或者通过将变更放入请求队列来请求变更，而不是立即应用变更。请求队列本身受锁保护，队列中请求的处理将被推迟到存储桶更新完成后。

单个存储桶内对象间变异的另一种选择是在下一个存储桶中生成一个作业，该作业负责同步这些变异操作。通过将操作推迟到后续的存储桶，我们可以确保相关对象已完成更新，并且不会与我们的工作同时更新。顽皮狗就曾这样做过，以处理玩家和 NPC 之间的同步近战动作。

16.7.4.6 未来的改进

本节中描述的分桶更新系统并非完美无缺。分桶更新的效率并未达到应有的水平，因为每次分桶之间的转换都会在游戏循环中引入一个同步点。在这些同步点，某些 CPU 核心可能会处于空闲状态，等待分桶中的所有游戏对象完成更新。

Whenever a game object in bucket B needs to query the state of another object, it now has three options:

1. When querying an object in buckets $B - \Delta$, it can query it directly or query the object's snapshot.
2. When querying an object in bucket B, it queries the snapshot.
3. When querying an object in buckets $B + \Delta$, it again can query either the object itself, or its snapshot.

16.7.4.5 Handling Inter-Object Mutation

Snapshots allow us to avoid locks when game objects *read* one another's game state. However, they do nothing to address the data races that can occur when one game object needs to *mutate* the state of another object in the same bucket. To handle these situations, Naughty Dog applied a combination of the following rules of thumb and techniques:

- Minimize inter-object mutation wherever possible.
- Inter-object mutation *between* buckets is safe, but
- inter-object mutation *within* a bucket must be handled carefully...
  - with locks,
  - or by requesting mutations by placing them onto a request queue, rather than applying the mutations immediately. The request queue is itself protected by a lock, and the handling of the requests in the queue is deferred until after the bucket update has completed.

Another option for inter-object mutation within a single bucket is to spawn a job in the *next* bucket whose job it is to synchronize these mutation operations. By deferring the action to a later bucket, we can be certain that the objects in question have completed their updated and will not be updating concurrently with our work. We did this at Naughty Dog to handle synchronized melee moves involving the player and an NPC.

16.7.4.6 Future Improvements

The bucketed updating system we've described in this section isn't perfect by any means. Bucketed updates aren't as efficient as they could be, because each transition between buckets introduces a *sync point* into the game loop. At these sync points, some CPU cores may sit idle while we wait for all game objects in the bucket to complete their updates.

快照对于存储桶内依赖关系问题来说也是一个不完美的解决方案，因为它只处理只读查询；对象间的变更仍然需要锁。快照本身的更新成本也可能很高（尽管这里有一个易于应用的优化，即仅在需要时为对象生成快照）。

还有很多其他方法可以解决这些问题。探索如何并发更新游戏对象的最佳方法是进行实验。希望我们在本节中提出的一些有用的想法能够为您接下来的实验铺平道路！

16.8 事件和消息传递

游戏本质上是由事件驱动的。事件是指游戏过程中发生的任何有趣的事情。爆炸发生、玩家被敌人发现、医疗包被拾取——这些都是事件。游戏通常需要一种方法来 (a) 在事件发生时通知感兴趣的游戏对象，以及 (b) 安排这些对象以各种方式响应有趣的事件——我们称之为“事件处理”。不同类型的游戏对象会以不同的方式响应事件。特定类型的游戏对象响应事件的方式是其行为的关键方面，与对象的状态在没有任何外部输入的情况下如何随时间变化同样重要。例如，在《乒乓球》中，球的行为部分取决于其速度，部分取决于它对撞击墙壁或球拍并弹起事件的反应，部分取决于当球被玩家遗漏时会发生什么。

16.8.1 静态类型函数绑定的问题

通知游戏对象事件发生的一个简单方法是直接调用该对象的方法（成员函数）。例如，当爆炸发生时，我们可以查询游戏世界中爆炸伤害半径内的所有对象，然后对每个对象调用一个名为 OnExplosion() 的虚函数。以下伪代码演示了这一过程：

```
void Explosion::Update()
{
    // ...

    if (ExplosionJustWentOff())
    {
        GameObjectCollection damagedObjects;
```


Snapshotting is also an imperfect solution to within-bucket dependency problems, because it only handles read-only queries; locks are still required for inter-object mutations. The snapshots themselves can be expensive to update as well (although one easy-to-apply optimization here is to only generate snapshots for objects on an as-needed basis).

There are lots of other ways to tackle these problems. The best way to discover how to update game objects concurrently is to experiment. Hopefully we've presented some useful ideas in this section that will pave the way forward as you experiment for yourself!

16.8 Events and Message-Passing

Games are inherently event-driven. An *event* is anything of interest that happens during gameplay. An explosion going off, the player being sighted by an enemy, a health pack getting picked up—these are all events. Games generally need a way to (a) notify interested game objects when an event occurs and (b) arrange for those objects to respond to interesting events in various ways—we call this *handling* the event. Different types of game objects will respond in different ways to an event. The way in which a particular type of game object responds to an event is a crucial aspect of its behavior, just as important as how the object's state changes over time in the absence of any external inputs. For example, the behavior of the ball in *Pong* is governed in part by its velocity, in part by how it reacts to the event of striking a wall or paddle and bouncing off, and in part by what happens when the ball is missed by one of the players.

16.8.1 The Problem with Statically Typed Function Binding

One simple way to notify a game object that an event has occurred is to simply call a method (member function) on the object. For example, when an explosion goes off, we could query the game world for all objects within the explosion's damage radius and then call a virtual function named something like `OnExplosion()` on each one. This is illustrated by the following pseudocode:

```
void Explosion::Update()
{
    // ...

    if (ExplosionJustWentOff())
    {
        GameObjectCollection damagedObjects;
```

```
        g_world.QueryObjectsInSphere(GetDamageSphere(),
                                     damagedObjects);

        for (each object in damagedObjects)
        {
            object.OnExplosion(*this);
        }
    }

    // ...
}
```

对 OnExplosion() 的调用是静态类型后期函数绑定的一个示例。函数绑定是确定在特定调用位置调用哪个函数实现的过程 - 该实现实际上绑定到调用。虚函数（例如我们的 OnExplosion() 事件处理函数）被称为后期绑定。这意味着编译器实际上并不知道在编译时将调用该函数的众多可能实现中的哪一个 - 只有在运行时，当目标对象的类型已知时，才会调用适当的实现。我们说虚函数调用是静态类型的，因为编译器知道在给定特定对象类型时应该调用哪个实现。例如，它知道当目标对象是 Tank 时应该调用 Tank::OnExplosion() ，而当对象是 Crate 时应该调用 Crate::OnExplosion() 。

静态类型函数绑定的问题在于它会给我们的实现带来一定程度的不灵活性。首先，虚拟 OnExplosion() 函数要求所有游戏对象都从一个公共基类继承。此外，它还要求该基类声明虚拟函数 OnExplosion()，即使并非所有游戏对象都能响应爆炸。事实上，使用静态类型虚拟函数作为事件处理程序将要求我们的基础 GameObject 类为游戏中所有可能的事件声明虚拟函数！这会使向系统添加新事件变得困难。它阻止以数据驱动的方式创建事件 - 例如，在世界编辑工具中。它也没有为某些类型的对象或某些单独的对象实例提供机制来注册对某些事件的兴趣而不注册其他事件。实际上，游戏中的每个对象都“知道”每个可能的事件，即使它对事件的响应是什么都不做（即实现一个空的、不执行任何操作的事件处理程序函数）。

那么，我们真正需要的事件处理程序是动态类型的后期函数绑定。一些编程语言原生支持此功能（例如 C# 的委托）。在其他语言中，工程师必须自己实现它。


```
        g_world.QueryObjectsInSphere(GetDamageSphere(),
                                     damagedObjects);

        for (each object in damagedObjects)
        {
            object.OnExplosion(*this);
        }
    }

    // ...
}
```

The call to `OnExplosion()` is an example of *statically typed late function binding*. Function binding is the process of determining which function implementation to invoke at a particular call location—the implementation is, in effect, bound to the call. Virtual functions, such as our `OnExplosion()` event-handling function, are said to be *late-bound*. This means that the compiler doesn't actually know *which* of the many possible implementations of the function is going to be invoked at compile time—only at runtime, when the type of the target object is known, will the appropriate implementation be invoked. We say that a virtual function call is *statically typed* because the compiler *does* know which implementation to invoke given a particular object type. It knows, for example, that `Tank::OnExplosion()` should be called when the target object is a `Tank` and that `Crate::OnExplosion()` should be called when the object is a `Crate`.

The problem with statically typed function binding is that it introduces a degree of inflexibility into our implementation. For one thing, the virtual `OnExplosion()` function requires all game objects to inherit from a common base class. Moreover, it requires that base class to *declare* the virtual function `OnExplosion()`, even if not all game objects can respond to explosions. In fact, using statically typed virtual functions as event handlers would require our base `GameObject` class to declare virtual functions for *all possible events* in the game! This would make adding new events to the system difficult. It precludes events from being created in a data-driven manner—for example, within the world editing tool. It also provides no mechanism for certain types of objects, or certain individual object instances, to register interest in certain events but not others. Every object in the game, in effect, "knows" about every possible event, even if its response to the event is to do nothing (i.e., to implement an empty, do-nothing event handler function).

What we really need for our event handlers, then, is *dynamically typed late function binding*. Some programming languages support this feature natively (e.g., C#'s delegates). In other languages, the engineers must implement it

手动操作。解决这个问题的方法有很多，但大多数都归结为数据驱动的方法。换句话说，我们将函数调用的概念封装在一个对象中，并在运行时传递该对象，以实现动态类型的后期绑定函数调用。

16.8.2 将事件封装在对象中

一个事件实际上由两部分组成：事件类型（爆炸、队友受伤、玩家被发现、医疗包拾取等）及其参数。参数提供了事件的具体信息。（爆炸造成了多少伤害？哪位队友受伤了？玩家被发现的位置在哪里？医疗包里还有多少生命值？）我们可以将这两个部分封装在一个对象中，如以下略微简化的代码片段所示：

```
struct Event
{
    const U32 MAX_ARGS = 8;

    EventType  m_type;
    U32        m_numArgs;
    EventArg   m_aArgs[MAX_ARGS];
};
```

一些游戏引擎将这些称为消息或命令，而不是事件。这些名称强调了这样一种观点：向对象通知事件本质上等同于向这些对象发送消息或命令。

实际上，事件对象通常不会这么简单。例如，我们可以通过从根事件类派生出不同类型的事件来实现它们。参数可以实现为链表或动态分配的数组，该数组可以包含任意数量的参数，并且参数的数据类型也可能多种多样。

将事件（或消息）封装在对象中有很多好处：

- 单个事件处理函数。由于事件对象在内部编码其类型，因此任意数量的不同事件类型都可以用单个类（或继承层次结构的根类）的实例来表示。这意味着我们只需要一个虚函数来处理所有类型的事件（例如，virtual void OnEvent(Event& event);）。

- 持久性。与函数调用不同，函数调用的参数在函数返回后会超出作用域，而事件对象会将其类型和参数都存储为数据。因此，事件对象具有持久性。它可以存储在队列中以便稍后处理，也可以复制并广播给多个接收者等等。

manually. There are many ways to approach this problem, but most boil down to taking a data-driven approach. In other words, we encapsulate the notion of a function call in an object and pass that object around at runtime in order to implement a dynamically typed late-bound function call.

16.8.2 Encapsulating an Event in an Object

An event is really comprised of two components: its *type* (explosion, friend injured, player spotted, health pack picked up, etc.) and its *arguments*. The arguments provide specifics about the event. (How much damage did the explosion do? Which friend was injured? Where was the player spotted? How much health was in the health pack?) We can encapsulate these two components in an object, as shown by the following rather over-simplified code snippet:

```
struct Event
{
    const U32 MAX_ARGS = 8;

    EventType  m_type;
    U32        m_numArgs;
    EventArg   m_aArgs[MAX_ARGS];
};
```

Some game engines call these things *messages* or *commands* instead of *events*. These names emphasize the idea that informing objects about an event is essentially equivalent to sending a message or command to those objects.

Practically speaking, event objects are usually not quite this simple. We might implement different types of events by deriving them from a root event class, for example. The arguments might be implemented as a linked list or a dynamically allocated array capable of containing arbitrary numbers of arguments, and the arguments might be of various data types.

Encapsulating an event (or message) in an object has many benefits:

- *Single event handling function*. Because the event object encodes its type internally, any number of different event types can be represented by an instance of a single class (or the root class of an inheritance hierarchy). This means that we only need *one* virtual function to handle *all* types of events (e.g., `virtual void OnEvent(Event& event);`).
- *Persistence*. Unlike a function call, whose arguments go out of scope after the function returns, an event object stores both its type and its arguments as data. An event object therefore has persistence. It can be stored in a queue for handling at a later time, copied and broadcast to multiple receivers and so on.

• 盲事件转发。一个对象可以将其接收到的事件转发给另一个对象，而无需“了解”该事件的任何信息。例如，如果一辆车接收到一个“下车”事件，它可以将其转发给所有乘客，从而允许他们下车，即使车辆本身对下车一无所知。

将事件/消息/命令封装在对象中的想法在计算机科学的许多领域都很常见。它不仅存在于游戏引擎中，还存在于图形用户界面、分布式通信系统等许多其他系统中。著名的“四人帮”设计模式一书[19]将其称为“命令”设计模式。

16.8.3 事件类型

区分不同类型的事件有很多方法。在 C 或 C++ 中，一种简单的方法是定义一个全局枚举，将每种事件类型映射到一个唯一的整数。

```
enum EventType
{
    EVENT_TYPE_LEVEL_STARTED,
    EVENT_TYPE_PLAYER_SPAWNED,
    EVENT_TYPE_ENEMY_SPOTTED,
    EVENT_TYPE_EXPLOSION,
    EVENT_TYPE_BULLET_HIT,
    // ...
}
```

这种方法具有简单和高效的优点（因为整数通常读取、写入和比较速度极快）。然而，它也存在两个问题。首先，整个游戏中所有事件类型的知识都是集中的，这可以看作是一种破坏封装的形式（无论是好是坏——对此的看法各不相同）。其次，事件类型是硬编码的，这意味着新的事件类型无法轻松地以数据驱动的方式定义。第三，枚举器只是索引，因此它们与顺序有关。如果有人不小心在列表中间添加了一个新的事件类型，则所有后续事件 ID 的索引都会发生变化，如果事件 ID 存储在数据文件中，这可能会导致问题。因此，基于枚举的事件类型系统适用于小型演示和原型，但不能很好地扩展到真实游戏。

另一种对事件类型进行编码的方式是通过字符串。这种方法完全自由，只需想出一个名称，就可以将新的事件类型添加到系统中。但它存在许多问题，包括

- *Blind event forwarding*. An object can forward an event that it receives to another object without having to “know” anything about the event. For example, if a vehicle receives a Dismount event, it can forward it to all of its passengers, thereby allowing them to dismount the vehicle, even though the vehicle itself knows nothing about dismounting.

This idea of encapsulating an event/message/command in an object is commonplace in many fields of computer science. It is found not only in game engines but in other systems like graphical user interfaces, distributed communication systems and many others. The well-known “Gang of Four” design patterns book [19] calls this the Command design pattern.

16.8.3 Event Types

There are many ways to distinguish between different types of events. One simple approach in C or C++ is to define a global enum that maps each event type to a unique integer.

```
enum EventType
{
    EVENT_TYPE_LEVEL_STARTED,
    EVENT_TYPE_PLAYER_SPAWNED,
    EVENT_TYPE_ENEMY_SPOTTED,
    EVENT_TYPE_EXPLOSION,
    EVENT_TYPE_BULLET_HIT,
    // ...
}
```

This approach enjoys the benefits of simplicity and efficiency (since integers are usually extremely fast to read, write and compare). However, it also suffers from two problems. First, knowledge of *all* event types in the entire game is centralized, which can be seen as a form of broken encapsulation (for better or for worse—opinions on this vary). Second, the event types are hard-coded, which means new event types cannot easily be defined in a data-driven manner. Third, enumerators are just indices, so they are order-dependent. If someone accidentally adds a new event type in the middle of the list, the indices of all subsequent event ids change, which can cause problems if event ids are stored in data files. As such, an enumeration-based event typing system works well for small demos and prototypes but does not scale very well at all to real games.

Another way to encode event types is via strings. This approach is totally free-form, and it allows a new event type to be added to the system by merely thinking up a name for it. But it suffers from many problems, including a

事件名称冲突的可能性很高，事件可能因为简单的拼写错误而无法正常工作，字符串本身的内存需求增加，并且比较字符串的成本相对于比较整数的成本相对较高。可以使用散列字符串 ID 代替原始字符串来消除性能问题和增加的内存需求，但这并不能解决事件名称冲突或拼写错误。尽管如此，许多游戏团队（包括顽皮狗）认为，基于字符串或字符串 ID 的事件系统具有极高的灵活性和数据驱动的特性，值得冒险。

可以实施一些工具来帮助避免使用字符串识别事件所涉及的一些风险。例如，可以维护一个包含所有事件类型名称的中央数据库。可以提供一个用户界面，允许将新的事件类型添加到数据库中。添加新事件时，可以自动检测命名冲突，并禁止用户添加重复的事件类型。当选择现有事件时，该工具可以在下拉组合框中提供排序列表，而无需用户记住名称并手动输入。事件数据库还可以存储每种事件类型的元数据，包括有关其目的和正确用法的文档，以及有关其支持的参数数量和类型的信息。这种方法效果很好，但我们不应忘记考虑建立和维护这样一个系统的成本，因为它们并非微不足道。

16.8.4 事件参数

事件的参数通常类似于函数的参数列表，提供有关事件的可能对接收者有用的信息。
事件参数可以通过各种方式实现。

我们可能会为每种独特类型的事件派生一种新类型的事件类。
然后可以将参数硬编码为类的数据成员。例如：

```
class ExplosionEvent : public Event
{
    Point  m_center;
    float  m_damage;
    float  m_radius;
};
```

另一种方法是将事件的参数存储为变体的集合。变体是一个能够容纳多种类型数据的数据对象。它通常存储当前正在存储的数据类型以及数据本身的信息。
在事件系统中，我们可能希望我们的参数

strong potential for event name conflicts, the possibility of events not working because of a simple typo, increased memory requirements for the strings themselves, and the relatively high cost of comparing strings relative to that of comparing integers. Hashed string ids can be used instead of raw strings to eliminate the performance problems and increased memory requirements, but they do nothing to address event name conflicts or typos. Nonetheless, the extreme flexibility and data-driven nature of a string- or string-id-based event system is considered worth the risks by many game teams, including Naughty Dog.

Tools can be implemented to help avoid some of the risks involved in using strings to identify events. For example, a central database of all event type names could be maintained. A user interface could be provided to permit new event types to be added to the database. Naming conflicts could be automatically detected when a new event is added, and the user could be disallowed from adding duplicate event types. When selecting a preexisting event, the tool could provide a sorted list in a drop-down combo box rather than requiring the user to remember the name and type it manually. The event database could also store metadata about each type of event, including documentation about its purpose and proper usage and information about the number and types of arguments it supports. This approach can work really well, but we should not forget to account for the costs of setting up and maintaining such a system, as they are not insignificant.

16.8.4 Event Arguments

The arguments of an event usually act like the argument list of a function, providing information about the event that might be useful to the receiver. Event arguments can be implemented in all sorts of ways.

We might derive a new type of `Event` class for each unique type of event. The arguments can then be hard-coded as data members of the class. For example:

```
class ExplosionEvent : public Event
{
    Point  m_center;
    float  m_damage;
    float  m_radius;
};
```

Another approach is to store the event's arguments as a collection of *variants*. A variant is a data object that is capable of holding more than one type of data. It usually stores information about the data type that is currently being stored, as well as the data itself. In an event system, we might want our argu-

变量可以是整数、浮点值、布尔值或散列字符串 ID。因此，在 C 或 C++ 中，我们可以定义一个如下所示的变体类：

```
struct Variant
{
    enum Type
    {
        TYPE_INTEGER,
        TYPE_FLOAT,
        TYPE_BOOL,
        TYPE_STRING_ID,
        TYPE_COUNT // number of unique types
    };

    Type        m_type;

    union
    {
        I32     m_asInteger;
        F32     m_asFloat;
        bool    m_asBool;
        U32     m_asStringId;
    };
};
```

事件中的变量集合可以实现为一个较小的、最大大小固定的数组（例如 4、8 或 16 个元素）。这可以对事件传递的参数数量施加任意限制，但也避免了为每个事件的参数负载动态分配内存的问题，这对于内存受限的主机游戏来说是一个很大的优势。

变体集合可以实现为动态大小的数据结构，例如动态大小的数组（例如 std::vector ）或链表（例如 std::list ）。与固定大小的设计相比，这提供了极大的灵活性，但会产生动态内存分配的开销。假设每个变体的大小相同，池分配器可以在这里发挥巨大作用。

16.8.4.1 事件参数作为键值对

事件参数索引集合的一个根本问题是顺序依赖性。事件的发送者和接收者都必须“知道”参数是按特定顺序排列的。这可能会导致混淆和错误。例如，一个必需的参数可能会被意外遗漏，或者一个额外的参数可能会被添加。

ments to be integers, floating-point values, Booleans or hashed string ids. So in C or C++, we could define a variant class that looks something like this:

```
struct Variant
{
    enum Type
    {
        TYPE_INTEGER,
        TYPE_FLOAT,
        TYPE_BOOL,
        TYPE_STRING_ID,
        TYPE_COUNT // number of unique types
    };

    Type        m_type;

    union
    {
        I32     m_asInteger;
        F32     m_asFloat;
        bool    m_asBool;
        U32     m_asStringId;
    };
};
```

The collection of variants within an `Event` might be implemented as an array with a small, fixed maximum size (say 4, 8 or 16 elements). This imposes an arbitrary limit on the number of arguments that can be passed with an event, but it also side-steps the problems of dynamically allocating memory for each event's argument payload, which can be a big benefit, especially in memory-constrained console games.

The collection of variants might be implemented as a dynamically sized data structure, like a dynamically sized array (like `std::vector`) or a linked list (like `std::list`). This provides a great deal of additional flexibility over a fixed size design, but it incurs the cost of dynamic memory allocation. A pool allocator could be used to great effect here, presuming that each `Variant` is the same size.

16.8.4.1 Event Arguments as Key-Value Pairs

A fundamental problem with an *indexed* collection of event arguments is order dependency. Both the sender and the receiver of an event must "know" that the arguments are listed in a specific order. This can lead to confusion and bugs. For example, a required argument might be accidentally omitted or an extra one added.

可以通过将事件参数实现为键值对来避免此问题。每个参数都由其键唯一标识，因此参数可以按任意顺序出现，并且可选参数可以完全省略。参数集合可以实现为封闭式或开放式哈希表，键用于将哈希值存入表中；也可以实现为键值对的数组、链表或二叉搜索树。表 16.1 说明了这些想法。可能性有很多，只要能够有效且高效地满足游戏的特定需求，具体的实现选择并不重要。

16.8.5 事件处理程序

当游戏对象接收到事件、消息或命令时，它需要以某种方式响应该事件。这被称为事件处理，通常由一个称为事件处理程序的函数或一段脚本代码实现。（稍后我们将进一步学习游戏脚本。）事件处理程序通常是一个能够处理所有类型事件的原生虚函数或脚本函数（例如，virtual void On Event(Event& event)）。在这种情况下，该函数通常包含某种 switch 语句或级联的 if/else-if 子句来处理可能接收到的各种类型的事件。一个典型的事件处理程序函数可能如下所示：

```
virtual void SomeObject::OnEvent(Event& event)
{
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        break;

    case SID("EVENT_HEALTH_PACK"):
```

Key	类型	Value
"event"	stringid	"explosion"
"radius"	float	10.3
"damage"	int	25
"grenade"	bool	true

表 16.1。事件对象的参数可以实现为键值对的集合。键可以避免顺序依赖问题，因为每个事件参数都由其键唯一标识。

This problem can be avoided by implementing event arguments as key-value pairs. Each argument is uniquely identified by its key, so the arguments can appear in any order, and optional arguments can be omitted altogether. The argument collection might be implemented as a closed or open hash table, with the keys used to hash into the table, or it might be an array, linked list or binary search tree of key-value pairs. These ideas are illustrated in Table 16.1. The possibilities are numerous, and the specific choice of implementation is largely unimportant as long as the game's particular requirements have been effectively and efficiently met.

16.8.5 Event Handlers

When an event, message or command is received by a game object, it needs to respond to the event in some way. This is known as *handling* the event, and it is usually implemented by a function or a snippet of script code called an *event handler*. (We'll have more to learn about game scripting later on.)

Often an event handler is a single native virtual function or script function that is capable of handling all types of events (e.g., `virtual void OnEvent(Event& event)`). In this case, the function usually contains some kind of switch statement or cascaded if/else-if clause to handle the various types of events that might be received. A typical event handler function might look something like this:

```
virtual void SomeObject::OnEvent(Event& event)
{
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        break;

    case SID("EVENT_HEALTH_PACK"):
```

Key	Type	Value
"event"	stringid	"explosion"
"radius"	float	10.3
"damage"	int	25
"grenade"	bool	true

Table 16.1. The arguments of an event object can be implemented as a collection of key-value pairs. The keys prevent order-dependency problems because each event argument is uniquely identified by its key.

```
            AddHealth(event.GetHealthPack().GetHealth());
            break;

        // ...

        default:
            // Unrecognized event.
            break;
        }
    }
```

或者，我们可以实现一套处理函数，为每种类型的事件分别设置一个（例如，OnThis()、OnThat() 等等）。然而，正如我们上面所讨论的，事件处理函数的泛滥可能会带来问题。

一个名为微软基础类 (MFC) 的 Windows GUI 工具包以其消息映射而闻名——该系统允许任何 Windows 消息在运行时绑定到任意非虚拟或虚拟函数。这避免了在单个根类中声明所有可能的 Windows 消息处理程序的需要，同时也避免了非 MFC Windows 消息处理函数中常见的冗长的 switch 语句。但这样的系统可能不值得这么麻烦——switch 语句已经足够好用，而且简洁明了。

16.8.6 解析事件的论点

上面的例子掩盖了一个重要的细节，即如何以类型安全的方式从事件的参数列表中提取数据。例如，event.GetHealthPack() 大概会返回一个 HealthPack 游戏对象，而我们假设该对象又提供了一个名为 GetHealth() 的成员函数。这意味着根 Event 类“知道”健康包（以及游戏中所有其他类型的事件参数）。这可能是一个不切实际的设计。在真正的引擎中，可能会有派生的 Event 类，它们提供方便的数据访问 API，例如 GetHealthPack()。或者事件处理程序可能必须手动解包数据并将其转换为适当的类型。后一种方法引发了类型安全问题，尽管实际上它通常不是一个大问题，因为在解包参数时始终可以知道事件的类型。

16.8.7 责任链

游戏对象几乎总是以各种方式相互依赖。例如，游戏对象通常位于一个变换层次结构中，这使得一个对象可以放置在另一个对象上，或者被角色握在手中。游戏对象也可能由多个相互作用的组件组成，


```
            AddHealth(event.GetHealthPack().GetHealth());
            break;

        // ...

        default:
            // Unrecognized event.
            break;
        }
    }
```

Alternatively, we might implement a suite of handler functions, one for each type of event (e.g., `OnThis()`, `OnThat()`, ...). However, as we discussed above, a proliferation of event handler functions can be problematic.

A Windows GUI toolkit called Microsoft Foundation Classes (MFC) was well-known for its *message maps*—a system that permitted any Windows message to be bound at runtime to an arbitrary non-virtual or virtual function. This avoided the need to declare handlers for all possible Windows messages in a single root class, while at the same time avoiding the big switch statement that is commonplace in non-MFC Windows message-handling functions. But such a system is probably not worth the hassle—a switch statement works really well and is simple and clear.

16.8.6 Unpacking an Event's Arguments

The example above glosses over one important detail—namely, how to extract data from the event's argument list in a type-safe manner. For example, `event.GetHealthPack()` presumably returns a `HealthPack` game object, which in turn we presume provides a member function called `GetHealth()`. This implies that the root `Event` class "knows" about health packs (as well as, by extension, every other type of event argument in the game). This is probably an impractical design. In a real engine, there might be derived `Event` classes that provide convenient data-access APIs such as `GetHealthPack()`. Or the event handler might have to unpack the data manually and cast them to the appropriate types. This latter approach raises type safety concerns, although practically speaking it usually isn't a huge problem because the type of the event is always known when the arguments are unpacked.

16.8.7 Chains of Responsibility

Game objects are almost always dependent upon one another in various ways. For example, game objects usually reside in a transformation hierarchy, which allows an object to rest on another object or be held in the hand of a character. Game objects might also be made up of multiple interacting components,

从而形成星型拓扑结构或松散连接的组件对象“云”。体育游戏可能会维护每个队伍所有角色的列表。通常，我们可以将游戏对象之间的相互关系设想为一个或多个关系图（记住，列表和树只是关系图的特例）。图 16.21 显示了一些关系图的示例。

在这些关系图中，能够将事件从一个对象传递到另一个对象通常是有意义的。例如，当一辆车接收到一个事件时，将事件传递给车上的所有乘客可能很方便，而这些乘客可能希望将事件转发给他们库存中的对象。当一个多组件游戏对象接收到一个事件时，可能需要将事件传递给所有组件，以便它们都能处理该事件。或者，当体育游戏中的角色接收到一个事件时，我们可能也希望将其传递给他或她的所有队友。

在对象图中转发事件的技术是面向对象、事件驱动编程中常见的设计模式，有时也称为责任链 [19]。通常，事件在系统中传递的顺序由工程师预先确定。事件被传递给链中的第一个对象，事件处理程序返回一个布尔值或枚举代码，指示它是否识别并处理了该事件。如果事件被接收者消费，则事件转发过程停止；否则，事件将被转发给链中的下一个接收者。支持责任链式事件转发的事件处理程序可能如下所示：

图 16.21。游戏对象以各种方式相互关联，我们可以绘制图表来描述这些关系。任何这样的图表都可以作为事件的分发渠道。

leading to a star topology or a loosely connected “cloud” of component objects. A sports game might maintain a list of all the characters on each team. In general, we can envision the interrelationships between game objects as one or more *relationship graphs* (remembering that a list and a tree are just special cases of a graph). A few examples of relationship graphs are shown in Figure 16.21.

It often makes sense to be able to pass events from one object to the next within these relationship graphs. For example, when a vehicle receives an event, it may be convenient to pass the event to all of the passengers riding on the vehicle, and those passengers may wish to forward the event to the objects in their inventories. When a multicomponent game object receives an event, it may be necessary to pass the event to all of the components so that they all get a crack at handling it. Or when an event is received by a character in a sports game, we might want to pass it on to all of his or her teammates as well.

The technique of forwarding events within a graph of objects is a common design pattern in object-oriented, event-driven programming, sometimes referred to as *Chain of Responsibility* [19]. Usually, the order in which the event is passed around the system is predetermined by the engineers. The event is passed to the first object in the chain, and the event handler returns a Boolean or an enumerated code indicating whether or not it recognized and handled the event. If the event is consumed by a receiver, the process of event forwarding stops; otherwise, the event is forwarded on to the next receiver in the chain. An event handler that supports Chain of Responsibility style event forwarding might look something like this:

Figure 16.21. Game objects are interrelated in various ways, and we can draw graphs depicting these relationships. Any such graph might serve as a distribution channel for events.

```
virtual bool SomeObject::OnEvent(Event& event)
{
    // Call the base class' handler first.
    if (BaseClass::OnEvent(event))
    {
        return true;
    }

    // Now try to handle the event myself.
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        return false; // OK to forward this event to others.

    case SID("EVENT_HEALTH_PACK"):
        AddHealth(event.GetHealthPack().GetHealth());
        return true; // I consumed the event; don't forward.

    // ...

    default:
        return false; // I didn't recognize this event.
    }
}
```

当派生类重写事件处理程序时，如果派生类只是扩充基类的响应而不是替换基类的响应，则调用基类的实现也是合适的。在其他情况下，派生类可能会完全替换基类的响应，在这种情况下不应调用基类的处理程序。这是另一种责任链。

事件转发还有其他应用。例如，我们可能希望将事件多播到影响半径内的所有对象（例如爆炸）。为了实现这一点，我们可以利用游戏世界的对象查询机制来查找相关范围内的所有对象，然后将事件转发给所有返回的对象。

16.8.8 注册对活动的兴趣

可以肯定地说，游戏中的大多数对象不需要响应所有可能的事件。大多数类型的游戏对象只对一小部分事件感兴趣。这会导致多播或广播事件效率低下，因为我们需要遍历一组事件。


```
virtual bool SomeObject::OnEvent(Event& event)
{
    // Call the base class' handler first.
    if (BaseClass::OnEvent(event))
    {
        return true;
    }

    // Now try to handle the event myself.
    switch (event.GetType())
    {
    case SID("EVENT_ATTACK"):
        RespondToAttack(event.GetAttackInfo());
        return false; // OK to forward this event to others.

    case SID("EVENT_HEALTH_PACK"):
        AddHealth(event.GetHealthPack().GetHealth());
        return true; // I consumed the event; don't forward.

    // ...

    default:
        return false; // I didn't recognize this event.
    }
}
```

When a derived class overrides an event handler, it can be appropriate to call the base class's implementation as well if the class is augmenting but not replacing the base class's response. In other situations, the derived class might be entirely replacing the response of the base class, in which case the base class's handler should not be called. This is another kind of responsibility chain.

Event forwarding has other applications as well. For example, we might want to multicast an event to all objects within a radius of influence (for an explosion, for example). To implement this, we can leverage our game world's object query mechanism to find all objects within the relevant sphere and then forward the event to all of the returned objects.

16.8.8 Registering Interest in Events

It's reasonably safe to say that most objects in a game do not need to respond to every possible event. Most types of game objects have a relatively small set of events in which they are "interested." This can lead to inefficiencies when multicasting or broadcasting events, because we need to iterate over a group

对象并调用每个对象的事件处理程序，即使该对象对该特定类型的事件不感兴趣。

克服这种低效率的一种方法是允许游戏对象注册对特定类型事件的兴趣。例如，我们可以为每种不同类型的事件维护一个感兴趣的游戏对象链表，或者每个游戏对象可以维护一个位数组，其中每个位的设置对应于该对象是否对特定类型的事件感兴趣。通过这样做，我们可以避免调用任何不关心该事件的对象的事件处理程序。

更好的是，我们或许可以限制原始游戏对象查询，使其仅包含对我们希望多播的事件感兴趣的对象。例如，当爆炸发生时，我们可以向碰撞系统请求所有位于伤害半径内且能够响应爆炸事件的对象。这可以节省整体时间，因为我们避免了迭代那些我们知道对多播事件不感兴趣的对象。这种方法是否会产生净收益取决于查询机制的实现方式，以及在查询过程中过滤对象与在多播迭代过程中过滤对象的相对成本。

16.8.9 排队还是不排队

大多数游戏引擎都提供了一种机制，可以在事件发送后立即处理。除此之外，一些引擎还允许将事件排队，以便在未来的任意时间进行处理。事件排队有一些吸引人的优点，但它也显著增加了事件系统的复杂性，并带来了一些独特的问题。我们将在接下来的章节中探讨事件排队的优缺点，并了解此类系统是如何实现的。

16.8.9.1 事件队列的一些好处

以下部分概述了事件排队的一些好处。

控制何时处理事件

我们已经看到，我们必须谨慎地按照特定顺序更新引擎子系统和游戏对象，以确保正确的行为并最大限度地提高运行时性能。同样，某些类型的事件可能对它们在游戏循环中的处理时间高度敏感。如果所有事件在发送后都立即处理，那么在整个游戏循环过程中，事件处理函数最终会以不可预测且难以控制的方式被调用。通过事件队列延迟事件，引擎

of objects and call each one's event handler, even if the object is not interested in that particular kind of event.

One way to overcome this inefficiency is to permit game objects to *register interest* in particular kinds of events. For example, we could maintain one linked list of interested game objects for each distinct type of event, or each game object could maintain a bit array, in which the setting of each bit corresponds to whether or not the object is interested in a particular type of event. By doing this, we can avoid calling the event handlers of any objects that do not care about the event.

Even better, we might be able to restrict our original game object query to include only those objects that are interested in the event we wish to multicast. For example, when an explosion goes off, we can ask the collision system for all objects that are within the damage radius *and* that can respond to Explosion events. This can save time overall, because we avoid iterating over objects that we know aren't interested in the event we're multicasting. Whether or not such an approach will produce a net gain depends on how the query mechanism is implemented and the relative costs of filtering the objects during the query versus filtering them during the multicast iteration.

16.8.9 To Queue or Not to Queue

Most game engines provide a mechanism for handling events immediately when they are sent. In addition to this, some engines also permit events to be queued for handling at an arbitrary future time. Event queuing has some attractive benefits, but it also increases the complexity of the event system significantly and poses some unique problems. We'll investigate the pros and cons of event queuing in the following sections and learn how such systems are implemented in the process.

16.8.9.1 Some Benefits of Event Queuing

The following sections outline some of the benefits of event queuing.

Control over When Events Are Handled

We have seen that we must be careful to update engine subsystems and game objects in a specific order to ensure correct behavior and maximize runtime performance. In the same sense, certain kinds of events may be highly sensitive to exactly when within the game loop they are handled. If all events are handled immediately upon being sent, the event handler functions end up being called in unpredictable and difficult-to-control ways throughout the course of the game loop. By deferring events via an event queue, the engi-

人员可以采取措施确保仅在安全和适当的情况下处理事件。

能够将事件发布到未来

发送事件时，发送者通常可以指定一个传递时间——例如，我们可能希望事件在同一帧、下一帧或发送后的几秒内得到处理。此功能相当于将事件发布到未来的能力，它有各种有趣的用途。我们可以通过将事件发布到未来来实现一个简单的闹钟。周期性任务（例如每两秒闪烁一次灯光）可以通过发布一个事件来执行，该事件的处理程序执行该周期性任务，然后在未来一个时间段内发布一个同类型的新事件。

为了实现将事件提交到未来的功能，每个事件在加入队列之前都会被标记一个期望的提交时间。只有当当前游戏时钟匹配或超过其提交时间时，才会处理该事件。实现此操作的一个简单方法是按提交时间的递增顺序对队列中的事件进行排序。每帧，都可以检查队列中的第一个事件并检查其提交时间。如果提交时间是未来的，我们会立即中止，因为我们知道所有后续事件也都是未来的。但是，如果我们发现某个事件的提交时间是现在或过去，我们会将其从队列中提取并处理。此过程持续进行，直到找到提交时间是未来的事件。以下伪代码说明了此过程：

```
// This function is called at least once per frame. Its
// job is to dispatch all events whose delivery time is
// now or in the past.

void EventQueue::DispatchEvents(F32 currentTime)
{
    // Look at, but don't remove, the next event on the
    // queue.
    Event* pEvent = PeekNextEvent();

    while (pEvent
        && pEvent->GetDeliveryTime() <= currentTime)
    {
        // Remove the event from the queue.
        RemoveNextEvent();

        // Dispatch it to its receiver's event handler.
        pEvent->Dispatch();

        // Peek at the next event on the queue (again
```


neers can take steps to ensure that events are only handled when it is safe and appropriate to do so.

Ability to Post Events into the Future

When an event is sent, the sender can usually specify a delivery time—for example, we might want the event to be handled later in the same frame, next frame or some number of seconds after it was sent. This feature amounts to an ability to post events into the future, and it has all sorts of interesting uses. We can implement a simple alarm clock by posting an event into the future. A periodic task, such as blinking a light every two seconds, can be executed by posting an event whose handler performs the periodic task and then posts a new event of the same type one time period into the future.

To implement the ability to post events into the future, each event is stamped with a desired delivery time prior to being queued. An event is only handled when the current game clock matches or exceeds its delivery time. An easy way to make this work is to sort the events in the queue in order of increasing delivery time. Each frame, the first event on the queue can be inspected and its delivery time checked. If the delivery time is in the future, we abort immediately because we know that all subsequent events are also in the future. But if we see an event whose delivery time is now or in the past, we extract it from the queue and handle it. This continues until an event is found whose delivery time is in the future. The following pseudocode illustrates this process:

```
// This function is called at least once per frame. Its
// job is to dispatch all events whose delivery time is
// now or in the past.

void EventQueue::DispatchEvents(F32 currentTime)
{
    // Look at, but don't remove, the next event on the
    // queue.
    Event* pEvent = PeekNextEvent();

    while (pEvent
        && pEvent->GetDeliveryTime() <= currentTime)
    {
        // Remove the event from the queue.
        RemoveNextEvent();

        // Dispatch it to its receiver's event handler.
        pEvent->Dispatch();

        // Peek at the next event on the queue (again
```

```
            // without removing it).
            pEvent = PeekNextEvent();
        }
    }
```

事件优先级

即使我们的事件在事件队列中按交付时间排序，当两个或多个事件的交付时间完全相同时，交付顺序仍然会变得模糊。这种情况发生的频率可能比你想象的要高，因为事件的交付时间通常被量化为整数帧。例如，如果两个发送者请求在“此帧”、“下一帧”或“七帧后”调度事件，那么这些事件的交付时间将完全相同。

解决这些歧义的一种方法是为事件分配优先级。当两个事件具有相同的时间戳时，应始终优先处理优先级较高的事件。这很容易实现：首先按传递时间的升序对事件队列进行排序，然后按优先级降序对每组传递时间相同的事件进行排序。

我们可以通过将优先级编码为原始的无符号 32 位整数，来允许最多 40 亿个唯一优先级，或者我们可以将优先级限制为仅两到三个（例如，低、中、高）。每个游戏引擎都存在一个最小优先级数，可以解决系统中所有实际的模糊性问题。通常情况下，最好尽可能接近这个最小值。如果优先级数量非常多，那么在任何特定情况下确定哪个事件应该首先被处理可能会变成一场小噩梦。然而，每个游戏事件系统的需求各不相同，您的情况也可能有所不同。

16.8.9.2 事件队列的一些问题

事件系统复杂性增加

为了实现队列事件系统，我们需要比实现即时事件系统更多的代码、额外的数据结构和更复杂的算法。复杂性的增加通常意味着更长的开发时间，以及在游戏开发过程中维护和改进系统的更高成本。

深度复制事件及其参数

使用即时事件处理方法，事件参数中的数据只需在事件处理函数的持续时间内保留（以及任何


```
            // without removing it).
            pEvent = PeekNextEvent();
        }
    }
```

Event Prioritization

Even if our events are sorted by delivery time in the event queue, the order of delivery is still ambiguous when two or more events have exactly the same delivery time. This can happen more often than you might think, because it is quite common for events' delivery times to be quantized to an integral number of frames. For example, if two senders request that events be dispatched "this frame," "next frame" or "in seven frames from now," then those events will have identical delivery times.

One way to resolve these ambiguities is to assign *priorities* to events. Whenever two events have the same time stamp, the one with higher priority should always be serviced first. This is easily accomplished by first sorting the event queue by increasing delivery times and then sorting each group of events with identical delivery times in order of decreasing priority.

We could allow up to four billion unique priority levels by encoding our priorities in a raw, unsigned 32-bit integer, or we could limit ourselves to only two or three unique priority levels (e.g., low, medium and high). In every game engine, there exists some minimum number of priority levels that will resolve all real ambiguities in the system. It's usually best to aim as close to this minimum as possible. With a very large number of priority levels, it can become a small nightmare to figure out which event will be handled first in any given situation. However, the needs of every game's event system are different, and your mileage may vary.

16.8.9.2 Some Problems with Event Queuing

Increased Event System Complexity

In order to implement a queued event system, we need more code, additional data structures and more complex algorithms than would be necessary to implement an immediate event system. Increased complexity usually translates into longer development times and a higher cost to maintain and evolve the system during development of the game.

Deep-Copying Events and Their Arguments

With an immediate event handling approach, the data in an event's arguments need only persist for the duration of the event handling function (and any

它可能调用的函数）。这意味着事件及其参数数据可以驻留在内存中的任何位置，包括调用堆栈中。例如，我们可以编写一个如下所示的函数：

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // Allocate event args on the call stack.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;

    // Allocate the event on the call stack.
    Event event("Explosion");
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // Send the event, which causes the receiver's event
    // handler to be called immediately, as shown below.
    event.Send(receiver);
    //{
    //    receiver.OnEvent(event);
    //}
}
```

当事件排队时，其参数必须持续存在于发送函数的作用域之外。这意味着我们必须在将事件存储到队列之前复制整个事件对象。我们必须执行深度复制，这意味着我们不仅要复制事件对象本身，还要复制其整个参数负载，包括它可能指向的任何数据。深度复制事件可确保不存在仅存在于发送函数作用域中的数据的悬垂引用，并且允许无限期地存储事件。上面显示的示例事件发送函数在使用排队事件系统时仍然看起来基本相同，但正如您在下面的斜体代码中看到的，Event::Queue() 函数的实现比其对应的 Send() 函数稍微复杂一些：

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // We can still allocate event args on the call
    // stack.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;
```


functions it may call). This means that the event and its argument data can reside literally anywhere in memory, including on the call stack. For example, we could write a function that looks something like this:

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // Allocate event args on the call stack.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;

    // Allocate the event on the call stack.
    Event event("Explosion");
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // Send the event, which causes the receiver's event
    // handler to be called immediately, as shown below.
    event.Send(receiver);
    //{
    //    receiver.OnEvent(event);
    //}
}
```

When an event is queued, its arguments must persist beyond the scope of the sending function. This implies that we must copy the entire event object prior to storing the event in the queue. We must perform a *deep-copy*, meaning that we copy not only the event object itself but its entire argument payload as well, including any data to which it may be pointing. Deep-copying the event ensures that there are no dangling references to data that exist only in the sending function's scope, and it permits the event to be stored indefinitely. The example event-sending function shown above still looks basically the same when using a queued event system, but as you can see in the italicized code below, the implementation of the `Event::Queue()` function is a bit more complex than its `Send()` counterpart:

```
void SendExplosionEventToObject(GameObject& receiver)
{
    // We can still allocate event args on the call
    // stack.
    Point centerPoint(-2.0f, 31.5f, 10.0f);
    F32   damage = 5.0f;
    F32   radius = 2.0f;
```

```
        // Still OK to allocate the event on the call stack
        Event event("Explosion
        event.SetArgFloat("Damage", damage);
        event.SetArgPoint("Center", &centerPoint);
        event.SetArgFloat("Radius", radius);

        // This stores the event in the receiver's queue for
        // handling at a future time. Note how the event
        // must be deep-copied prior to being enqueued, since
        // the original event resides on the call stack and
        // will go out of scope when this function returns.
        event.Queue(receiver);
        //{
        //    Event* pEventCopy = DeepCopy(event);
        //    receiver.EnqueueEvent(pEventCopy);
        //}
    }
```

排队事件的动态内存分配

事件对象的深度复制意味着需要动态内存分配，正如我们多次提到的，动态分配在游戏引擎中并不可取，因为它的潜在成本以及容易造成内存碎片。尽管如此，如果我们想要将事件排队，就需要为它们动态分配内存。

与游戏引擎中的所有动态分配一样，最好选择一个快速且无碎片的分配器。我们也许能够使用池分配器，但这仅当我们所有的事件对象大小相同并且它们的参数列表由大小相同的数据元素组成时才有效。情况很可能就是这样 - 例如，每个参数可能都是 Variant ，如上所述。如果我们的事件对象和/或它们的参数大小可以变化，则可能适用小型内存分配器。（回想一下，小型内存分配器维护多个池，每个池用于几个预定的小分配大小。）在设计排队事件系统时，务必注意考虑动态分配要求。

当然，其他设计也是可行的。例如，在顽皮狗，我们将排队事件分配为可重定位内存块。有关可重定位内存的更多信息，请参阅第 6.2.2.2 节。

调试困难

对于排队事件，事件处理程序并非由事件的发送者直接调用。因此，与即时事件处理不同，调用堆栈不会告诉我们事件的来源。我们无法在调试器中遍历调用堆栈来检查发送者的状态或事件发送的具体情况。这会使延迟事件的调试变得有些棘手，并且


```
        // Still OK to allocate the event on the call stack
        Event event("Explosion
        event.SetArgFloat("Damage", damage);
        event.SetArgPoint("Center", &centerPoint);
        event.SetArgFloat("Radius", radius);

        // This stores the event in the receiver's queue for
        // handling at a future time. Note how the event
        // must be deep-copied prior to being enqueued, since
        // the original event resides on the call stack and
        // will go out of scope when this function returns.
        event.Queue(receiver);
        //{
        //    Event* pEventCopy = DeepCopy(event);
        //    receiver.EnqueueEvent(pEventCopy);
        //}
    }
```

Dynamic Memory Allocation for Queued Events

Deep-copying of event objects implies a need for dynamic memory allocation, and as we've already noted many times, dynamic allocation is undesirable in a game engine due to its potential cost and its tendency to fragment memory. Nonetheless, if we want to queue events, we'll need to dynamically allocate memory for them.

As with all dynamic allocation in a game engine, it's best if we can select a fast and fragmentation-free allocator. We might be able to use a *pool allocator*, but this will only work if all of our event objects are the same size and if their argument lists are comprised of data elements that are themselves all the same size. This may well be the case—for example, the arguments might each be a `Variant`, as described above. If our event objects and/or their arguments can vary in size, a *small memory allocator* might be applicable. (Recall that a small memory allocator maintains multiple pools, one for each of a few pre-determined small allocation sizes.) When designing a queued event system, always be careful to take dynamic allocation requirements into account.

Other designs are possible, of course. For example, at Naughty Dog we allocate queued events as relocatable memory blocks. See Section 6.2.2.2 for more information on relocatable memory.

Debugging Difficulties

With queued events, the event handler is not called directly by the sender of that event. So, unlike in immediate event handling, the call stack does not tell us where the event came from. We cannot walk up the call stack in the debugger to inspect the state of the sender or the circumstances under which the event was sent. This can make debugging deferred events a bit tricky, and

当事件从一个对象转发到另一个对象时，事情变得更加困难。

一些引擎存储调试信息，形成事件在整个系统中传播的纸质记录，但无论如何切分，在没有排队的情况下事件调试通常会容易得多。

事件排队也会导致一些有趣且难以追踪的竞争条件错误。我们可能需要在整个游戏循环中多次触发事件调度，以确保事件的传递不会造成不必要的一帧延迟，同时仍能确保游戏对象在帧内以正确的顺序更新。例如，在动画更新期间，我们可能会检测到某个动画已运行完成。这可能会导致发送一个事件，其处理程序想要播放新的动画。显然，我们希望避免第一个动画结束和下一个动画开始之间出现一帧延迟。为了实现这一点，我们需要首先更新动画时钟（以便能够检测到动画的结束并发送事件）；然后我们应该调度事件（以便事件处理程序有机会请求新的动画），最后我们可以开始动画混合（以便处理和显示新动画的第一帧）。以下代码片段演示了这一点：

```
while (true) // main game loop
{
    // ...

    // Update animation clocks. This may detect the end
    // of a clip, and cause EndOfAnimation events to
    // be sent.
    g_animationEngine.UpdateLocalClocks(dt);

    // Next, dispatch events. This allows an
    // EndOfAnimation event handler to start up a new
    // animation this frame if desired.
    g_eventSystem.DispatchEvents();

    // Finally, start blending all currently playing
    // animations (including any new clips started
    // earlier this frame).
    g_animationEngine.StartAnimationBlending();

    // ...
}
```


things get even more difficult when events are forwarded from one object to another.

Some engines store debugging information that forms a paper trail of the event's travels throughout the system, but no matter how you slice it, event debugging is usually much easier in the absence of queuing.

Event queuing also leads to interesting and hard-to-track-down *race condition* bugs. We may need to pepper multiple event dispatches throughout our game loop, to ensure that events are delivered without incurring unwanted one-frame delays yet still ensuring that game objects are updated in the proper order during the frame. For example, during the animation update, we might detect that a particular animation has run to completion. This might cause an event to be sent whose handler wants to play a new animation. Clearly, we want to avoid a one-frame delay between the end of the first animation and the start of the next. To make this work, we need to update animation clocks first (so that the end of the animation can be detected and the event sent); then we should dispatch events (so that the event handler has a chance to request a new animation), and finally we can start animation blending (so that the first frame of the new animation can be processed and displayed). This is illustrated in the code snippet below:

```
while (true) // main game loop
{
    // ...

    // Update animation clocks. This may detect the end
    // of a clip, and cause EndOfAnimation events to
    // be sent.
    g_animationEngine.UpdateLocalClocks(dt);

    // Next, dispatch events. This allows an
    // EndOfAnimation event handler to start up a new
    // animation this frame if desired.
    g_eventSystem.DispatchEvents();

    // Finally, start blending all currently playing
    // animations (including any new clips started
    // earlier this frame).
    g_animationEngine.StartAnimationBlending();

    // ...
}
```

16.8.10 立即事件发送的一些问题

不对事件进行排队也存在一些问题。例如，立即事件处理可能会导致极深的调用堆栈。对象 A 可能向对象 B 发送一个事件，而在其事件处理程序中，B 可能发送另一个事件，而该事件又可能发送另一个事件，如此反复。在支持立即事件处理的游戏引擎中，经常会看到类似这样的调用堆栈：

```
...
ShoulderAngel::OnEvent()
Event::Send()
Characer::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
HandleSoundEffect()
AnimationEngine::PlayAnimation()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::Update()
GameWorld::UpdateObjectsInBucket()
Engine::GameLoop()
main()
```

像这样的深层调用栈在极端情况下可能会耗尽可用的堆栈空间（尤其是在事件发送无限循环的情况下），但问题的关键在于，每个事件处理函数都必须编写为完全可重入的。这意味着事件处理函数可以递归调用而不会产生任何不良副作用。举个例子，假设一个函数会增加全局变量的值。如果全局变量每帧只增加一次，那么这个函数就不是可重入的，因为多次递归调用该函数会多次增加该变量的值。

16.8.10 Some Problems with Immediate Event Sending

Not queuing events also has its share of issues. For example, immediate event handling can lead to extremely deep call stacks. Object A might send object B an event, and in its event handler, B might send another event, which might send another event, and another and so on. In a game engine that supports immediate event handling, it's not uncommon to see a call stack that looks something like this:

```
...
ShoulderAngel::OnEvent()
Event::Send()
Characer::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
HandleSoundEffect()
AnimationEngine::PlayAnimation()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::Update()
GameWorld::UpdateObjectsInBucket()
Engine::GameLoop()
main()
```

A deep call stack like this can exhaust available stack space in extreme cases (especially if we have an infinite loop of event sending), but the real crux of the problem here is that every event handler function must be written to be fully *re-entrant*. This means that the event handler can be called recursively without any ill side-effects. As a contrived example, imagine a function that increments the value of a global variable. If the global is supposed to be incremented only once per frame, then this function is *not* re-entrant, because multiple recursive calls to the function will increment the variable multiple times.

16.8.11 数据驱动的事件/消息传递系统

事件系统为游戏程序员提供了极大的灵活性，这远超 C 和 C++ 等语言提供的静态类型函数调用机制所能实现的功能。然而，我们可以做得更好。在迄今为止的讨论中，事件发送和接收的逻辑仍然是硬编码的，因此完全由工程师控制。如果我们能够将事件系统设计成数据驱动的，我们就能将其功能扩展到游戏设计师手中。

实现事件系统数据驱动的方法有很多。从极端的完全硬编码（非数据驱动）事件系统开始，我们可以设想提供一些简单的数据驱动可配置性。例如，设计师可以配置单个对象或整个对象类如何响应某些事件。在世界编辑器中，我们可以设想选择一个对象，然后弹出一个滚动列表，其中包含它可能接收的所有事件。对于每个事件，设计师可以使用下拉组合框和复选框来控制对象是否响应以及如何响应，方法是从一组硬编码的预定义选项中进行选择。例如，给定事件“PlayerSpotted”，AI控制的角色可以配置为执行以下操作之一：逃跑、攻击或完全忽略该事件。一些实际的商业游戏引擎的事件系统基本上就是以这种方式实现的。

另一方面，我们的引擎可能会为游戏设计师提供一种简单的脚本语言（我们将在16.9节详细探讨）。在这种情况下，设计师可以编写代码来定义特定类型的游戏对象如何响应特定类型的事件。在脚本模型中，设计师实际上只是程序员（使用一种功能稍弱但更易于使用且希望比工程师更不容易出错的语言），因此一切皆有可能。设计师可以定义新的事件类型，发送事件，并以任意方式接收和处理事件。这就是我们在顽皮狗所做的。

简单、可配置的事件系统的问题在于，它会严重限制游戏设计师在没有程序员帮助的情况下独立完成任务的能力。另一方面，完全脚本化的解决方案也存在一些问题：许多游戏设计师并非受过专业软件工程师的培训，因此他们发现学习和使用脚本语言是一项艰巨的任务。此外，除非设计师在脚本或编程方面有丰富的实践经验，否则他们可能比工程师更容易在游戏中引入bug。这可能会在 Alpha 测试期间导致一些令人不快的意外。

16.8.11 Data-Driven Event/Message-Passing Systems

Event systems give the game programmer a great deal of flexibility over and above what can be accomplished with the statically typed function calling mechanisms provided by languages like C and C++. However, we can do better. In our discussions thus far, the logic for sending and receiving events is still hard-coded and therefore under the exclusive control of the engineers. If we could make our event system data-driven, we could extend its power into the hands of our game designers.

There are many ways to make an event system data-driven. Starting with the extreme of an entirely hard-coded (non-data-driven) event system, we could imagine providing some simple data-driven configurability. For example, designers might be allowed to configure how individual objects, or entire classes of object, respond to certain events. In the world editor, we can imagine selecting an object and then bringing up a scrolling list of all possible events that it might receive. For each one, the designer could use drop-down combo boxes and check boxes to control if, and how, the object responds, by selecting from a set of hard-coded, predefined choices. For example, given the event “PlayerSpotted,” AI-controlled characters might be configured to do one of the following actions: run away, attack or ignore the event altogether. The event systems of some real commercial game engines are implemented in essentially this way.

At the other end of the gamut, our engine might provide the game designers with a simple scripting language (a topic we’ll explore in detail in Section 16.9). In this case, the designer can literally write code that defines how a particular kind of game object will respond to a particular kind of event. In a scripted model, the designers are really just programmers (working with a somewhat less powerful but also easier-to-use and hopefully less error-prone language than the engineers), so anything is possible. Designers might define new types of events, send events and receive and handle events in arbitrary ways. This is what we do at Naughty Dog.

The problem with a simple, configurable event system is that it can severely limit what the game designers are capable of doing on their own, without the help of a programmer. On the other hand, a fully scripted solution has its own share of problems: Many game designers are not professional software engineers by training, so some designers find learning and using a scripting language a daunting task. Designers are also probably more prone to introducing bugs into the game than their engineer counterparts, unless they have practiced scripting or programming for some time. This can lead to some nasty surprises during alpha.

因此，一些游戏引擎力求找到一个折中方案。它们采用复杂的图形用户界面，提供极大的灵活性，但又不至于为用户提供功能齐全、形式自由的脚本语言。一种方法是提供流程图式的图形编程语言。这种系统背后的理念是为用户提供一组有限且可控的原子操作供其选择，同时又赋予用户足够的自由度，让他们能够以任意方式连接这些操作。例如，为了响应“PlayerSpotted”之类的事件，设计师可以连接一个流程图，使角色撤退到最近的掩体点，播放动画，等待5秒，然后发起攻击。GUI还可以提供错误检查和验证，以帮助确保不会无意中引入错误。虚幻引擎的蓝图就是这种系统的一个例子——更多详情请参阅下一节。

16.8.11.1 数据通路通信系统

将类似函数调用的事件系统转换为数据驱动系统的问题之一是，不同类型的事件往往互不兼容。例如，假设在一款游戏中，玩家拥有一把电磁脉冲枪。这种脉冲会关闭灯光和电子设备，吓跑小动物，并产生冲击波，使附近的植物摇晃。这些游戏对象类型中的每一种可能都已经有一个执行所需行为的事件响应。小动物可能会通过“惊吓”事件来响应，从而逃走。电子设备可能会通过关闭自身来响应“关闭”事件。植物可能有一个用于“风”事件的事件处理程序，该事件会导致它们摇晃。问题在于，我们的电磁脉冲枪与这些对象的任何事件处理程序都不兼容。因此，我们最终不得不实现一种新的事件类型，可能名为“EMP”，然后为每种类型的游戏对象编写自定义事件处理程序来响应它。

解决这个问题的一个方法是将事件类型从等式中剔除，只考虑将数据流从一个游戏对象发送到另一个游戏对象。在这样的系统中，每个游戏对象都有一个或多个输入端口，可以连接数据流；还有一个或多个输出端口，可以通过这些端口将数据发送到其他对象。如果我们有某种方式将这些端口连接在一起，例如一个图形用户界面，其中的端口可以通过橡皮筋线相互连接，那么我们就可以构建任意复杂的行为。继续我们的例子，电磁脉冲枪会有一个输出端口，可能名为“Fire”，用于发送布尔信号。大多数情况下，该端口会输出值 0（假），但当枪发射时，它会发送一个短暂的（一帧）脉冲，值为 1（真）。其他游戏对象

As a result, some game engines aim for a middle ground. They employ sophisticated graphical user interfaces to provide a great deal of flexibility without going so far as to provide users with a full-fledged, free-form scripting language. One approach is to provide a flowchart-style graphical programming language. The idea behind such a system is to provide the user with a limited and controlled set of atomic operations from which to choose but with plenty of freedom to wire them up in arbitrary ways. For example, in response to an event like “PlayerSpotted,” the designer could wire up a flowchart that causes a character to retreat to the nearest cover point, play an animation, wait 5 seconds, and then attack. A GUI can also provide error-checking and validation to help ensure that bugs aren’t inadvertently introduced. Unreal’s Blueprints is an example of such a system—see the following section for more details.

16.8.11.1 Data Pathway Communication Systems

One of the problems with converting a function-call-like event system into a data-driven system is that different types of events tend to be incompatible. For example, let’s imagine a game in which the player has an electromagnetic pulse gun. This pulse causes lights and electronic devices to turn off, scares small animals and produces a shock wave that causes any nearby plants to sway. Each of these game object types may already have an event response that performs the desired behavior. A small animal might respond to the “Scare” event by scurrying away. An electronic device might respond to the “TurnOff” event by turning itself off. And plants might have an event handler for a “Wind” event that causes them to sway. The problem is that our EMP gun is not compatible with any of these objects’ event handlers. As a result, we end up having to implement a new event type, perhaps called “EMP,” and then write custom event handlers for every type of game object in order to respond to it.

One solution to this problem is to take the event type out of the equation and to think solely in terms of *sending streams of data* from one game object to another. In such a system, every game object has one or more *input ports* to which a data stream can be connected, and one or more output ports through which data can be sent to other objects. Provided we have some way of wiring these ports together, such as a graphical user interface in which ports can be connected to each other via rubber-band lines, then we can construct arbitrarily complex behaviors. Continuing our example, the EMP gun would have an output port, perhaps named “Fire,” that sends a Boolean signal. Most of the time, the port produces the value 0 (false), but when the gun is fired, it sends a brief (one-frame) pulse of the value 1 (true). The other game objects in

图 16.22。电磁脉冲枪发射时，其“Fire”输出端输出 1。该输出端可以连接到任何需要布尔值的输入端口，以触发与该输入相关的行为。

现实世界中存在二进制输入端口，可以触发各种响应。动物可能有一个“惊吓”输入，电子设备有一个“开启”输入，树叶对象有一个“摇摆”输入。如果我们将电磁脉冲枪的“开火”输出端口连接到这些游戏对象的输入端口，就能让枪触发所需的行为。（请注意，在将枪的“开火”输出连接到电子设备的“开启”输入之前，我们必须先将其输入反转的节点连接到管道。这是因为我们希望它们在枪开火时关闭。）本例的接线图如图 16.22 所示。

程序员决定每种游戏对象应具有哪些类型的端口。使用 GUI 的设计人员可以任意方式连接这些端口，从而在游戏中构建任意行为。程序员还提供各种其他类型的节点供图形中使用，例如反转输入的节点、产生正弦波的节点或以秒为单位输出当前游戏时间的节点。

各种类型的数据可能沿着数据路径发送。一些端口可能生成或期望布尔数据，而其他端口可能被编码为生成或期望单位浮点数形式的数据。还有一些端口可能操作三维矢量、颜色、整数等等。在这样的系统中，重要的是确保仅在数据类型兼容的端口之间建立连接，或者我们必须提供某种机制，以便在两个不同类型的端口连接在一起时自动转换数据类型。例如，将单位浮点数输出连接到布尔输入可能会自动导致任何小于 0.5 的值转换为 false，而任何大于或等于 0.5 的值转换为 true。这就是基于 GUI 的事件系统（例如虚幻引擎 4 的蓝图）的精髓。蓝图的屏幕截图如图 16.23 所示。

Figure 16.22. The EMP gun produces a 1 at its “Fire” output when fired. This can be connected to any input port that expects a Boolean value, in order to trigger the behavior associated with that input.

the world have binary input ports that trigger various responses. The animals might have a “Scare” input, the electronic devices a “TurnOn” input and the foliage objects a “Sway” input. If we connect the EMP gun’s “Fire” output port to the input ports of these game objects, we can cause the gun to trigger the desired behaviors. (Note that we’d have to pipe the gun’s “Fire” output through a node that *inverts* its input, prior to connecting it to the “TurnOn” input of the electronic devices. This is because we want them to turn *off* when the gun is firing.) The wiring diagram for this example is shown in Figure 16.22.

Programmers decide what kinds of port(s) each type of game object will have. Designers using the GUI can then wire these ports together in arbitrary ways in order to construct arbitrary behaviors in the game. The programmers also provide various other kinds of nodes for use within the graph, such as a node that inverts its input, a node that produces a sine wave or a node that outputs the current game time in seconds.

Various types of data might be sent along a data pathway. Some ports might produce or expect Boolean data, while others might be coded to produce or expect data in the form of a unit float. Still others might operate on 3D vectors, colors, integers and so on. It’s important in such a system to ensure that connections are only made between ports with compatible data types, or we must provide some mechanism for automatically converting data types when two differently typed ports are connected together. For example, connecting a unit-float output to a Boolean input might automatically cause any value less than 0.5 to be converted to false, and any value greater than or equal to 0.5 to be converted to true. This is the essence of GUI-based event systems like Unreal Engine 4’s Blueprints. A screenshot of Blueprints is shown in Figure 16.23.

图 16.23.虚幻引擎 4 的蓝图。

16.8.11.2 基于 GUI 编程的一些优点和缺点

图形用户界面相对于基于文本文件的简单脚本语言的优势显而易见：易于使用、循序渐进的学习曲线、内置帮助和工具提示，以及丰富的错误检查功能。流程图式 GUI 的缺点包括开发、调试和维护成本高昂，额外的复杂性（可能导致恼人甚至拖延进度的错误），以及设计师有时使用该工具的功能有限。基于文本文件的编程语言相比基于 GUI 的编程系统具有一些独特的优势，包括相对简单（意味着更不容易出现错误）、能够在源代码中轻松搜索和替换，以及每个用户可以自由选择自己最熟悉的文本编辑器。

16.9 脚本

脚本语言可以定义为一种编程语言，其主要目的是允许用户控制和自定义软件应用程序的行为。例如，Visual Basic 语言可以用来自定义 Microsoft Excel 的行为；MEL 语言和 Python 语言都可以用来自定义 Maya 的行为。在游戏引擎领域，

Figure 16.23. Unreal Engine 4's Blueprints.

16.8.11.2 Some Pros and Cons of GUI-Based Programming

The benefits of a graphical user interface over a straightforward, text-file-based scripting language are probably pretty obvious: ease of use, a gradual learning curve with the potential for in-tool help and tool tips to guide the user, and plenty of error-checking. The downsides of a flowchart style GUI include the high cost to develop, debug, and maintain such a system, the additional complexity, which can lead to annoying or sometimes schedule-killing bugs, and the fact that designers are sometimes limited in what they can do with the tool. A text-file-based programming language has some distinct advantages over a GUI-based programming system, including its relative simplicity (meaning that it is much less prone to bugs), the ability to easily search and replace within the source code, and the freedom of each user to choose the text editor with which they are most comfortable.

16.9 Scripting

A *scripting language* can be defined as a programming language whose primary purpose is to permit users to control and customize the behavior of a software application. For example, the Visual Basic language can be used to customize the behavior of Microsoft Excel; both MEL language and Python language can be used to customize the behavior of Maya. In the context of game engines, a

脚本语言是一种高级且相对易用的编程语言，它能让用户便捷地访问引擎的大部分常用功能。因此，程序员和非程序员都可以使用脚本语言来开发新游戏，或定制（或“修改”）现有游戏。

16.9.1 运行时与数据定义

我们应该在这里谨慎地做出一个重要的区分。游戏脚本语言通常有两种类型：

- 数据定义语言。数据定义语言的主要目的是允许用户创建和填充数据结构，以便引擎稍后使用。这类语言通常是声明式的（见下文），可以在离线状态下执行或解析，也可以在数据加载到内存时在运行时执行或解析。

- 运行时脚本语言。运行时脚本语言旨在在引擎运行时执行。这些语言通常用于扩展或定制引擎游戏对象模型和/或其他引擎系统的硬编码功能。

在本节中，我们将主要关注使用运行时脚本语言，通过扩展和自定义游戏的对象模型来实现游戏功能。

16.9.2 编程语言特性

在讨论脚本语言时，大家最好对编程语言术语达成共识。编程语言种类繁多，但大致可以按照一些标准进行分类。让我们简单看一下这些标准：

- 解释型语言与编译型语言。编译型语言的源代码由编译器翻译成机器码，可由 CPU 直接执行。相比之下，解释型语言的源代码要么在运行时直接解析，要么预编译成独立于平台的字节码，然后由虚拟机 (VM) 在运行时执行。虚拟机就像一个虚拟 CPU 的仿真器，而字节码就像一串机器指令。

scripting language is a high-level, relatively easy-to-use programming language that provides its users with convenient access to most of the commonly used features of the engine. As such, a scripting language can be used by programmers and non-programmers alike to develop a new game or to customize—or “mod”—an existing game.

16.9.1 Runtime versus Data Definition

We should be careful to make an important distinction here. Game scripting languages generally come in two flavors:

- *Data-definition languages*. The primary purpose of a data-definition language is to permit users to create and populate data structures that are later consumed by the engine. Such languages are often declarative (see below) and are either executed or parsed offline or at runtime when the data is loaded into memory.
- *Runtime scripting language*. Runtime scripting languages are intended to be executed within the context of the engine at runtime. These languages are usually used to extend or customize the hard-coded functionality of the engine’s game object model and/or other engine systems.

In this section, we’ll focus primarily on using a runtime scripting language for the purpose of implementing gameplay features by extending and customizing the game’s object model.

16.9.2 Programming Language Characteristics

In our discussion of scripting languages, it will be helpful for us all to be on the same page with regard to programming language terminology. There are all sorts of programming languages out there, but they can be classified approximately according to a relatively small number of criteria. Let’s take a brief look at these criteria:

- *Interpreted versus compiled languages*. The source code of a *compiled* language is translated by a program called a compiler into machine code, which can be executed directly by the CPU. In contrast, the source code of an *interpreted* language is either parsed directly at runtime or is precompiled into platform-independent *byte code*, which is then executed by a *virtual machine* (VM) at runtime. A virtual machine acts like an emulation of an imaginary CPU, and byte code acts like a list of machine

虚拟 CPU 执行的语言指令。虚拟机的优势在于它可以轻松移植到几乎任何硬件平台，并嵌入到主机应用程序中，例如游戏引擎。我们为这种灵活性付出的最大代价是执行速度——虚拟机执行字节码指令的速度通常比原生 CPU 执行机器语言指令的速度慢得多。

• 命令式语言。在命令式语言中，程序由一系列指令描述，每条指令执行一项操作和/或更改内存中数据的状态。C 和 C++ 都是命令式语言。

• 声明式语言。声明式语言描述了要做什么，但没有明确说明如何获得结果。最终决定权留给了语言的实现者。Prolog 就是一个声明式语言的例子。HTML 和 TeX 等标记语言也可以归类为声明式语言。

• 函数式语言。函数式语言从技术上讲是声明式语言的一个子集，旨在完全避免状态。在函数式语言中，程序由一组函数定义。每个函数都会产生其结果，且没有副作用（即，除了产生输出数据外，它不会对系统造成任何可观察到的更改）。程序的构建过程是将输入数据从一个函数传递到另一个函数，直到生成最终所需的结果。这些语言往往非常适合实现数据处理流水线。它们在实现多线程应用程序时也具有明显的优势，因为函数式语言没有可变状态，因此不需要互斥锁。OCaml、Haskell 和 F# 都是函数式语言的例子。

• 过程式语言 vs. 面向对象语言。在过程式语言中，程序构造的主要单元是过程（或函数）。这些过程和函数执行操作、计算结果和/或更改内存中各种数据结构的状态。相比之下，面向对象语言的程序构造的主要单元是类，它是一种数据结构，与一组“知道”如何管理和操作该数据结构中数据的过程/函数紧密耦合。

• 反射型语言。在反射型语言中，系统中的数据类型、数据成员布局、函数和层次类关系等信息可在运行时检查。在非反射型语言中，这些元信息大部分仅供参考。

language instructions that are consumed by this virtual CPU. The benefit of a virtual machine is that it can be quite easily ported to almost any hardware platform and embedded within a host application like a game engine. The biggest cost we pay for this flexibility is execution speed—a virtual machine usually executes its byte code instructions much more slowly than the native CPU executes its machine language instructions.

- *Imperative languages*. In an imperative language, a program is described by a sequence of instructions, each of which performs an operation and/or changes the state of data in memory. C and C++ are imperative languages.
- *Declarative languages*. A declarative language describes *what* is to be done but does not specify exactly *how* the result should be obtained. That decision is left up to the people implementing the language. Prolog is an example of a declarative language. Mark-up languages like HTML and TeX can also be classified as declarative languages.
- *Functional languages*. Functional languages, which are technically a subset of declarative languages, aim to avoid state altogether. In a functional language, programs are defined by a collection of functions. Each function produces its results with no side-effects (i.e., it causes no observable changes to the system, other than to produce its output data). A program is constructed by passing input data from one function to the next until the final desired result has been generated. These languages tend to be well-suited to implementing data processing pipelines. They also offer distinct advantages when implementing multithreaded applications, because with no mutable state, a functional language requires no mutex locking. OCaml, Haskell and F# are examples of functional languages.
- *Procedural versus object-oriented languages*. In a procedural language, the primary atom of program construction is the *procedure* (or *function*). These procedures and functions perform operations, calculate results and/or change the state of various data structures in memory. In constrast, an object-oriented language's primary unit of program construction is the *class*, a data structure that is tightly coupled with a set of procedures/functions that "know" how to manage and manipulate the data within that data structure.
- *Reflective languages*. In a reflective language, information about the data types, data member layouts, functions and hierarchical class relationships in the system is available for inspection at runtime. In a non-reflective language, the majority of this meta-information is known only

在编译时；只有非常有限的部分暴露给运行时代码。C# 是反射语言的一个例子，而 C 和 C++ 是非反射语言的例子。

16.9.2.1 游戏脚本语言的典型特征

游戏脚本语言与其原生编程语言的区别特征包括：

• 解释型。大多数游戏脚本语言由虚拟机解释执行，而非编译型。这种选择是为了提高灵活性、可移植性和快速迭代（见下文）。当代码以独立于平台的字节码形式表示时，引擎可以轻松地将其视为数据。它可以像任何其他资源一样加载到内存中，而无需操作系统的帮助（例如，PC 平台上的 DLL 或 PlayStation 3 上的 PRX 就需要操作系统的帮助）。由于代码由虚拟机执行，而不是直接由 CPU 执行，因此游戏引擎在脚本代码的运行方式和时间方面拥有极大的灵活性。

• 轻量级。大多数游戏脚本语言都是为嵌入式系统设计的。因此，它们的虚拟机往往很简单，占用的内存也往往很小。

• 支持快速迭代。每当本机代码发生更改时，都必须重新编译和重新链接程序，并且必须关闭并重新运行游戏才能看到更改的效果（除非您的开发环境支持某种形式的“编辑并继续”功能）。另一方面，当脚本代码发生更改时，通常可以非常快速地看到更改的效果。某些游戏引擎允许动态重新加载脚本代码，而无需关闭游戏。其他引擎则要求关闭并重新运行游戏。但无论哪种方式，从进行更改到在游戏中看到其效果的周转时间通常比对本机语言源代码进行更改要快得多。

• 便捷易用。脚本语言通常会根据特定游戏的需求进行定制。可以提供一些功能，使常见任务变得简单、直观且不易出错。例如，游戏脚本语言可能提供一些函数或自定义语法，用于按名称查找游戏对象、发送和处理事件、暂停或操控时间流逝、等待指定时间流逝、实现有限状态机、公开可调整的参数……

at compile time; only a very limited amount of it is exposed to the runtime code. C# is an example of a reflective language, while C and C++ are examples of non-reflective languages.

16.9.2.1 Typical Characteristics of Game Scripting Languages

The characteristics that set a game *scripting* language apart from its *native* programming language brethren include:

- *Interpreted*. Most game scripting languages are interpreted by a virtual machine, not compiled. This choice is made in the interest of flexibility, portability and rapid iteration (see below). When code is represented as platform-independent byte code, it can easily be treated like data by the engine. It can be loaded into memory just like any other asset rather than requiring help from the operating system (as is necessary with a DLL on a PC platform or a PRX on the PlayStation 3, for example). Because the code is executed by a virtual machine rather than directly by the CPU, the game engine is afforded a great deal of flexibility regarding how and when script code will be run.
- *Lightweight*. Most game scripting languages have been designed for use in an embedded system. As such, their virtual machines tend to be simple, and their memory footprints tend to be quite small.
- *Support for rapid iteration*. Whenever native code is changed, the program must be recompiled and relinked, and the game must be shut down and rerun in order to see the effects of the changes (unless your development environment supports some form of edit-and-continue). On the other hand, when script code is changed, the effects of the changes can usually be seen very rapidly. Some game engines permit script code to be reloaded on the fly, without shutting down the game at all. Others require the game to be shut down and rerun. But either way, the turnaround time between making a change and seeing its effects in-game is usually much faster than when making changes to the native language source code.
- *Convenience and ease of use*. Scripting languages are often customized to suit the needs of a particular game. Features can be provided that make common tasks simple, intuitive and less error-prone. For example, a game scripting language might provide functions or custom syntax for finding game objects by name, sending and handling events, pausing or manipulating the passage of time, waiting for a specified amount of time to pass, implementing finite state machines, exposing tweakable param-

将其提供给世界编辑器供游戏设计师使用，甚至可以处理多人游戏的网络复制。

16.9.3 一些常见的游戏脚本语言

在实现运行时游戏脚本系统时，我们必须做出一个基本选择：我们是否选择第三方商业或开源语言并根据我们的需求进行定制，还是从头开始设计和实现自定义语言？

从头开始创建自定义语言通常不值得，而且在整个项目过程中维护成本也很高。聘请熟悉自定义内部语言的游戏设计师和程序员也很困难，甚至根本不可能，因此通常还需要培训成本。然而，这显然是最灵活、可定制性最高的方案，这种灵活性值得投入。

对于许多工作室来说，选择一种知名度较高且成熟的脚本语言，并为其添加游戏引擎特有的功能，会更加便捷。市面上有很多第三方脚本语言可供选择，其中许多语言成熟可靠，已在游戏行业内外的众多项目中得到广泛应用。

在以下部分中，我们将探讨一些自定义游戏脚本语言和一些通常适用于游戏引擎的游戏无关语言。

16.9.3.1 地震C

Id Software 的 John Carmack 为 Quake 实现了一种自定义脚本语言，称为 QuakeC (QC)。该语言本质上是 C 编程语言的简化版本，可直接与 Quake 引擎集成。它不支持指针或定义任意结构体，但可以便捷地操作实体（Quake 对游戏对象的称呼），并可用于发送和接收/处理游戏事件。QuakeC 是一种解释型、命令式、过程式编程语言。

QuakeC 将权力交到玩家手中，这也是如今所谓的“模组社区”诞生的因素之一。脚本语言和其他形式的数据驱动定制技术，让玩家能够将许多商业游戏转化为各种全新的游戏体验——从对原始主题的细微修改，到打造全新的游戏。

16.9.3.2 虚幻脚本

最著名的完全自定义脚本语言示例可能是
虚幻引擎的 UnrealScript 。这种语言基于类似 C++ 的语法。

eters to the world editor for use by the game designers, or even handling network replication for multiplayer games.

16.9.3 Some Common Game Scripting Languages

When implementing a runtime game scripting system, we have one fundamental choice to make: Do we select a third-party commercial or open source language and customize it to suit our needs, or do we design and implement a custom language from scratch?

Creating a custom language from scratch is usually not worth the hassle and the cost of maintenance throughout the project. It can also be difficult or impossible to hire game designers and programmers who are already familiar with a custom, in-house language, so there's usually a training cost as well. However, this is clearly the most flexible and customizable approach, and that flexibility can be worth the investment.

For many studios, it is more convenient to select a reasonably well-known and mature scripting language and extend it with features specific to your game engine. There are a great many third-party scripting languages from which to choose, and many are mature and robust, having been used in a great many projects both within and outside the game industry.

In the following sections, we'll explore a number of custom game scripting languages and a number of game-agnostic languages that are commonly adapted for use in game engines.

16.9.3.1 QuakeC

Id Software's John Carmack implemented a custom scripting language for *Quake*, known as *QuakeC* (QC). This language was essentially a simplified variant of the C programming language with direct hooks into the *Quake* engine. It had no support for pointers or defining arbitrary structs, but it could manipulate *entities* (*Quake*'s name for game objects) in a convenient manner, and it could be used to send and receive/handle game events. QuakeC is an interpreted, imperative, procedural programming language.

The power that QuakeC put into the hands of gamers is one of the factors that gave birth to what is now known as the *mod community*. Scripting languages and other forms of data-driven customization allow gamers to turn many commercial games into all sorts of new gaming experiences—from slight modifications on the original theme to entirely new games.

16.9.3.2 UnrealScript

Probably the best-known example of an entirely custom scripting language is Unreal Engine's *UnrealScript*. This language is based on a C++-like syntacti-

它采用 C 风格，并支持 C 和 C++ 程序员所熟悉的大多数概念，包括类、局部变量、循环、用于数据组织的数组和结构体、字符串、散列字符串 ID（在 Unreal 中称为 FName）以及对象引用（但不支持自由格式的指针）。UnrealScript 是一种解释型、命令式、面向对象的语言。

Epic 不再支持 UnrealScript 语言。开发者可以通过蓝图图形"脚本"系统或编写 C++ 代码来自定义游戏行为。

16.9.3.3 第二

Lua 是一种著名且流行的脚本语言，易于集成到游戏引擎等应用程序中。Lua 网站 (http://www.lua.org/about.html) 称其为"游戏领域的领先脚本语言"。

根据 Lua 网站介绍，Lua 的主要优点是：

- 强大而成熟。Lua 已被用于众多商业产品，包括 Adob□□e 的 Photoshop Lightroom，以及许多游戏，包括《魔兽世界》。
- 完善的文档。Lua 的参考手册 [25] 完整易懂，可以在线获取，也可以以书籍形式获取。目前已经有很多关于 Lua 的书籍，包括 [26] 和 [50]。
- 出色的运行时性能。Lua 比许多其他脚本语言更快、更高效地执行其字节码。
- 可移植性。Lua 开箱即用，可在各种 Windows 和 UNIX 系统、移动设备以及嵌入式微处理器上运行。Lua 采用可移植的方式编写，使其易于适应新的硬件平台。
- 专为嵌入式系统设计。Lua 的内存占用非常小（解释器和所有库大约占用 350 KiB）。
- 简洁、强大且可扩展。Lua 语言的核心非常精简，但它被设计为支持元机制，从而能够以几乎无限的方式扩展其核心功能。例如，Lua 本身并非面向对象语言，但可以通过元机制添加 OOP 支持。
- 免费。Lua 是开源的，并根据非常自由的 MIT 许可证进行分发。

Lua 是一种动态类型语言，这意味着变量没有类型，只有值才有。（每个值都带有其类型信息。）Lua 的主要数据结构是表，也称为关联数组。

cal style, and it supports most of the concepts that C and C++ programmers have become accustomed to, including classes, local variables, looping, arrays and structs for data organization, strings, hashed string ids (called `FName` in Unreal) and object references (but not free-form pointers). UnrealScript is an interpreted, imperative, object-oriented language.

Epic no longer supports the UnrealScript language. Instead, developers customize game behavior either via the Blueprints graphical "scripting" system, or by writing C++ code.

16.9.3.3 Lua

Lua is a well-known and popular scripting language that is easy to integrate into an application such as a game engine. The Lua website (http://www.lua.org/about.html) calls the language the "leading scripting language in games."

According to the Lua website, Lua's key benefits are:

- *Robust and mature*. Lua has been used on numerous commercial products, including Adobe's *Photoshop Lightroom*, and many games, including *World of Warcraft*.
- *Good documentation*. Lua's reference manual [25] is complete and understandable and is available online and in book formats. A number of books have been written about Lua, including [26] and [50].
- *Excellent runtime performance*. Lua executes its byte code more quickly and efficiently than many other scripting languages.
- *Portable*. Out of the box, Lua runs on all flavors of Windows and UNIX, mobile devices and embedded microprocessors. Lua is written in a portable manner, making it easy to adapt to new hardware platforms.
- *Designed for embedded systems*. Lua's memory footprint is very small (approximately 350 KiB for the interpreter and all libraries).
- *Simple, powerful and extensible*. The core Lua language is very small and simple, but it is designed to support meta-mechanisms that extend its core functionality in virtually limitless ways. For example, Lua itself is not an object-oriented language, but OOP support can and has been added via a meta-mechanism.
- *Free*. Lua is open source and is distributed under the very liberal MIT license.

Lua is a dynamically typed language, meaning that variables don't have types—only values do. (Every value carries its type information along with it.) Lua's primary data structure is the *table*, also known as an associative array. A

表本质上是键值对的列表，具有按键索引数组的优化能力。

Lua 为 C 语言提供了一个方便的接口——Lua 虚拟机可以像调用和操作用 Lua 本身编写的函数一样轻松地调用和操作用 C 编写的函数。

Lua 将代码块（称为 chunk ）视为可由 Lua 程序本身操作的一级对象。代码可以以源代码格式或预编译的字节码格式执行。这使得虚拟机可以执行包含 Lua 代码的字符串，就像代码被编译到原始程序中一样。Lua 还支持一些强大的高级编程结构，包括 协程 。这是一种简单的协作式多任务处理形式，其中每个线程必须明确地将 CPU 让给其他线程（而不是像在抢占式多线程系统中那样进行时间分片）。

Lua 确实存在一些缺陷。例如，它灵活的函数绑定机制使得重新定义一个重要的全局函数（例如 sin()）来执行完全不同的任务（这通常不是人们想要做的事情）成为可能（而且相当容易）。但总而言之，Lua 已经证明自己是游戏脚本语言的绝佳选择。

16.9.3.4 Python

Python 是一种过程式、面向对象、动态类型的脚本语言，其设计理念是易于使用、与其他编程语言集成以及灵活性。与 Lua 一样，Python 也是游戏脚本语言的常见选择。根据 Python 官方网站 (http://www.python.org) 的介绍，Python 的一些最佳特性包括：

- 清晰易读的语法。Python 代码易于阅读，部分原因在于其语法强制采用特定的缩进样式。（它实际上会解析用于缩进的空格，以确定每行代码的作用域。）
- 反射式语言。Python 包含强大的运行时自省功能。Python 中的类是一等对象，这意味着它们可以在运行时被操作和查询，就像任何其他对象一样。
- 面向对象。Python 相较于 Lua 的一个优势是其核心语言内置了面向对象编程 (OOP)。这使得 Python 与游戏对象模型的集成更加容易。
- 模块化。Python 支持分层包，鼓励清晰的系统设计和良好的封装。

table is essentially a list of key-value pairs with an optimized ability to index into the array by key.

Lua provides a convenient interface to the C language—the Lua virtual machine can call and manipulate functions written in C as easily as it can those written in Lua itself.

Lua treats blocks of code, called *chunks*, as first-class objects that can be manipulated by the Lua program itself. Code can be executed in source code format or in precompiled byte code format. This allows the virtual machine to execute a string that contains Lua code, just as if the code were compiled into the original program. Lua also supports some powerful advanced programming constructs, including *coroutines*. This is a simple form of cooperative multitasking, in which each thread must yield the CPU to other threads explicitly (rather than being time-sliced as in a preemptive multithreading system).

Lua does have some pitfalls. For example, its flexible function binding mechanism makes it possible (and quite easy) to redefine an important global function like `sin()` to perform a totally different task (which is usually not something one intends to do). But all in all, Lua has proven itself to be an excellent choice for use as a game scripting language.

16.9.3.4 Python

Python is a procedural, object-oriented, dynamically typed scripting language designed with ease of use, integration with other programming languages, and flexibility in mind. Like Lua, Python is a common choice for use as a game scripting language. According to the official Python website (http://www.python.org), some of Python's best features include:

- *Clear and readable syntax*. Python code is easy to read, in part because the syntax enforces a specific indentation style. (It actually parses the whitespace used for indentation in order to determine the scope of each line of code.)
- *Reflective language*. Python includes powerful runtime introspection facilities. Classes in Python are first-class objects, meaning they can be manipulated and queried at runtime, just like any other object.
- *Object-oriented*. One advantage of Python over Lua is that OOP is built into the core language. This makes integrating Python with a game's object model a little easier.
- *Modular*. Python supports hierarchical packages, encouraging clean system design and good encapsulation.

- 基于异常的错误处理。与非基于异常的语言中的类似代码相比，异常使得 Python 中的错误处理代码更简单、更优雅、更本地化。

- 丰富的标准库和第三方模块。Python 库几乎涵盖了所有你能想到的任务。（真的！）
- 可嵌入。Python 可以轻松嵌入到应用程序中，例如游戏引擎。

- 丰富的文档。Python 有大量的文档和教程，既有在线的，也有书籍形式的。Python 网站 http://www.python.org 是一个不错的起点。

Python 的语法在很多方面都与 C 语言相似（例如，它使用 = 运算符进行赋值，使用 == 进行相等性测试）。然而，在 Python 中，代码缩进是定义作用域的唯一方式（与 C 语言的左括号和右括号不同）。Python 的主要数据结构是列表（由原子值或其他嵌套列表组成的线性索引序列）和字典（由键值对组成的表）。这两种数据结构都可以保存彼此的实例，从而可以轻松构建任意复杂的数据结构。此外，类（数据元素和函数的统一集合）也内置于 Python 语言中。

Python 支持鸭子类型，这是一种动态类型，其中对象的功能接口决定了对象的类型（而不是由静态继承层次结构定义）。换句话说，任何支持特定接口的类（即具有特定签名的函数集合）都可以与支持相同接口的任何其他类互换使用。这是一个强大的范例：实际上，Python 支持多态性而无需使用继承。鸭子类型在某些方面类似于 C++ 模板元编程，尽管它可以说更灵活，因为调用者和被调用者之间的绑定是在运行时动态形成的。鸭子类型的名字来源于詹姆斯 · 惠特科姆 · 莱利的一句名言："如果它走起来像鸭子，叫起来也像鸭子，我就称它为鸭子。"有关鸭子类型的更多信息，请参阅 http://en.wikipedia.org/wiki/Duck_typing。

总而言之，Python 易于使用和学习，可以轻松嵌入到游戏引擎中，与游戏的对象模型很好地集成，并且可以成为一种出色且强大的游戏脚本语言选择。

16.9.3.5 兵/小/小-C

Pawn 是由 Marc Peter 创建的一种轻量级、动态类型、类似 C 的脚本语言。该语言之前名为 Small，而 Small 本身就是

- *Exception-based error handling*. Exceptions make error-handling code in Python simpler, more elegant and more localized than similar code in a non-exception-based language.
- *Extensive standard libraries and third-party modules*. Python libraries exist for virtually every task imaginable. (Really!)
- *Embeddable*. Python can be easily embedded into an application, such as a game engine.
- *Extensive documentation*. There's plenty of documentation and tutorials on Python, both online and in book form. A good place to start is the Python website, http://www.python.org.

Python syntax is reminiscent of C in many respects (for example, its use of the = operator for assignment and == for equality testing). However, in Python, *code indentation* serves as the only means of defining *scope* (as opposed to C's opening and closing braces). Python's primary data structures are the *list*—a linearly indexed sequence of atomic values or other nested lists—and the *dictionary*—a table of key-value pairs. Each of these two data structures can hold instances of the other, allowing arbitrarily complex data structures to be constructed easily. In addition, *classes*—unified collections of data elements and functions—are built right into the language.

Python supports *duck typing*, which is a style of dynamic typing in which the functional interface of an object determines its type (rather than being defined by a static inheritance hierarchy). In other words, any class that supports a particular interface (i.e., a collection of functions with specific signatures) can be used interchangeably with any other class that supports that same interface. This is a powerful paradigm: In effect, Python supports polymorphism without requiring the use of inheritance. Duck typing is similar in some respects to C++ template metaprogramming, although it is arguably more flexible because the bindings between caller and callee are formed dynamically, at runtime. Duck typing gets its name from the well-known phrase (attributed to James Whitcomb Riley), "If it walks like a duck and quacks like a duck, I would call it a duck." See http://en.wikipedia.org/wiki/Duck_typing for more information on duck typing.

In summary, Python is easy to use and learn, embeds easily into a game engine, integrates well with a game's object model, and can be an excellent and powerful choice as a game scripting language.

16.9.3.5 Pawn/Small/Small-C

Pawn is a lightweight, dynamically typed, C-like scripting language created by Marc Peter. The language was formerly known as *Small*, which itself was

Small-C 是 C 语言早期子集 Small-C 的演化版本，由 Ron Cain 和 James Hendrix 编写。Small-C 是一种解释型语言——源代码被编译成字节码（也称为 P 码），由虚拟机在运行时解释执行。

Pawn 的设计目标是占用较少的内存，并能快速执行其字节码。与 C 语言不同，Pawn 的变量是动态类型的。Pawn 还支持有限状态机，包括状态局部变量。这一独特特性使其非常适合许多游戏应用程序。Pawn 提供了完善的在线文档（http://www.compuphase.com/pawn/pawn.htm）。Pawn 是开源的，可根据 Zlib/libpng 许可证免费使用 (http://www.opensource.org/licenses/zlib-license.php)。

Pawn 的语法与 C 语言类似，这使得任何 C/C++ 程序员都可以轻松学习，并且易于与用 C 语言编写的游戏引擎集成。它对有限状态机的支持对于游戏编程非常有用。它已成功应用于许多游戏项目，包括 Midway 的《Freaky Flyers》。Pawn 已证明自己是一种可行的游戏脚本语言。

16.9.4 脚本架构

脚本代码在游戏引擎中可以扮演各种角色。其架构多种多样，从代表对象或引擎系统执行简单功能的脚本代码片段，到管理游戏运行的高级脚本。以下仅列举部分可能的架构：

- 脚本回调。在这种方法中，引擎的功能大部分是用本机编程语言硬编码的，但某些关键功能被设计为可定制的。这通常通过钩子函数或回调来实现——用户提供的函数，由引擎调用以允许定制。钩子函数当然可以用本机语言编写，但也可以用脚本语言编写。例如，在游戏循环期间更新游戏对象时，引擎可能会调用一个可以用脚本编写的可选回调函数。这使用户有机会自定义游戏对象随时间更新自身的方式。

- 脚本事件处理程序。事件处理程序实际上只是一种特殊类型的钩子函数，其目的是允许游戏对象响应游戏世界中发生的某些相关事件（例如，响应爆炸发生）或引擎本身中的某些相关事件（例如，响应

an evolution of an earlier subset of the C language called *Small-C*, written by Ron Cain and James Hendrix. It is an interpreted language—the source code is compiled into byte code (also known as P-code), which is interpreted by a virtual machine at runtime.

Pawn was designed to have a small memory footprint and to execute its byte code very quickly. Unlike C, Pawn's variables are dynamically typed. Pawn also supports finite state machines, including state-local variables. This unique feature makes it a good fit for many game applications. Good online documentation is available for Pawn (http://www.compuphase.com/pawn/pawn.htm). Pawn is open source and can be used free of charge under the Zlib/libpng license (http://www.opensource.org/licenses/zlib-license.php).

Pawn's C-like syntax makes it easy to learn for any C/C++ programmer and easy to integrate with a game engine written in C. Its finite state machine support can be very useful for game programming. It has been used successfully on a number of game projects, including *Freaky Flyers* by Midway. Pawn has shown itself to be a viable game scripting language.

16.9.4 Architectures for Scripting

Script code can play all sorts of roles within a game engine. There's a gamut of possible architectures, from tiny snippets of script code that perform simple functions on behalf of an object or engine system to high-level scripts that manage the operation of the game. Here are just a few of the possible architectures:

- *Scripted callbacks*. In this approach, the engine's functionality is largely hard-coded in the native programming language, but certain key bits of functionality are designed to be customizable. This is often implemented via a *hook function* or *callback*—a user-supplied function that is called by the engine for the purpose of allowing customization. Hook functions can be written in the native language, of course, but they can also be written in a scripting language. For example, when updating game objects during the game loop, the engine might call an optional callback function that can be written in script. This gives users the opportunity to customize the way in which the game object updates itself over time.
- *Scripted event handler*. An event handler is really just a special type of hook function whose purpose is to allow a game object to respond to some relevant occurrence within the game world (e.g., responding to an explosion going off) or within the engine itself (e.g., responding to

内存不足的情况）。许多游戏引擎允许用户使用脚本和本机语言编写事件处理程序挂钩。

• 使用脚本扩展游戏对象类型或定义新的类型。某些脚本语言允许通过脚本扩展已用原生语言实现的游戏对象类型。事实上，回调和事件处理程序只是小规模的例子，但这个想法甚至可以扩展到允许在脚本中定义全新类型的游戏对象。这可以通过继承（即，从原生语言编写的类派生出用脚本编写的类）或组合（即，将脚本类的实例附加到原生游戏对象）来实现。

• 脚本组件或属性。在基于组件或属性的游戏对象模型中，只有允许部分或全部使用脚本构建新组件或属性对象才有意义。Gas Powered Games 在《地牢围攻》(Dungeon Siege) 中就采用了这种方法。该游戏对象模型基于属性，可以使用 C++ 或 Gas Powered Games 的自定义脚本语言 Skrit (http://ds.heavengames.com/library/dstk/skrit/skrit) 实现属性。项目结束时，他们拥有大约 148 种脚本属性类型和 21 种原生 C++ 属性类型。

• 脚本驱动引擎。脚本可以用来驱动整个引擎系统。例如，游戏对象模型可以完全用脚本编写，只有当需要底层引擎组件的服务时才调用原生引擎代码。

• 脚本驱动的游戏。一些游戏引擎实际上完全颠覆了原生语言和脚本语言之间的关系。在这些引擎中，脚本代码运行整个流程，而原生引擎代码仅仅充当一个库，用于访问引擎的某些高速功能。Panda3D 引擎 (http://www.panda3d.org) 就是这种架构的一个例子。Panda3D 游戏可以完全用 Python 语言编写，而原生引擎（用 C++ 实现）则充当一个库，由脚本代码调用。

（Panda3D 游戏也可以完全用 C++ 编写。）

16.9.5 运行时游戏脚本语言的功能

许多游戏脚本语言的主要目的是实现游戏功能，这通常是通过扩充和自定义游戏的对象模型来实现的。在本节中，我们将探讨此类脚本系统的一些最常见的需求和功能。

an out-of-memory condition). Many game engines allow users to write event handler hooks in script as well as in the native language.

- *Extending game object types, or defining new ones, with script*. Some scripting languages allow game object types that have been implemented in the native language to be extended via script. In fact, callbacks and event handlers are examples of this on a small scale, but the idea can be extended even to the point of allowing entirely new types of game objects to be defined in script. This might be done via *inheritance* (i.e., deriving a class written in script from a class written in the native language) or via *composition* (i.e., attaching an instance of a scripted class to a native game object).
- *Scripted components or properties*. In a component- or property-based game object model, it only makes sense to permit new components or property objects to be constructed partially or entirely in script. This approach was used by Gas Powered Games for *Dungeon Siege*. The game object model was property-based, and it was possible to implement properties in either C++ or Gas Powered Games' custom scripting language, *Skrit* (http://ds.heavengames.com/library/dstk/skrit/skrit). By the end of the project, they had approximately 148 scripted property types and 21 native C++ property types.
- *Script-driven engine*. Script might be used to drive an entire engine system. For example, the game object model could conceivably be written entirely in script, calling into the native engine code only when it requires the services of lower-level engine components.
- *Script-driven game*. Some game engines actually flip the relationship between the native language and the scripting language on its head. In these engines, the script code runs the whole show, and the native engine code acts merely as a library that is called to access certain high-speed features of the engine. The Panda3D engine (http://www.panda3d.org) is an example of this kind of architecture. Panda3D games can be written entirely in the Python language, and the native engine (implemented in C++) acts like a library that is called by script code. (Panda3D games can also be written entirely in C++.)

16.9.5 Features of a Runtime Game Scripting Language

The primary purpose of many game scripting languages is to implement gameplay features, and this is often accomplished by augmenting and customizing a game's object model. In this section, we'll explore some of the most common requirements and features of such a scripting system.

16.9.5.1 与本机编程语言的接口

为了使脚本语言发挥作用，它不能孤立地运行。游戏引擎必须能够执行脚本代码，而脚本代码能够启动引擎内部的操作通常也同样重要。

运行时脚本语言的虚拟机 (VM) 通常嵌入在游戏引擎中。引擎初始化虚拟机，在需要时运行脚本代码，并管理这些脚本的执行。执行单元根据语言的具体特性和游戏的具体实现而有所不同。

- 在函数式脚本语言中，函数通常是主要的执行单元。为了让引擎调用脚本函数，它必须查找与所需函数名称对应的字节码，并启动一个虚拟机来执行该函数（或指示现有的虚拟机执行该函数）。

- 在面向对象的脚本语言中，类通常是主要的执行单元。在这样的系统中，对象可以被创建和销毁，并且方法（成员函数）可以在单个类实例上调用。

允许脚本和本机代码之间的双向通信通常是有益的。因此，大多数脚本语言也允许从脚本调用本机代码。细节因语言和实现而异，但基本方法通常是允许某些脚本函数用本机语言而不是脚本语言实现。要调用引擎函数，脚本代码只需进行普通的函数调用。虚拟机检测到该函数具有本机实现，查找相应的本机函数的地址（可能通过名称或通过某种其他类型的唯一函数标识符），然后调用它。例如，Python 类或模块的部分或全部成员函数可以使用 C 函数实现。Python 维护一个称为方法表的数据结构，它将每个 Python 函数的名称（表示为字符串）映射到实现它的 C 函数的地址。

案例研究：顽皮狗的 DC 语言

作为一个例子，我们来简单看一下顽皮狗的运行时脚本语言（一种名为 DC 的语言）是如何集成到引擎中的。

DC 是 Scheme 语言（它本身也是 Lisp 语言的变体）的一个变体。DC 中的可执行代码块被称为脚本 lambda 表达式，它

16.9.5.1 Interface with the Native Programming Language

In order for a scripting language to be useful, it must not operate in a vacuum. It's imperative for the game engine to be able to execute script code, and it's usually equally important for script code to be capable of initiating operations within the engine as well.

A runtime scripting language's virtual machine (VM) is generally embedded within the game engine. The engine initializes the virtual machine, runs script code whenever required, and manages those scripts' execution. The unit of execution varies depending on the specifics of the language and the game's implementation.

- In a functional scripting language, the *function* is often the primary unit of execution. In order for the engine to call a script function, it must look up the byte code corresponding to the name of the desired function and spawn a virtual machine to execute it (or instruct an existing VM to do so).
- In an object-oriented scripting language, *classes* are typically the primary unit of execution. In such a system, objects can be spawned and destroyed, and methods (member functions) can be invoked on individual class instances.

It's usually beneficial to allow two-way communication between script and native code. Therefore, most scripting languages allowing native code to be invoked from script as well. The details are language- and implementation-specific, but the basic approach is usually to allow certain script functions to be implemented in the native language rather than in the scripting language. To call an engine function, script code simply makes an ordinary function call. The virtual machine detects that the function has a native implementation, looks up the corresponding native function's address (perhaps by name or via some other kind of unique function identifier), and calls it. For example, some or all of the member functions of a Python class or module can be implemented using C functions. Python maintains a data structure, known as a *method table*, that maps the name of each Python function (represented as a string) to the address of the C function that implements it.

Case Study: Naughty Dog's DC Language

As an example, let's have a brief look at how Naughty Dog's runtime scripting language, a language called DC, was integrated into the engine.

DC is a variant of the Scheme language (which is itself a variant of Lisp language). Chunks of executable code in DC are known as *script lambdas*, which

大致相当于 Lisp 语言家族中的函数或代码块。DC 程序员编写脚本 lambda，并通过赋予它们全局唯一的名称来标识它们。DC 编译器将这些脚本 lambda 转换为字节码块，这些字节码块在游戏运行时加载到内存中，并可以通过 C++ 中简单的函数式接口通过名称进行查找。

一旦引擎获得指向一段脚本 lambda 字节码的指针，它就可以通过调用引擎中的“虚拟机执行”函数并将字节码指针传递给它来执行该代码。该函数本身非常简单。它循环执行，逐条读取字节码指令，并执行每条指令。当所有指令都执行完毕后，函数返回。

虚拟机包含一组寄存器，用于保存脚本需要处理的任何类型的数据。这通过变体数据类型（所有数据类型的联合）实现（有关变体的讨论，请参见第 16.8.4 节）。一些指令用于将数据加载到寄存器中；另一些指令用于查找和使用寄存器中保存的数据。有些指令用于执行该语言中所有可用的数学运算，有些指令用于执行条件检查——例如 DC 的 (if ...) 、 (when ...) 和 (cond ...) 指令的实现等等。

虚拟机还支持函数调用堆栈。DC 中的脚本 lambda 可以调用脚本程序员通过 DC 的 (defun ...) 语法定义的其他脚本 lambda（即函数）。与任何过程式编程语言一样，当一个函数调用另一个函数时，需要一个堆栈来跟踪寄存器的状态和返回地址。在 DC 虚拟机中，调用堆栈实际上是一个寄存器组堆栈 - 每个新函数都有自己的私有寄存器组。这使我们不必保存寄存器状态、调用函数，然后在被调用函数返回时恢复寄存器。当虚拟机遇到告诉它调用另一个脚本 lambda 的字节码指令时，将按名称查找该脚本 lambda 的字节码，推送一个新的堆栈帧，并从该脚本 lambda 的第一条指令继续执行。当虚拟机遇到返回指令时，堆栈帧将从堆栈中弹出，同时弹出返回“地址”（实际上只是调用脚本 lambda 中在首先调用该函数之后的字节码指令的索引）。

以下伪代码应该可以让您了解 DC 虚拟机的核心指令处理循环是什么样的：

are the approximate equivalent of functions or code blocks in the Lisp family of languages. A DC programmer writes script lambdas and identifies them by giving them globally unique names. The DC compiler converts these script lambdas into chunks of byte code, which are loaded into memory when the game runs and can be looked up by name using a simple functional interface in C++.

Once the engine has a pointer to a chunk of script lambda byte code, it can execute the code by calling a “virtual machine execution” function in the engine and passing the byte code pointer to it. The function itself is surprisingly simple. It spins in a loop, reading byte code instructions one-by-one, and executing each instruction. When all instructions have been executed, the function returns.

The virtual machine contains a bank of registers, which can hold any kind of data the script may want to deal with. This is implemented using a *variant* data type—a union of all the data types (see Section 16.8.4 for a discussion of variants). Some instructions cause data to be loaded into a register; others cause the data held in a register to be looked up and used. There are instructions for performing all of the mathematical operations available in the language, as well as instructions for performing conditional checks—implementations of DC’s `(if ...)`, `(when ...)` and `(cond ...)` instructions and so on.

The virtual machine also supports a *function call stack*. Script lambdas in DC can call other script lambdas (i.e., functions) that have been defined by a script programmer via DC’s `(defun ...)` syntax. Just like any procedural programming language, a stack is needed to keep track of the states of the registers and the return address when one function calls another. In the DC virtual machine, the call stack is literally a stack of register banks—each new function gets its own private bank of registers. This prevents us from having to save off the state of the registers, call the function, and then restore the registers when the called function returns. When the virtual machine encounters a byte code instruction that tells it to call another script lambda, the byte code for that script lambda is looked up by name, a new stack frame is pushed, and execution continues at the first instruction of that script lambda. When the virtual machine encounters a return instruction, the stack frame is popped from the stack, along with the return “address” (which is really just the index of the byte code instruction in the calling script lambda after the one that called the function in the first place).

The following pseudocode should give you a feel for what the core instruction-processing loop of the DC virtual machine looks like:

```
void DcExecuteScript(DCByteCode* pCode)
{
    DCStackFrame* pCurStackFrame
     = DcPushStackFrame(pCode);

    // Keep going until we run out of stack frames (i.e.,
    // the top-level script lambda "function" returns).
    while (pCurStackFrame != nullptr)
    {
        // Get the next instruction. We will never run
        // out, because the return instruction is always
        // last, and it will pop the current stack frame
        // below.
        DCInstruction& instr
         = pCurStackFrame->GetNextInstruction();

        // Perform the operation of the instruction.
        switch (instr.GetOperation())
        {
        case DC_LOAD_REGISTER_IMMEDIATE:
            {
                // Grab the immediate value to be loaded
                // from the instruction.
                Variant& data = instr.GetImmediateValue();

                // Also determine into which register to
                // put it.
                U32 iReg = instr.GetDestRegisterIndex();

                // Grab the register from the stack frame.
                Variant& reg
                 = pCurStackFrame->GetRegister(iReg);

                // Store the immediate data into the
                // register.
                reg = data;
            }
            break;

        // Other load and store register operations...

        case DC_ADD_REGISTERS:
            {
                // Determine the two registers to add. The
                // result will be stored in register A.
                U32 iRegA = instr.GetDestRegisterIndex();
                U32 iRegB = instr.GetSrcRegisterIndex();
```



```
void DcExecuteScript(DCByteCode* pCode)
{
    DCStackFrame* pCurStackFrame
     = DcPushStackFrame(pCode);

    // Keep going until we run out of stack frames (i.e.,
    // the top-level script lambda "function" returns).
    while (pCurStackFrame != nullptr)
    {
        // Get the next instruction. We will never run
        // out, because the return instruction is always
        // last, and it will pop the current stack frame
        // below.
        DCInstruction& instr
         = pCurStackFrame->GetNextInstruction();

        // Perform the operation of the instruction.
        switch (instr.GetOperation())
        {
        case DC_LOAD_REGISTER_IMMEDIATE:
            {
                // Grab the immediate value to be loaded
                // from the instruction.
                Variant& data = instr.GetImmediateValue();

                // Also determine into which register to
                // put it.
                U32 iReg = instr.GetDestRegisterIndex();

                // Grab the register from the stack frame.
                Variant& reg
                 = pCurStackFrame->GetRegister(iReg);

                // Store the immediate data into the
                // register.
                reg = data;
            }
            break;

        // Other load and store register operations...

        case DC_ADD_REGISTERS:
            {
                // Determine the two registers to add. The
                // result will be stored in register A.
                U32 iRegA = instr.GetDestRegisterIndex();
                U32 iRegB = instr.GetSrcRegisterIndex();
```

```
            // Grab the 2 register variants from the
            // stack.
            Variant& dataA
             = pCurStackFrame->GetRegister(iRegA);

            Variant& dataB
             = pCurStackFrame->GetRegister(iRegB);

            // Add the registers and store in
            // register A.
            dataA = dataA + dataB;
        }
        break;

    // Other math operations...

    case DC_CALL_SCRIPT_LAMBDA:
        {
            // Determine in which register the name of
            // the script lambda to call is stored.
            // (Presumably it was loaded by a previous
            // load instr.)
            U32 iReg = instr.GetSrcRegisterIndex();

            // Grab the appropriate register, which
            // contains the name of the lambda to call.
            Variant& lambda
             = pCurStackFrame->GetRegister(iReg);

            // Look up the byte code of the lambda by
            // name.
            DCByteCode* pCalledCode
             = DcLookUpByteCode(lambda.AsStringId());

            // Now "call" the lambda by pushing a new
            // stack frame.
            if (pCalledCode)
            {
                pCurStackFrame
                 = DcPushStackFrame(pCalledCode);
            }
        }
        break;

    case DC_RETURN:
        {
```



```
            // Grab the 2 register variants from the
            // stack.
            Variant& dataA
             = pCurStackFrame->GetRegister(iRegA);

            Variant& dataB
             = pCurStackFrame->GetRegister(iRegB);

            // Add the registers and store in
            // register A.
            dataA = dataA + dataB;
        }
        break;

    // Other math operations...

    case DC_CALL_SCRIPT_LAMBDA:
        {
            // Determine in which register the name of
            // the script lambda to call is stored.
            // (Presumably it was loaded by a previous
            // load instr.)
            U32 iReg = instr.GetSrcRegisterIndex();

            // Grab the appropriate register, which
            // contains the name of the lambda to call.
            Variant& lambda
             = pCurStackFrame->GetRegister(iReg);

            // Look up the byte code of the lambda by
            // name.
            DCByteCode* pCalledCode
             = DcLookUpByteCode(lambda.AsStringId());

            // Now "call" the lambda by pushing a new
            // stack frame.
            if (pCalledCode)
            {
                pCurStackFrame
                 = DcPushStackFrame(pCalledCode);
            }
        }
        break;

    case DC_RETURN:
        {
```

```
                // Just pop the stack frame. If we're in
                // the top lambda on the stack, this
                // function will return nullptr, and the
                // loop will terminate.
                pCurStackFrame = DcPopStackFrame();
            }
            break;

        // Other instructions...

            // ...

        } // end switch
    } // end while
}
```

在上面的例子中，我们假设全局函数 DcPushStack Frame() 和 DcPopStackFram e() 以某种合适的方式为我们管理寄存器组堆栈，并且全局函数 DcLookUpByte Code() 能够通过名称查找任何脚本 lambda。我们不会在这里展示这些函数的实现，因为本例的目的仅仅是展示脚本虚拟机的内部循环是如何工作的，而不是提供完整的函数实现。

DC 脚本 lambda 还可以调用本机函数，即用 C++ 编写的全局函数，它们充当引擎本身的钩子。当虚拟机遇到调用本机函数的指令时，会使用引擎程序员硬编码的全局表按名称查找 C++ 函数的地址。如果找到合适的 C++ 函数，则会从当前堆栈帧中的寄存器中获取该函数的参数，然后调用该函数。这意味着 C ++ 函数的参数始终为 Variant 类型。如果 C++ 函数返回一个值，它也必须是 Variant 类型，并且它的值将存储在当前堆栈帧中的寄存器中，以供后续指令使用。

全局函数表可能看起来像这样：

```
typedef Variant DcNativeFunction(U32 argCount,
                                 Variant* aArgs);

struct DcNativeFunctionEntry
{
    StringId           m_name;
    DcNativeFunction*  m_pFunc;
};
```



```
                // Just pop the stack frame. If we're in
                // the top lambda on the stack, this
                // function will return nullptr, and the
                // loop will terminate.
                pCurStackFrame = DcPopStackFrame();
            }
            break;

        // Other instructions...

            // ...

        } // end switch
    } // end while
}
```

In the above example, we assume that the global functions `DcPushStack Frame()` and `DcPopStackFrame()` manage the stack of register banks for us in some suitable way and that the global function `DcLookUpByteCode()` is capable of looking up any script lambda by name. We won't show implementations of those functions here, because the purpose of this example is simply to show how the inner loop of a script virtual machine might work, not to provide a complete functional implementation.

DC script lambdas can also call native functions—i.e., global functions written in C++ that serve as hooks into the engine itself. When the virtual machine comes across an instruction that calls a native function, the address of the C++ function is looked up by name using a global table that has been hard-coded by the engine programmers. If a suitable C++ function is found, the arguments to the function are taken from registers in the current stack frame, and the function is called. This implies that the C++ function's arguments are always of type `Variant`. If the C++ function returns a value, it too must be a `Variant`, and its value will be stored into a register in the current stack frame for possible use by subsequent instructions.

The global function table might look something like this:

```
typedef Variant DcNativeFunction(U32 argCount,
                                 Variant* aArgs);

struct DcNativeFunctionEntry
{
    StringId           m_name;
    DcNativeFunction*  m_pFunc;
};
```

```
DcNativeFunctionEntry g_aNativeFunctionLookupTable[] =
{
    { SID("get-object-pos"), DcGetObjectPos },
    { SID("animate-object"), DcAnimateObject },
    // etc.
};
```

原生 DC 函数实现可能如下所示。请注意，Variant 参数是如何以数组形式传递给函数的。函数必须验证传递给它的参数数量是否等于其预期数量。它还必须验证参数的类型是否符合预期，并准备好处理 DC 脚本程序员在调用函数时可能犯的错误。在顽皮狗，我们编写了一个参数迭代器，它使我们能够以便捷的方式逐个提取和验证参数。

```
Variant DcGetObjectPos(U32 argCount, Variant* aArgs)
{
    // Argument iterator expecting at most 2 args.
    DcArgIterator args(argCount, aArgs, 2);

    // Set up a default return value.
    Variant result;
    result.SetAsVector(Vector(0.0f, 0.0f, 0.0f));

    // Use iterator to extract the args. It flags missing
    // or invalid arguments as errors automatically.
    StringId objectName = args.NextStringId();
    Point* pDefaultPos = args.NextPoint(kDcOptional);

    GameObject* pObject
        = GameObject::LookUpByName(objectName);
    if (pObject)
    {
        result.SetAsVector(pObject->GetPosition());
    }
    else
    {
        if (pDefaultPos)
        {
            result.SetAsVector(*pDefaultPos);
        }
        else
        {
            DcErrorMessage("get-object-pos: "
                           "Object '%s' not found.\n",
                           objectName.ToDebugString());
        }
```



```
DcNativeFunctionEntry g_aNativeFunctionLookupTable[] =
{
    { SID("get-object-pos"), DcGetObjectPos },
    { SID("animate-object"), DcAnimateObject },
    // etc.
};
```

A native DC function implementation might look something like the following. Notice how the `Variant` arguments are passed to the function as an array. The function must verify that the number of arguments passed to it equals the number of arguments it expects. It must also verify that the types of the argument(s) are as expected and be prepared to handle errors that the DC script programmer may have made when calling the function. At Naughty Dog, we wrote an argument iterator which allows us to extract and verify the arguments one by one in a convenient manner.

```
Variant DcGetObjectPos(U32 argCount, Variant* aArgs)
{
    // Argument iterator expecting at most 2 args.
    DcArgIterator args(argCount, aArgs, 2);

    // Set up a default return value.
    Variant result;
    result.SetAsVector(Vector(0.0f, 0.0f, 0.0f));

    // Use iterator to extract the args. It flags missing
    // or invalid arguments as errors automatically.
    StringId objectName = args.NextStringId();
    Point* pDefaultPos = args.NextPoint(kDcOptional);

    GameObject* pObject
        = GameObject::LookUpByName(objectName);
    if (pObject)
    {
        result.SetAsVector(pObject->GetPosition());
    }
    else
    {
        if (pDefaultPos)
        {
            result.SetAsVector(*pDefaultPos);
        }
        else
        {
            DcErrorMessage("get-object-pos: "
                           "Object '%s' not found.\n",
                           objectName.ToDebugString());
        }
```

```
        }

        return result;
    }
```

请注意，函数 StringId::ToDebugString() 执行反向查找，将字符串 ID 转换回其原始字符串。这需要游戏引擎维护某种数据库，将每个字符串 ID 映射到其原始字符串。在开发过程中，这样的数据库可以简化工作，但由于它占用大量内存，因此最终发布的产品中应该省略该数据库。（函数名 To Debug String() 提醒我们，从字符串 ID 到字符串的反向转换应该仅用于调试目的——游戏本身绝不能依赖此功能！）

16.9.5.2 游戏对象引用

脚本函数通常需要与游戏对象交互，而游戏对象本身可能部分或全部用引擎的原生语言实现。原生语言引用对象的机制（例如 C++ 中的指针或引用）在脚本语言中不一定有效。（例如，它可能根本不支持指针。）因此，我们需要找到一种可靠的方法让脚本代码引用游戏对象。

有多种方法可以实现这一点。一种方法是通过不透明的数字句柄引用脚本中的对象。脚本代码可以通过多种方式获取对象句柄。它可能由引擎传递一个句柄，或者它可能执行某种查询，例如询问玩家半径范围内所有游戏对象的句柄，或者查找与特定对象名称对应的句柄。然后，脚本可以通过调用本机函数并将对象的句柄作为参数传递来对游戏对象执行操作。在本机语言方面，句柄被转换回指向本机对象的指针，然后可以根据需要操作该对象。

数字句柄的优点是简单，并且应该很容易在任何支持整数数据的脚本语言中支持。然而，它们可能不够直观，难以操作。另一种选择是使用对象的名称（以字符串表示）作为句柄。与数字句柄技术相比，这有一些有趣的优势。首先，字符串易于理解且操作直观。它与游戏世界编辑器中的对象名称直接对应。此外，我们可以选择保留某些特殊的对象名称，并赋予它们“魔力”。


```
        }

        return result;
    }
```

Note that the function `StringId::ToDebugString()` performs a reverse look-up to convert a string id back to its original string. This requires the game engine to maintain some kind of database mapping each string id to its original string. During development such a database can make life much easier, but because it consumes a lot of memory, the database should be omitted from the final shipped product. (The function name `To`*`Debug`*`String()` reminds us that the reverse conversion from string id back to string should only be performed for debugging purposes—the game itself must *never* rely on this functionality!)

16.9.5.2 Game Object References

Script functions often need to interact with game objects, which themselves may be implemented partially or entirely in the engine's native language. The native language's mechanisms for referencing objects (e.g., pointers or references in C++) won't necessarily be valid in the scripting language. (It may not support pointers at all, for example.) Therefore, we need to come up with some reliable way for script code to reference game objects.

There are a number of ways to accomplish this. One approach is to refer to objects in script via opaque numeric *handles*. The script code can obtain object handles in various ways. It might be passed a handle by the engine, or it might perform some kind of query, such as asking for the handles of all game objects within a radius of the player or looking up the handle that corresponds to a particular object name. The script can then perform operations on the game object by calling native functions and passing the object's handle as an argument. On the native language side, the handle is converted back into a pointer to the native object, and then the object can be manipulated as appropriate.

Numeric handles have the benefit of simplicity and should be easy to support in any scripting language that supports integer data. However, they can be unintuitive and difficult to work with. Another alternative is to use the names of the objects, represented as strings, as our handles. This has some interesting benefits over the numeric handle technique. For one thing, strings are human-readable and intuitive to work with. There is a direct correspondence to the names of the objects in the game's world editor. In addition, we can choose to reserve certain special object names and give them "magic"

含义。例如，在顽皮狗的脚本语言中，保留名称“self”始终指当前正在运行的脚本所附加的对象。这允许游戏设计师编写脚本，将其附加到游戏中的某个对象，然后使用该脚本在该对象上播放动画，只需编写 (animate 'self name-of-animation) 即可。

当然，使用字符串作为对象句柄也有其缺陷。字符串通常比整数 ID 占用更多内存。而且由于字符串的长度各不相同，复制它们需要动态内存分配。字符串比较速度很慢。脚本程序员在输入游戏对象名称时容易出错，这可能会导致错误。此外，如果有人在游戏世界编辑器中更改了对象的名称，却忘记在脚本中更新该对象的名称，脚本代码可能会被破坏。

散列字符串 ID 通过将任何字符串（无论长度如何）转换为整数来解决大多数这些问题。理论上，散列字符串 ID 兼具两者的优点 - 用户可以像读取字符串一样读取它们，但它们具有整数的运行时性能特征。但是，要使其工作，您的脚本语言需要以某种方式支持散列字符串 ID。理想情况下，我们希望脚本编译器将字符串转换为散列 ID。这样，运行时代码根本不需要处理字符串，只需要处理散列 ID（可能除了出于调试目的 - 能够在调试器中看到与散列 ID 对应的字符串是很好的）。但是，这并非在所有脚本语言中都可行。另一种方法是允许用户在脚本中使用字符串，并在运行时（每当调用本机函数时）将它们转换为散列 ID。

顽皮狗的 DC 脚本语言利用 Scheme 编程语言中原生的“符号”概念来编码其字符串 ID。在 DC/Scheme 中，'foo（或者更详细地说，(quote foo)）对应于 C++ 中的字符串 ID SID("foo")。

16.9.5.3 在脚本中接收和处理事件

事件是大多数游戏引擎中普遍存在的通信机制。通过允许在脚本中编写事件处理函数，我们为自定义游戏的硬编码行为开辟了一条强大的途径。

事件通常会发送给单个对象，并在该对象的上下文中处理。因此，脚本事件处理程序需要以某种方式与对象关联。一些引擎为此使用游戏对象类型系统——脚本事件处理程序可以按对象类型注册。这允许不同类型的游戏对象以不同的方式响应同一事件，但确保每种类型的所有实例都以一致且统一的方式响应。事件处理程序函数本身可以是简单的脚本函数，或者如果脚本语言是

meanings. For example, in Naughty Dog's scripting language, the reserved name "self" always refers to the object to which the currently running script is attached. This allows game designers to write a script, attach it to an object in the game, and then use the script to play an animation on the object by simply writing `(animate 'self` *name-of-animation*`)`.

Using strings as object handles has its pitfalls, of course. Strings typically occupy more memory than integer ids. And because strings vary in length, dynamic memory allocation is required in order to copy them. String comparisons are slow. Script programmers are apt to make mistakes when typing the names of game objects, which can lead to bugs. In addition, script code can be broken if someone changes the name of an object in the game world editor but forgets to update the name of the object in script.

Hashed string ids overcome most of these problems by converting any strings (regardless of length) into an integer. In theory, hashed string ids enjoy the best of both worlds—they can be read by users just like strings, but they have the runtime performance characteristics of an integer. However, for this to work, your scripting language needs to support hashed string ids in some way. Ideally, we'd like the script compiler to convert our strings into hashed ids for us. That way, the runtime code doesn't have to deal with the strings at all, only the hashed ids (except possibly for debugging purposes—it's nice to be able to see the string corresponding to a hashed id in the debugger). However, this isn't always possible in all scripting languages. Another approach is to allow the user to use strings in script and convert them into hashed ids at runtime, whenever a native function is called.

Naughty Dog's DC scripting language leverages the concept of "symbols," which are native to the Scheme programming language, to encode its string ids. Writing `'foo`—or more verbosely, `(quote foo)`—in DC/Scheme corresponds to the string id `SID("foo")` in C++.

16.9.5.3 Receiving and Handling Events in Script

Events are a ubiquitous communication mechanism in most game engines. By permitting event handler functions to be written in script, we open up a powerful avenue for customizing the hard-coded behavior of our game.

Events are usually sent to individual objects and handled within the context of that object. Hence, scripted event handlers need to be associated with an object in some way. Some engines use the game object type system for this purpose—scripted event handlers can be registered on a per-object-type basis. This allows different types of game objects to respond in different ways to the same event but ensures that all instances of each type respond in a consistent and uniform way. The event handler functions themselves can be simple script functions, or they can be members of a class if the scripting language is

面向对象。无论哪种情况，事件处理程序通常都会传递一个指向事件所发送到的特定对象的句柄，就像 C++ 成员函数传递 this 指针一样。

在其他引擎中，脚本事件处理程序与单个对象实例相关联，而不是与对象类型相关联。在这种方法中，同一类型的不同实例可能对同一事件做出不同的响应。

当然，还有各种其他可能性。例如，在顽皮狗引擎（用于创作《神秘海域》和《最后生还者》系列）中，脚本本身就是对象。它们可以与单个游戏对象关联，可以附加到区域（用于触发游戏事件的凸体），也可以作为游戏世界中的独立对象存在。每个脚本可以具有多个状态（也就是说，在顽皮狗引擎中，脚本是有限状态机）。反过来，每个状态可以包含一个或多个事件处理程序代码块。当游戏对象接收到事件时，它可以选择使用原生 C++ 来处理该事件。它还会检查附加的脚本对象，如果找到，则将事件发送到该脚本的当前状态。如果该状态具有该事件的事件处理程序，则调用该处理程序。否则，脚本将忽略该事件。

16.9.5.4 发送事件

允许脚本处理引擎生成的游戏事件无疑是一项强大的功能。更强大的功能是能够从脚本代码生成事件并将其发送回引擎或其他脚本。

理想情况下，我们不仅希望能够从脚本发送预定义类型的事件，还希望能够在脚本中定义全新的事件类型。如果事件类型是字符串或字符串 ID，则实现这一点很简单。要定义新的事件类型，脚本程序员只需提出一个新的事件类型名称并将其输入到他或她的脚本代码中。这可以为脚本之间的相互通信提供高度灵活的方式。脚本 A 可以定义一种新的事件类型并将其发送给脚本 B。如果脚本 B 为这种类型的事件定义了一个事件处理程序，我们就为脚本 A 实现了一种与脚本 B“对话”的简单方法。在某些游戏引擎中，事件消息传递是脚本中唯一支持的对象间通信方式。这是一个优雅而强大且灵活的解决方案。

16.9.5.5 面向对象的脚本语言

有些脚本语言本质上是面向对象的。有些脚本语言不直接支持对象，但提供了可用于实现类和对象的机制。在许多引擎中，游戏玩法是通过对象实现的。

object-oriented. In either case, the event handler is typically passed a handle to the particular object to which the event was sent, much as C++ member functions are passed the `this` pointer.

In other engines, scripted event handlers are associated with individual object instances rather than with object types. In this approach, different instances of the same type might respond differently to the same event.

There are all sorts of other possibilities, of course. For example, in Naughty Dog's engine (used to create the *Uncharted* and *The Last of Us* series), scripts are objects in their own right. They can be associated with individual game objects, they can be attached to regions (convex volumes that are used to trigger game events), or they can exist as stand-alone objects in the game world. Each script can have multiple states (that is, scripts are finite state machines in the Naughty Dog engine). In turn, each state can have one or more event handler code blocks. When a game object receives an event, it has the option of handling the event in native C++. It also checks for an attached script object, and if one is found, the event is sent to that script's current state. If the state has an event handler for the event, it is called. Otherwise, the script simply ignores the event.

16.9.5.4 Sending Events

Allowing scripts to handle game events that are generated by the engine is certainly a powerful feature. Even more powerful is the ability to generate and send events from script code either back to the engine or to other scripts.

Ideally, we'd like to be able not only to send predefined types of events from script but to define entirely new event types in script. Implementing this is trivial if event types are strings or string ids. To define a new event type, the script programmer simply comes up with a new event type name and types it into his or her script code. This can be a highly flexible way for scripts to communicate with one another. Script A can define a new event type and send it to Script B. If Script B defines an event handler for this type of event, we've implemented a simple way for Script A to "talk" to Script B. In some game engines, event- or message-passing is the only supported means of inter-object communication in script. This can be an elegant yet powerful and flexible solution.

16.9.5.5 Object-Oriented Scripting Languages

Some scripting languages are inherently object-oriented. Others do not support objects directly but provide mechanisms that can be used to implement classes and objects. In many engines, gameplay is implemented via an object-

某种面向对象游戏对象模型。因此，在脚本中允许某种形式的面向对象编程也是有意义的。

在脚本中定义类

类实际上只是一堆数据和一些相关函数的组合。因此，任何允许定义新数据结构并提供某种方式存储和操作函数的脚本语言都可以用来实现类。例如，在 Lua 中，类可以由一个存储数据成员和成员函数的表构建而成。

脚本中的继承

面向对象语言不一定支持继承。但是，如果支持继承，它将非常有用，就像在 C++ 等原生编程语言中一样。

在游戏脚本语言中，有两种继承方式：从其他脚本类派生脚本类，以及从原生类派生脚本类。如果您的脚本语言是面向对象的，那么前者很可能是开箱即用的。然而，即使脚本语言支持继承，后者的实现也十分困难。问题在于如何弥合两种语言和两种底层对象模型之间的差距。我们不会在这里详细讨论如何实现这一点，因为具体实现必然取决于集成的两种语言。UnrealScript 是我遇到的唯一一种允许脚本类无缝地从原生类派生的脚本语言。

脚本中的组合/聚合

我们不需要依赖继承来扩展类的层次结构——我们也可以使用组合或聚合来达到类似的效果。因此，在脚本中，我们真正需要的是一种定义类并将这些类的实例与已在原生编程语言中定义的对象关联起来的方法。例如，一个游戏对象可以持有一个指向完全用脚本编写的可选组件的指针或引用。我们可以将某些关键功能委托给脚本组件（如果存在）。脚本组件可能包含一个 Update() 函数，每当游戏对象更新时都会调用该函数；脚本组件也可能被允许将其某些成员函数/方法注册为事件处理程序。当事件发送到游戏对象时，它会调用脚本组件上相应的事件处理程序，从而为脚本程序员提供了修改或扩展行为的机会。

oriented game object model of some kind. So it makes sense to permit some form of object-oriented programming in script as well.

Defining Classes in Scripts

A class is really just a bunch of data with some associated functions. So any scripting language that permits new data structures to be defined, and provides some way to store and manipulate functions, can be used to implement classes. For example, in Lua, a class can be built out of a table that stores data members and member functions.

Inheritance in Script

Object-oriented languages do not necessarily support inheritance. However, if this feature is available, it can be extremely useful, just as it is in native programming languages like C++.

In the context of game scripting languages, there are two kinds of inheritance: deriving scripted classes from other scripted classes and deriving scripted classes from native classes. If your scripting language is object-oriented, chances are the former is supported out of the box. However, the latter is tough to implement even if the scripting language supports inheritance. The problem is bridging the gap between two languages and two low-level object models. We won't get into the details of how this might be implemented here, as the implementation is bound to be specific to the pair of languages being integrated. UnrealScript is the only scripting language I've encountered that allows scripted classes to derive from native classes in a seamless way.

Composition/Aggregation in Script

We don't need to rely on inheritance to extend a hierarchy of classes—we can also use composition or aggregation to similar effect. In script, then, all we really need is a way to define classes and associate instances of those classes with objects that have been defined in the native programming language. For example, a game object could hold a pointer or reference to an optional component written entirely in script. We can delegate certain key functionality to the script component, if it exists. The script component might have an `Update()` function that is called whenever the game object is updated, and the scripted component might also be permitted to register some of its member functions/methods as event handlers. When an event is sent to the game object, it calls the appropriate event handler on the scripted component, thus giving the script programmer an opportunity to modify or extend the behavior

本机实现的游戏对象。

16.9.5.6 脚本有限状态机

游戏编程中的许多问题可以通过有限状态机 (FSM) 自然解决。因此，一些引擎将 FSM 的概念直接构建到核心游戏对象模型中。在这样的引擎中，每个游戏对象可以拥有一个或多个状态，并且这些状态（而不是游戏对象本身）包含更新函数、事件处理函数等等。简单的游戏对象可以通过定义单个状态来创建，但更复杂的游戏对象可以自由定义多个状态，每个状态具有不同的更新和事件处理行为。

如果您的引擎支持状态驱动的游戏对象模型，那么在脚本语言中提供有限状态机支持就非常有意义。当然，即使核心游戏对象模型本身不支持有限状态机，仍然可以通过在脚本端使用状态机来提供状态驱动的行为。任何编程语言都可以通过使用类实例来表示状态来实现有限状态机 (FSM)，但某些脚本语言专门为此目的提供了工具。面向对象的脚本语言可能提供自定义语法，允许一个类包含多个状态，或者提供工具帮助脚本程序员轻松地将状态对象聚合到一个中心对象中，然后以直接的方式将更新和事件处理函数委托给它。即使您的脚本语言不提供此类功能，您也可以始终采用一种方法来实现有限状态机 (FSM)，并在编写的每个脚本中遵循这些约定。

16.9.5.7 多线程脚本

能够并行执行多个脚本通常很有用，尤其是在当今高度并行化的硬件架构上。如果多个脚本可以同时运行，我们实际上是在脚本代码中提供了并行执行线程，就像大多数多任务操作系统提供的线程一样。当然，这些脚本实际上可能并非并行运行——如果它们都在单个 CPU 上运行，则该 CPU 必须轮流执行每个脚本。然而，从脚本程序员的角度来看，这种范式是一种并行编程。

大多数提供并行性的脚本系统都是通过协作式多任务来实现的。这意味着一个脚本会一直执行，直到它明确地让位于另一个脚本。这与抢占式多任务方法形成对比，在抢占式多任务方法中，任何脚本的执行都可以随时中断，以允许另一个脚本执行。

of the natively implemented game object.

16.9.5.6 Scripted Finite State Machines

Many problems in game programming can be solved naturally using finite state machines (FSMs). For this reason, some engines build the concept of FSMs right into the core game object model. In such engines, every game object can have one or more states, and it is the states—not the game object itself—that contain the update function, event handler functions and so on. Simple game objects can be created by defining a single state, but more complex game objects have the freedom to define multiple states, each with a different update and event-handling behavior.

If your engine supports a state-driven game object model, it makes a lot of sense to provide finite state machine support in the scripting language as well. And of course, even if the core game object model doesn't support finite state machines natively, one can still provide state-driven behavior by using a state machine on the script side. An FSM can be implemented in any programming language by using class instances to represent states, but some scripting languages provide tools especially for this purpose. An object-oriented scripting language might provide custom syntax that allows a class to contain multiple states, or it might provide tools that help the script programmer easily aggregate state objects together within a central hub object and then delegate the update and event-handling functions to it in a straightforward way. But even if your scripting language provides no such features, you can always adopt a methodology for implementing FSMs and follow those conventions in every script you write.

16.9.5.7 Multithreaded Scripts

It's often useful to be able to execute multiple scripts in parallel. This is especially true on today's highly parallelized hardware architectures. If multiple scripts can run at the same time, we are in effect providing *parallel threads of execution* in script code, much like the threads provided by most multitasking operating systems. Of course, the scripts may not actually run in parallel—if they are all running on a single CPU, the CPU must take turns executing each one. However, from the point of view of the script programmer, the paradigm is one of parallel programming.

Most scripting systems that provide parallelism do so via *cooperative multitasking*. This means that a script will execute until it explicitly yields to another script. This is in contrast with a *preemptive multitasking* approach, in which the execution of any script could be interrupted at any time to permit another

要执行的脚本。

在脚本中实现协作式多任务处理的一种简单方法是允许脚本明确进入睡眠状态，等待相关事件发生。脚本可以等待指定的秒数，也可以等到收到特定事件。它还可以等到另一个执行线程到达预定义的同步点。无论原因是什么，每当脚本进入睡眠状态时，它都会将自己添加到睡眠脚本线程列表中，并告知虚拟机它可以开始执行另一个符合条件的脚本。系统会跟踪唤醒每个睡眠脚本的条件——当其中一个条件成立时，等待该条件的脚本将被唤醒并允许继续执行。

为了了解其实际工作原理，我们来看一个多线程脚本的示例。此脚本管理两个角色和一扇门的动画。两个角色被指示走向门口——每个人到达门口所需的时间可能不同且不可预测。我们会让脚本的线程进入睡眠状态，同时等待角色到达门口。当他们都到达门口后，其中一个角色会打开门，即播放“开门”动画。请注意，我们不想将动画的持续时间硬编码到脚本本身中。这样，如果动画师更改动画，我们就不必返回并修改脚本。因此，我们会再次让线程进入睡眠状态，同时等待动画完成。下面显示了一个实现此操作的脚本，使用简单的类似 C 的伪代码语法。

```
procedure DoorCinematic()
{
    thread Guy1()
    {
        // Ask guy1 to walk to the door.
        CharacterWalkToPoint(guy1, doorPosition);

        // Go to sleep until he gets there.
        WaitUntil(CHARACTER_ARRIVAL);

        // OK, we're there. Tell the other threads
        // via a signal.
        RaiseSignal("Guy1Arrived");

        // Wait for the other guy to arrive as well.
        WaitUntil(SIGNAL, "Guy2Arrived");

        // Now tell guy1 to play the "open door"
        // animation.
        CharacterAnimate(guy1, "OpenDoor");
```


script to execute.

One simple approach to cooperative multitasking in script is to permit scripts to explicitly go to sleep, waiting for something relevant to happen. A script might wait for a specified number of seconds to elapse, or it might wait until a particular event is received. It might wait until another thread of execution has reached a predefined synchronization point. Whatever the reason, whenever a script goes to sleep, it puts itself on a list of sleeping script threads and tells the virtual machine that it can start executing another eligible script. The system keeps track of the conditions that will wake up each sleeping script—when one of these conditions becomes true, the script or scripts waiting on the condition are woken up and allowed to continue executing.

To see how this works in practice, let's look at an example of a multithreaded script. This script manages the animations of two characters and a door. The two characters are instructed to walk up to the door—each one might take a different, and unpredictable, amount of time to reach it. We'll put the script's threads to sleep while they wait for the characters to reach the door. Once they both arrive at the door, one of the two characters opens the door, which it does by playing an "open door" animation. Note that we don't want to hard-code the duration of the animation into the script itself. That way, if the animators change the animation, we won't have to go back and modify our script. So we'll put the threads to sleep again while the wait for the animation to complete. A script that accomplishes this is shown below, using a simple C-like pseudocode syntax.


```
procedure DoorCinematic()
{
    thread Guy1()
    {
        // Ask guy1 to walk to the door.
        CharacterWalkToPoint(guy1, doorPosition);

        // Go to sleep until he gets there.
        WaitUntil(CHARACTER_ARRIVAL);

        // OK, we're there. Tell the other threads
        // via a signal.
        RaiseSignal("Guy1Arrived");

        // Wait for the other guy to arrive as well.
        WaitUntil(SIGNAL, "Guy2Arrived");

        // Now tell guy1 to play the "open door"
        // animation.
        CharacterAnimate(guy1, "OpenDoor");
```

```
        WaitUntil(ANIMATION_DONE);

        // OK, the door is open. Tell the other threads.
        RaiseSignal("DoorOpen");

        // Now walk thru the door.
        CharacterWalkToPoint(guy1, beyondDoorPosition);
    }

    thread Guy2()
    {
        // Ask guy2 to walk to the door.
        CharacterWalkToPoint(guy2, doorPosition);

        // Go to sleep until he gets there.
        WaitUntil(CHARACTER_ARRIVAL);

        // OK, we're there. Tell the other threads
        // via a signal.
        RaiseSignal("Guy2Arrived");

        // Wait for the other guy to arrive as well.
        WaitUntil(SIGNAL, "Guy1Arrived");

        // Now wait until guy1 opens the door for me.
        WaitUntil(SIGNAL, "DoorOpen");

        // OK, the door is open. Now walk thru the door.
        CharacterWalkToPoint(guy2, beyondDoorPosition);
    }
}
```

上面，我们假设我们假设的脚本语言提供了一种简单的语法，用于在单个函数中定义执行线程。我们定义了两个线程，一个用于 Guy1，另一个用于 Guy2。

Guy1 的线程告诉角色走到门口，然后进入休眠状态等待角色到达。我们这里稍微有点不着边际，但我们可以想象一下，脚本语言神奇地允许线程进入休眠状态，等待游戏中的角色到达他被要求到达的目标点。实际上，这可以通过安排角色向脚本发送一个事件，然后在事件到达时唤醒线程来实现。

一旦 Guy1 到达门口，他的线程就会做两件事，这值得进一步解释。首先，它会发出一个名为“Guy1Arrived”的信号。其次，它会进入睡眠状态，等待另一个名为“Guy2Arrived”的信号。如果我们查看 Guy2 的线程，我们会看到类似的模式，只是顺序相反。这个模式


```
        WaitUntil(ANIMATION_DONE);

        // OK, the door is open. Tell the other threads.
        RaiseSignal("DoorOpen");

        // Now walk thru the door.
        CharacterWalkToPoint(guy1, beyondDoorPosition);
    }

    thread Guy2()
    {
        // Ask guy2 to walk to the door.
        CharacterWalkToPoint(guy2, doorPosition);

        // Go to sleep until he gets there.
        WaitUntil(CHARACTER_ARRIVAL);

        // OK, we're there. Tell the other threads
        // via a signal.
        RaiseSignal("Guy2Arrived");

        // Wait for the other guy to arrive as well.
        WaitUntil(SIGNAL, "Guy1Arrived");

        // Now wait until guy1 opens the door for me.
        WaitUntil(SIGNAL, "DoorOpen");

        // OK, the door is open. Now walk thru the door.
        CharacterWalkToPoint(guy2, beyondDoorPosition);
    }
}
```

In the above, we assume that our hypothetical scripting language provides a simple syntax for defining threads of execution within a single function. We define two threads, one for Guy1 and one for Guy2.

The thread for Guy1 tells the character to walk to the door and then goes to sleep waiting for his arrival. We're hand-waving a bit here, but let's imagine that the scripting language magically allows a thread to go to sleep, waiting until a character in the game arrives at a target point to which he was requested to walk. In reality, this might be implemented by arranging for the character to send an event back to the script and then waking the thread up when the event arrives.

Once Guy1 arrives at the door, his thread does two things that warrant further explanation. First, it raises a signal called "Guy1Arrived." Second, it goes to sleep waiting for another signal called "Guy2Arrived." If we look at the thread for Guy2, we see a similar pattern, only reversed. This pattern of

发出一个信号然后等待另一个信号是为了同步两个线程。

在我们假设的脚本语言中，信号只是一个带有名称的布尔标志。该标志初始为 false，但当线程调用 RaiseSignal(name) 时，指定标志的值将变为 true。其他线程可以进入休眠状态，等待特定的指定信号变为 true。当信号变为 true 时，休眠线程将被唤醒并继续执行。在此示例中，两个线程使用“Guy1Arrived”和“Guy2Arrived”信号相互同步。每个线程发出自己的信号，然后等待对方的信号。哪个信号先发出并不重要——只有当两个信号都发出时，两个线程才会被唤醒。当它们被唤醒时，它们将完全同步。图 16.24 展示了两种可能的情况：一种是 Guy1 先到达，另一种是 Guy2 先到达。如您所见，信号发出的顺序无关紧要，并且两个信号都发出后，线程总是同步的。

16.10 高级游戏流程

游戏对象模型提供了实现丰富有趣的游戏对象类型集合的基础，可以用来填充我们的游戏世界。然而，游戏对象模型本身只允许我们定义游戏世界中存在的对象类型以及它们各自的行为方式。它并没有提及玩家的目标、完成目标后会发生什么，以及失败后玩家的命运会如何。

为此，我们需要某种系统来控制高级游戏流程。这通常用有限状态机来实现。每个状态通常代表一个

图 16.24。两个示例展示了如何使用发出一个信号然后等待另一个信号的简单模式来同步一对脚本线程。

raising a signal and then waiting for another signal is to synchronize the two threads.

In our hypothetical scripting language, a *signal* is just a Boolean flag with a name. The flag starts out false, but when a thread calls `RaiseSignal(`*name*`)`, the named flag's value changes to true. Other threads can go to sleep, waiting for a particular named signal to become true. When it does, the sleeping thread(s) wake up and continue executing. In this example, the two threads are using the "Guy1Arrived" and "Guy2Arrived" signals to synchronize with one another. Each thread raises its signal and then waits for the other thread's signal. It does not matter which signal is raised first—only when both signals have been raised will the two threads wake up. And when they do, they will be in perfect synchronization. Two possible scenarios are illustrated in Figure 16.24, one in which Guy1 arrives first, the other in which Guy2 arrives first. As you can see, the order in which the signals are raised is irrelevant, and the threads always end up in sync after both signals have been raised.

16.10 High-Level Game Flow

A game object model provides the foundations upon which a rich and entertaining collection of game object types can be implemented with which to populate our game worlds. However, by itself, a game object model only permits us to define the kinds of objects that exist in our game world and how they behave individually. It says nothing of the player's objectives, what happens if he or she completes them, and what fate should befall the player if he or she fails.

For this, we need some kind of system to control high-level game flow. This is often implemented as a finite state machine. Each state usually represents a

Figure 16.24. Two examples showing how a simple pattern of raising one signal and then waiting on another can be used to synchronize a pair of script threads.

单人目标或遭遇，并与虚拟游戏世界中的特定场景相关联。随着玩家完成每个任务，状态机进入下一个状态，并向玩家呈现一组新的目标。状态机还定义了如果玩家未能完成必要的任务或目标时应该发生什么。通常，失败会让玩家回到当前状态的开始，以便他或她可以重试。有时在失败足够多之后，玩家将耗尽“生命”，并将被送回主菜单，在那里他或她可以选择玩新游戏。整个游戏的流程，从菜单到第一个“关卡”再到最后一个，都可以通过这个高级状态机来控制。

顽皮狗的《杰克与达斯特》、《神秘海域》和《最后生还者》系列游戏中使用的任务系统就是这种基于状态机的系统的一个示例。它允许线性状态序列（顽皮狗称之为任务）。它还允许并行任务，即一个任务分支成两个或多个并行任务，最终合并回主任务序列。这种并行任务特性使顽皮狗的任务图有别于常规状态机，因为状态机通常一次只能处于一种状态。

single player objective or encounter and is associated with a particular locale within the virtual game world. As the player completes each task, the state machine advances to the next state, and the player is presented with a new set of goals. The state machine also defines what should happen in the event of the player's failure to accomplish the necessary tasks or objectives. Often, failure sends the player back to the beginning of the current state, so he or she can try again. Sometimes after enough failures, the player has run out of "lives" and will be sent back to the main menu, where he or she can choose to play a new game. The flow of the entire game, from the menus to the first "level" to the last, can be controlled through this high-level state machine.

The task system used in Naughty Dog's *Jak and Daxter*, *Uncharted* and *The Last of Us* franchises is an example of such a state-machine-based system. It allows for linear sequences of states (called *tasks* at Naughty Dog). It also permits parallel tasks, where one task branches out into two or more parallel tasks, which eventually merge back into the main task sequence. This parallel task feature sets the Naughty Dog task graph apart from a regular state machine, since state machines typically can only be in one state at a time.

第五部分
结论

Part V
Conclusion

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

17
你的意思是还有更多吗?

恭喜！您已经完整地完成了游戏引擎架构之旅（希望一切顺利）。如果幸运的话，您已经学到了很多关于构成典型游戏引擎的主要组件的知识。当然，每一段旅程的结束都是另一段旅程的开始。本书涵盖的每个主题都还有许多值得学习的地方。随着技术和计算硬件的不断改进，游戏中将会出现更多的可能性，也会有更多引擎系统被发明来支持它们。而且，本书的重点是游戏引擎本身。我们甚至还没有开始讨论丰富的游戏编程世界，这个话题可以写成很多卷。

在以下简短的章节中，我将介绍一些本书中没有深入介绍的引擎和游戏系统，并为那些希望了解更多信息的人推荐一些资源。

17.1 一些我们未涉及的发动机系统

17.1.1 电影播放器

大多数游戏都包含一个电影播放器□□，用于播放预渲染的电影，也称为全动态视频 (FMV)。电影播放器□□的基本组件

17
You Mean There's More?

Congratulations! You've reached the end of your journey through the landscape of game engine architecture in one piece (and hopefully none the worse for wear). With any luck, you've learned a great deal about the major components that comprise a typical game engine. But of course, every journey's end is another's beginning. There's a great deal more to be learned about each and every topic covered within these pages. As technology and computing hardware continue to improve, more things will become possible in games—and more engine systems will be invented to support them. What's more, this book's focus was on the game engine itself. We haven't even begun to discuss the rich world of gameplay programming, a topic that could fill many more volumes.

In the following brief sections, I'll identify a few of the engine and gameplay systems we didn't have room to cover in any depth in this book, and I'll suggest some resources for those who wish to learn more about them.

17.1 Some Engine Systems We Didn't Cover

17.1.1 Movie Player

Most games include a movie player for displaying prerendered movies, also known as full-motion video (FMV). The basic components of the movie player

是流文件 I/O 系统的接口（参见第 7.1.3 节）、用于解码压缩视频流的编解码器，以及与音轨的音频播放系统的某种形式的同步。

目前有许多不同的视频编码标准和相应的编解码器，每种标准都适用于特定类型的应用。例如，视频 CD (VCD) 和 DVD 分别使用 MPEG-1 和 MPEG-2 (H.262) 编解码器。H.261 和 H.263 标准主要针对在线视频会议应用而设计。游戏通常使用 MPEG-4 第 2 部分（例如 DivX）、MPEG-4 第 10 部分 / H.264、Windows Media Video (WMV) 或 Bink Video（由 Rad Game Tools, Inc. 专为游戏设计的标准）等标准。有关视频编解码器的更多信息，请参阅 http://en.wikipedia.org/wiki/Video_codec 和 http://www.radgame tools.com/bnkmain.htm。

17.1.2 多人网络

虽然第四章中介绍的并发编程概念与多人游戏架构和分布式网络编程相关，但本书并未直接讨论这两个主题。有关多人游戏网络的深入探讨，请参阅[4]。

17.2 游戏系统

当然，游戏远不止引擎。在游戏玩法基础层（第16章讨论）之上，你会发现各种类型和特定游戏的玩法系统。这些系统将本书中描述的众多游戏引擎技术整合成一个紧密结合的整体，为游戏注入活力。

17.2.1 玩家机制

玩家机制当然是最重要的游戏系统。每个游戏类型都有其独特的玩家机制和玩法风格，而同一类型的每款游戏也都有其独特的设计。因此，玩家机制是一个庞大的课题。它涉及人机界面设备系统、运动模拟、碰撞检测、动画和音频的集成，更不用说与其他游戏系统的集成，例如游戏摄像机、武器、掩体、特殊移动机制（梯子、摆动绳索等）、载具系统、谜题机制等等。

显然，玩家机制和游戏本身一样千差万别，所以没有一个地方可以让你了解所有游戏机制。最好一次只研究一个游戏类型。玩游戏，并尝试进行逆向工程

are an interface to the streaming file I/O system (see Section 7.1.3), a codec to decode the compressed video stream, and some form of synchronization with the audio playback system for the sound track.

A number of different video encoding standards and corresponding codecs are available, each one suited to a particular type of application. For example, video CDs (VCD) and DVDs use MPEG-1 and MPEG-2 (H.262) codecs, respectively. The H.261 and H.263 standards are designed primarily for online video conferencing applications. Games often use standards like MPEG-4 part 2 (e.g., DivX), MPEG-4 Part 10 / H.264, Windows Media Video (WMV) or Bink Video (a standard designed specifically for games by Rad Game Tools, Inc.). See http://en.wikipedia.org/wiki/Video_codec and http://www.radgame tools.com/bnkmain.htm for more information on video codecs.

17.1.2 Multiplayer Networking

Although the concepts of concurrent programming covered in Chapter 4 are relevant to multiplayer game architecture and distributed network programming, this book doesn't address either topic directly. For an in-depth treatment of multiplayer networking, see [4].

17.2 Gameplay Systems

A game is of course much more than just its engine. On top of the gameplay foundation layer (discussed in Chapter 16), you'll find a rich assortment of genre- and game-specific gameplay systems. These systems tie the myriad game engine technologies described in this book together into a cohesive whole, breathing life into the game.

17.2.1 Player Mechanics

Player mechanics are of course the most important gameplay system. Each genre is defined by a general style of player mechanics and gameplay, and of course every game within a genre has its own specific designs. As such, player mechanics is a huge topic. It involves the integration of human interface device systems, motion simulation, collision detection, animation and audio, not to mention integration with other gameplay systems like the game camera, weapons, cover, specialized traversal mechanics (ladders, swinging ropes, etc.), vehicle systems, puzzle mechanics and so on.

Clearly, player mechanics are as varied as the games themselves, so there's no one place you can go to learn all about them. It's best to tackle this topic by studying a single genre at a time. Play games and try to reverse-engineer

他们的玩家机制。然后尝试自己实现它们！作为阅读的入门，你可以查看[9，第 4.11 节]，其中讨论了马里奥式平台游戏的玩家机制。

17.2.2 相机

游戏的摄像机系统几乎与玩家机制同等重要。事实上，摄像机可以成就或毁掉游戏体验。每种游戏类型往往都有其独特的摄像机控制风格，当然，同一类型的每款游戏的摄像机控制方式略有不同（有些甚至截然不同）。请参阅[8，第 4.3 节]，了解一些基本的游戏摄像机控制技巧。在接下来的段落中，我将简要概述 3D 游戏中一些最常见的摄像机类型，但请注意，这远非完整的列表。

• 观察相机。这种类型的相机围绕目标点旋转，并可以相对于该点移入或移出。

• 跟随摄像机。这种类型的摄像机在平台游戏、第三人称射击游戏和载具类游戏中很常见。它的作用类似于聚焦于玩家角色/虚拟形象/载具的观察摄像机，但其运动通常滞后于玩家。跟随摄像机还包含高级碰撞检测和规避逻辑，并为人类玩家提供一定程度的控制，使其能够控制摄像机相对于玩家虚拟形象的方向。

• 第一人称视角。当玩家角色在游戏世界中移动时，第一人称视角始终固定在角色的虚拟眼睛上。玩家通常可以通过鼠标或游戏手柄完全控制视角的指向。视角的注视方向也直接转化为玩家武器的瞄准方向，通常由屏幕底部的一组虚拟手臂和一把枪以及屏幕中央的十字线指示。

• RTS 摄像机。实时战略游戏和上帝之战游戏通常采用漂浮在地形上方的摄像机，以一定角度俯视。摄像机可以在地形上平移，但其俯仰和偏航通常不受玩家直接控制。

• 电影摄像机。大多数三维游戏至少有一些电影般的场景，在这些场景中，摄像机以更具电影感的方式在场景中飞来飞去，而不是被束缚在游戏中的某个物体上。

这些摄像机运动通常由动画师控制。

their player mechanics. Then try to implement them yourself! And as a very modest start on reading, you can check out [9, Section 4.11] for a discussion of Mario-style platformer player mechanics.

17.2.2 Cameras

A game's camera system is almost as important as the player mechanics. In fact, the camera can make or break the gameplay experience. Each genre tends to have its own camera control style, although of course every game within a particular genre does it a little bit differently (and some *very* differently). See [8, Section 4.3] for some basic game camera control techniques. In the following paragraphs, I'll briefly outline some of the most prevalent kinds of cameras in 3D games, but please note that this is far from a complete list.

- *Look-at cameras*. This type of camera rotates about a target point and can be moved in and out relative to this point.
- *Follow cameras*. This type of camera is prevalent in platformer, third-person shooter and vehicle-based games. It acts much like a look-at camera focused on the player character/avatar/vehicle, but its motion typically lags the player. A follow camera also includes advanced collision detection and avoidance logic and provides the human player with some degree of control over the camera's orientation relative to the player avatar.
- *First-person cameras*. As the player character moves about in the game world, a first-person camera remains affixed to the character's virtual eyes. The player typically has full control over the direction in which the camera should be pointed, either via mouse or joypad control. The look direction of the camera also translates directly into the aim direction of the player's weapon, which is typically indicated by a set of disembodied arms and a gun attached to the bottom of the screen, and a reticle at the center of the screen.
- *RTS cameras*. Real-time strategy and god games tend to employ a camera that floats above the terrain, looking down at an angle. The camera can be panned about over the terrain, but the pitch and yaw of the camera are usually not under direct player control.
- *Cinematic cameras*. Most three-dimensional games have at least some cinematic moments in which the camera flies about within the scene in a more filmic manner rather than being tethered to an object in the game. These camera movements are typically controlled by the animators.

17.2.3 人工智能

大多数角色扮演游戏的另一个主要组成部分是人工智能 (AI)。在最低层级，AI 系统通常基于诸如基本路径查找（通常使用著名的 A* 算法）、感知系统（视线、视锥、环境知识等）以及某种形式的记忆或知识等技术。

在这些基础之上，我们实现了角色控制逻辑。角色控制系统决定了如何让角色执行特定动作，例如移动、穿越特殊地形、使用武器、驾驶载具、寻找掩体等等。它通常涉及与引擎内部碰撞、物理和动画系统相关的复杂接口。角色控制将在 12.10 节中详细讨论。

在角色控制层之上，人工智能系统通常具有目标设定和决策逻辑，还可能具有情绪状态建模、群体行为（协调、侧翼攻击、人群和群集行为等），以及一些高级功能，例如从过去的错误中学习或适应不断变化的环境的能力。

当然，“人工智能”一词是游戏行业最大的误称之一。游戏AI与其说是真正模仿人类智能的尝试，不如说更像是一种障眼法。你的AI角色或许拥有各种复杂的内在情绪状态，并对游戏世界有着精准的感知。但如果玩家无法感知角色的动机，那么这一切都将毫无意义。

AI 编程是一个内容丰富的话题，本书当然没有对其进行详尽的阐述。更多信息，请参阅 [18]、[8 第 3 节]、[9 第 3 节] 和 [47 第 3 节]。另一个不错的起点是 Bungie 的 Chris Butcher 和 Jaime Griesemer 在 2002 年游戏开发者大会 (GDC 2002) 上发表的题为“智能的幻觉：《光晕》中 AI 与关卡设计的融合”的演讲 (http://bit.ly/1g7FbhD)。上网时，也可以搜索“游戏 AI 编程”。你会找到各种关于游戏 AI 的演讲、论文和书籍的链接。网站：http://aigamedev.com 和 http://www.gameai.com 也是很好的资源。

17.2.4 其他游戏系统

显然，游戏远不止玩家机制、摄像头和AI。有些游戏拥有可驾驶的车辆，配备特殊类型的武器，允许玩家借助动态物理模拟破坏环境，允许玩家创建自己的角色，构建自定义关卡，要求玩家解开谜题……当然，还有游戏类型和游戏特有功能的列表，以及所有专门的软件系统。

17.2.3 Artificial Intelligence

Another major component of most character-based games is *artificial intelligence* (AI). At its lowest level, an AI system is usually founded in technologies like basic path finding (which commonly makes use of the well-known A* algorithm), perception systems (line of sight, vision cones, knowledge of the environment, etc.) and some form of memory or knowledge.

On top of these foundations, character control logic is implemented. A character control system determines how to make the character perform specific actions like locomoting, navigating unusual terrain features, using weapons, driving vehicles, taking cover and so on. It typically involves complex interfaces to the collision, physics and animation systems within the engine. Character control is discussed in detail in Section 12.10.

Above the character control layer, an AI system typically has goal setting and decision making logic, and possibly also emotional state modeling, group behaviors (coordination, flanking, crowd and flocking behaviors, etc.), and perhaps some advanced features like an ability to learn from past mistakes or adapt to a changing environment.

Of course, the term “artificial intelligence” is one of the biggest misnomers around in the game industry. Game AI is always more of a smoke and mirrors job than an attempt at truly mimicking human intelligence. Your AI characters might have all sorts of complex internal emotional states and finely tuned perception of the game world. But if the player cannot *perceive* the characters' motivations, it's all for naught.

AI programming is a rich topic, and we certainly have not done it justice in this book. For more information, see [18], [8, Section 3], [9, Section 3] and [47, Section 3]. Another good starting point is the GDC 2002 talk entitled, “The Illusion of Intelligence: The Integration of AI and Level Design in Halo,” by Chris Butcher and Jaime Griesemer of Bungie (http://bit.ly/1g7FbhD). And while you're online, search for “game AI programming” too. You'll find all sorts of links to talks, papers and books on game AI. The websites http://aigamedev.com and http://www.gameai.com are great resources as well.

17.2.4 Other Gameplay Systems

Clearly there's a lot more to a game than just player mechanics, cameras and AI. Some games have drivable vehicles, implement specialized types of weaponry, allow the player to destroy the environment with the help of a dynamic physics simulation, let the player create his or her own characters, build custom levels, require the player to solve puzzles, …. Of course, the list of genre- and game-specific features, and all of the specialized software systems

实现这些目标的计划可以无限延续下去。游戏系统和游戏一样丰富多样。或许这就是你作为游戏程序员的下一段旅程的起点！

that implement them, could go on forever. Gameplay systems are as rich and varied as games are. Perhaps this is where your next journey as a game programmer will begin!

泰勒弗朗西斯 泰勒弗朗西斯集团 http://taylorandfrancis.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

参考书目

[1] Michael Abrash. Michael Abrash 的图形编程黑皮书（特别版）. 斯科茨代尔, AZ: Coriolis Group Books, 1997. （可在线访问 http://www.jagregory.com/abrash-black-book.）

[2] Tomas Akenine-Moller、Eric Haines 和 Naty Hoffman。《实时渲染》，第三版。马萨诸塞州韦尔斯利：AK Peters，2008 年。

[3] Andrei Alexandrescu. 现代 C++ 设计：泛型编程与设计模式应用. Reading, MA: Addison-Wesley, 2001.

[4] Grenville Armitage、Mark Claypool 和 Philip Branch。网络与在线游戏：理解和设计多人网络游戏。纽约，纽约州：
约翰威利父子公司，2006 年。

[5] James Arvo (编辑). Graphics Gems II. 圣地亚哥，加州：Academic Press，1991 年。

[6] David A. Bies 和 Colin H. Hansen.《工程噪声控制》，第四版。
纽约州纽约：CRC Press，2014 年。

[7] Grady Booch、Robert A. Maksimchuk、Michael W. Engel、Bobbi J. Young、Jim Conallen 和 Kelli A. Houston。《面向对象分析与设计及其应用》，第三版。马萨诸塞州雷丁：Addison-Wesley出版社，2007年。

[8] Mark DeLoura (编辑). 游戏编程精华. Hingham, MA: Charles River Media, 2000.

[9] Mark DeLoura (编辑). 游戏编程精华 2. Hingham, MA: Charles River Media, 2001.

Bibliography

[1] Michael Abrash. *Michael Abrash's Graphics Programming Black Book (Special Edition)*. Scottsdale, AZ: Coriolis Group Books, 1997. (Available online at http://www.jagregory.com/abrash-black-book.)

[2] Tomas Akenine-Moller, Eric Haines and Naty Hoffman. *Real-Time Rendering*, Third Edition. Wellesley, MA: A K Peters, 2008.

[3] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Reading, MA: Addison-Wesley, 2001.

[4] Grenville Armitage, Mark Claypool and Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. New York, NY: John Wiley and Sons, 2006.

[5] James Arvo (editor). *Graphics Gems II*. San Diego, CA: Academic Press, 1991.

[6] David A. Bies and Colin H. Hansen. *Engineering Noise Control*, Fourth Edition. New York, NY: CRC Press, 2014.

[7] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*, Third Edition. Reading, MA: Addison-Wesley, 2007.

[8] Mark DeLoura (editor). *Game Programming Gems*. Hingham, MA: Charles River Media, 2000.

[9] Mark DeLoura (editor). *Game Programming Gems 2*. Hingham, MA: Charles River Media, 2001.

[10] Philip Dutré、Kavita Bala 和 Philippe Bekaert。《高级全局照明》，第二版。马萨诸塞州韦尔斯利：AK Peters，2006 年。

[11] David H. Eberly. 3D游戏引擎设计：实时计算机图形的实用方法. 旧金山，加州：Morgan Kaufmann，2001年。

[12] David H. Eberly. 3D游戏引擎架构：利用Wild Magic构建实时应用. 旧金山，加州：Morgan Kaufmann，2005年。

[13] David H. Eberly. 游戏物理学. 旧金山, CA: Morgan Kaufmann, 2003.

[14] Christer Ericson. 实时碰撞检测. 旧金山, CA: Morgan Kaufmann, 2005.

[15] Randima Fernando（编辑）。《GPU精华：实时图形编程技术、技巧和窍门》。马萨诸塞州雷丁：Addison-Wesley出版社，2004年。

[16] James D. Foley、Andries van Dam、Steven K. Feiner 和 John F. Hughes。《计算机图形学：C 语言原理与实践》（第二版）。马萨诸塞州雷丁：AddisonWesley 出版社，1995 年出版。

[17] Grant R. Fowles 和 George L. Cassiday.《分析力学》，第七版。
加利福尼亚州太平洋丛林：布鲁克斯科尔，2005 年。

[18] John David Funge. 游戏和动画的人工智能：一种认知建模方法。
马萨诸塞州韦尔斯利：AK Peters，1999。

[19] Erich Gamma、Richard Helm、Ralph Johnson 和 John M. Vlissiddes。《设计模式：可复用面向对象软件的元素》。Reading, MA: AddisonWesley，1994 年。

[20] Andrew S. Glassner (编辑). Graphics Gems I. 旧金山，加州：Morgan Kaufmann，1990 年。

[21] Ananth Grama、Anshul Gupta、George Karypis 和 Vipin Kumar，《并行计算导论》，第二版，马萨诸塞州雷丁：Addison Wesley 出版社，2003 年。（可在线访问：http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf [原文如此]。）[22] Paul S. Heckbert（主编），《图形精粹 IV》，加州圣地亚哥：Academic Press，1994 年。[23] John L. Hennessey 和 David A. Patterson，《计算机架构：定量方法》，加州旧金山：Morgan Kaufmann 出版社，2011 年。

[24] Maurice Herlihy 和 Nir □□Shavit.《多处理器编程艺术》。加州旧金山：Morgan Kaufmann，2008 年。

[25] Roberto Ierusalimschy、Luiz Henrique de Figueiredo 和 Waldemar Celes。 Lua 5.1 参考手册。 Lua.org，2006。

[26] 罗伯托 · 耶鲁萨利姆斯奇。 Lua 编程，第二版。 Lua.org，2006。

[27] Isaac Victor Kerlow. 3D计算机动画与成像艺术（第二版）. 纽约：John Wiley and Sons，2000年。

[28] David Kirk (编辑). Graphics Gems III. 旧金山，加州：Morgan Kaufmann，1994 年。

[10] Philip Dutré, Kavita Bala and Philippe Bekaert. *Advanced Global Illumination*, Second Edition. Wellesley, MA: A K Peters, 2006.

[11] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. San Francisco, CA: Morgan Kaufmann, 2001.

[12] David H. Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. San Francisco, CA: Morgan Kaufmann, 2005.

[13] David H. Eberly. *Game Physics*. San Francisco, CA: Morgan Kaufmann, 2003.

[14] Christer Ericson. *Real-Time Collision Detection*. San Francisco, CA: Morgan Kaufmann, 2005.

[15] Randima Fernando (editor). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Reading, MA: Addison-Wesley, 2004.

[16] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes. *Computer Graphics: Principles and Practice in C*, Second Edition. Reading, MA: Addison-Wesley, 1995.

[17] Grant R. Fowles and George L. Cassiday. *Analytical Mechanics*, Seventh Edition. Pacific Grove, CA: Brooks Cole, 2005.

[18] John David Funge. *AI for Games and Animation: A Cognitive Modeling Approach*. Wellesley, MA: A K Peters, 1999.

[19] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissiddes. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[20] Andrew S. Glassner (editor). *Graphics Gems I*. San Francisco, CA: Morgan Kaufmann, 1990.

[21] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. *Introduction to Parallel Computing*, Second Edition. Reading, MA: Addison Wesley, 2003. (Available online at http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf [sic].)

[22] Paul S. Heckbert (editor). *Graphics Gems IV*. San Diego, CA: Academic Press, 1994.

[23] John L. Hennessey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 2011.

[24] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA: Morgan Kaufmann, 2008.

[25] Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.org, 2006.

[26] Roberto Ierusalimschy. *Programming in Lua*, Second Edition. Lua.org, 2006.

[27] Isaac Victor Kerlow. *The Art of 3-D Computer Animation and Imaging (Second Edition)*. New York, NY: John Wiley and Sons, 2000.

[28] David Kirk (editor). *Graphics Gems III*. San Francisco, CA: Morgan Kaufmann, 1994.

[29] Danny Kodicek，《游戏程序员的数学和物理学》，马萨诸塞州欣厄姆：查尔斯河媒体，2005 年。

[30] Raph Koster.《游戏设计的乐趣理论》。菲尼克斯，亚利桑那州：Paraglyph，2004 年。

[31] John Lakos，《大规模 C++ 软件设计》，马萨诸塞州雷丁：Addison-Wesley，1995 年出版。

[32] Eric Lengyel. 3D游戏编程与计算机图形学数学（第二版）. 马萨诸塞州欣厄姆: Charles River Media出版社, 2003年.

[33] Gary B. Little. Inside the Apple //e. Bowie, MD: Brady Communications Company, Inc., 1985. （可在线访问：http://www.apple2scans.net/files/InsidetheIIe）pdf。

[34] Tuoc V. Luong、James SH Lok、David J. Taylor 和 Kevin Driscoll。《国际化：面向全球市场的软件开发》。纽约：John Wiley & Sons，1995 年。

[35] Steve Maguire，《编写可靠代码：微软开发无 Bug C 程序的技术》。华盛顿州贝尔维尤：微软出版社，1993 年出版。

[36] Scott Meyers，《Effective C++：55 种改进程序和设计的具体方法》（第三版），马萨诸塞州雷丁：Addison-Wesley 出版社，2005 年出版。

[37] Scott Meyers，《更有效的 C++：改进程序和设计的 35 种新方法》。Reading, MA: Addison-Wesley，1996 年。

[38] Scott Meyers，《高效 STL：50 种改进标准模板库使用的具体方法》，马萨诸塞州雷丁：Addison-Wesley 出版社，2001 年出版。

[39] Ian Millington. 游戏物理引擎开发. 旧金山, CA: Morgan Kaufmann, 2007.

[40] Hubert Nguyen (编辑). GPU Gems 3. Reading, MA: Addison-Wesley, 2007.

[41] Alan V. Oppenheim 和 Alan S. Willsky. 信号与系统. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[42] Alan W. Paeth (编辑). Graphics Gems V. 旧金山，加州：Morgan Kaufmann，1995 年。

[43] C. Michael Pilato、Ben Collins-Sussman 和 Brian W. Fitzpatrick，《使用 Subversion 进行版本控制》，第二版。加州塞巴斯托波尔：O'Reilly Media，2008 年出版。（俗称“The Subversion Book”。可在线访问 http://svnbook）。red-bean.com。

[44] Matt Pharr（编辑）。《GPU精粹2：高性能图形和通用计算的编程技术》。马萨诸塞州雷丁：Addison-Wesley出版社，2005年。

[45] Richard Stevens 和 Dave Raybould. 游戏音频教程：互动游戏声音与音乐实用指南. 伯灵顿，马萨诸塞州：Focal Press，2011 年。

[46] Bjarne Stroustrup. C++编程语言，特别版（第三版）。Reading, MA: Addison-Wesley, 2000.

[47] Dante Treglia (编辑). 游戏编程精华 3. Hingham, MA: Charles River Media, 2002.

[29] Danny Kodicek. *Mathematics and Physics for Game Programmers.* Hingham, MA: Charles River Media, 2005.

[30] Raph Koster. *A Theory of Fun for Game Design.* Phoenix, AZ: Paraglyph, 2004.

[31] John Lakos. *Large-Scale C++ Software Design.* Reading, MA: Addison-Wesley, 1995.

[32] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*, Second Edition. Hingham, MA: Charles River Media, 2003.

[33] Gary B. Little. *Inside the Apple //e*. Bowie, MD: Brady Communications Company, Inc., 1985. (Available online at http://www.apple2scans.net/files/InsidetheIIe.pdf.)

[34] Tuoc V. Luong, James S. H. Lok, David J. Taylor and Kevin Driscoll. *Internationalization: Developing Software for Global Markets*. New York, NY: John Wiley & Sons, 1995.

[35] Steve Maguire. *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs.* Bellevue, WA: Microsoft Press, 1993.

[36] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Third Edition. Reading, MA: Addison-Wesley, 2005.

[37] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Reading, MA: Addison-Wesley, 1996.

[38] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Reading, MA: Addison-Wesley, 2001.

[39] Ian Millington. *Game Physics Engine Development.* San Francisco, CA: Morgan Kaufmann, 2007.

[40] Hubert Nguyen (editor). *GPU Gems 3*. Reading, MA: Addison-Wesley, 2007.

[41] Alan V. Oppenheim and Alan S. Willsky. *Signals and Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[42] Alan W. Paeth (editor). *Graphics Gems V*. San Francisco, CA: Morgan Kaufmann, 1995.

[43] C. Michael Pilato, Ben Collins-Sussman and Brian W. Fitzpatrick. *Version Control with Subversion*, Second Edition. Sebastopol, CA: O'Reilly Media, 2008. (Commonly known as "The Subversion Book." Available online at http://svnbook.red-bean.com.)

[44] Matt Pharr (editor). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Reading, MA: Addison-Wesley, 2005.

[45] Richard Stevens and Dave Raybould. *The Game Audio Tutorial: A Practical Guide to Sound and Music for Interactive Games*. Burlington, MA: Focal Press, 2011.

[46] Bjarne Stroustrup. *The C++ Programming Language*, Special Edition (Third Edition). Reading, MA: Addison-Wesley, 2000.

[47] Dante Treglia (editor). *Game Programming Gems 3.* Hingham, MA: Charles River Media, 2002.

[48] Gino van den Bergen. 交互式 3D 环境中的碰撞检测. 旧金山, CA: Morgan Kaufmann, 2003 年.

[49] Alan Watt. 3D计算机图形学, 第三版. Reading, MA: Addison Wesley, 1999.

[50] James Whitehead II, Bryan McLemore 和 Matthew Orlando. 《魔兽世界编程：魔兽世界插件创建指南与参考》。纽约，纽约州：
John Wiley & Sons，2008 年。

[51] Richard Williams. 动画师的生存指南. 英国伦敦：Faber & Faber，2002 年。

[48] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. San Francisco, CA: Morgan Kaufmann, 2003.

[49] Alan Watt. *3D Computer Graphics*, Third Edition. Reading, MA: Addison Wesley, 1999.

[50] James Whitehead II, Bryan McLemore and Matthew Orlando. *World of Warcraft Programming: A Guide and Reference for Creating WoW Addons*. New York, NY: John Wiley & Sons, 2008.

[51] Richard Williams. *The Animator's Survival Kit*. London, UK: Faber & Faber, 2002.

指数

#include， 148
_DEBUG， 82
__m128，参见单指令多指令数据
2 叶片，373 2D 声音，955 3 叶片，374 3D Studio Max，62，669 3D 声音，955 80/20 规则，99，216，317，608

A*算法，1164
AABB，参见边界框
ABA问题，297
ABI，参见应用程序二进制接口吸收

大气，918，923，956，958，964，980 光，633 声音，917 抽象工厂，参见设计模式 AC-3，参见音频，文件格式 加速处理单元，227 声学强度，915，961

声压，参见声音声学建模，956声学，920

ACP，参见资产调节管道获取围栏，参见内存排序语义
获取内存顺序，参见内存排序语义
动作状态机状态层，787，794转换，786 ADC，参见模数转换跨寄存器添加，参见单指令多数据
加法混合，769 寻址模式，参见 CPU
ADPCM，参见脉冲编码调制

高级 Linux 声音架构，994
高级矢量扩展，331 亲和力，参见线程聚合，111、1049、1051、1153 AI，参见人工智能

AIFF，查看音频、文件格式

Index

#include, 148
_DEBUG, 82
__m128, *see* single instruction multiple data
2-blade, 373
2D sound, 955
3-blade, 374
3D Studio Max, 62, 669
3D sound, 955
80/20 rule, 99, 216, 317, 608

A* algorithm, 1164
AABB, *see* bounding box
ABA problem, 297
ABI, *see* application binary interface
absorption
  atmospheric, 918, 923, 956, 958, 964, 980
  light, 633
  sound, 917
abstract factory, *see* design pattern
AC-3, *see* audio, file formats
accelerated processing unit, 227
acoustic intensity, 915, 961
acoustic pressure, *see* sound
acoustical modeling, 956
acoustics, 920
ACP, *see* asset conditioning pipeline
acquire fence, *see* memory ordering semantics
acquire memory order, *see* memory ordering semantics
action state machine
  state layer, 787, 794
  transition, 786
ADC, *see* analog-to-digital conversion
adding across a register, *see* single instruction multiple data
additive blending, 769
addressing mode, *see* CPU
ADPCM, *see* pulse-code modulation
Advanced Linux Sound Architecture, 994
advanced vector extensions, 331
affinity, *see* thread
aggregation, 111, 1049, 1051, 1153
AI, *see* artificial intelligence
AIFF, *see* audio, file formats

反照率图，640
代数，359
代数简化，参见编译器，
优化
别名，683，950 Alienbrain，请参阅版
本控制 对齐，请参阅内存分配 chunky
，451，514 删除运算符（C++），156
双缓冲，435 动态，155，156，426 堆
内存，156，511 看门人，另请参阅设计
模式，112 新运算符（C++），156 优
化，426 池，430，438，512 资源块分
配器，514 单帧分配器，43□□5 堆栈，112
基于堆栈的，427，438，512 静态，155
alpha，623，635 混合功能，676 测试
，676 ALSA，请参阅高级 Linux 声音
建筑学
Altivec，参见单指令多指令
数据
ALU，参见算术/逻辑单元 环境照明，
参见照明 环境光遮蔽，参见照明 高保
真立体声响复制，964 放大器，926，9
46，982 振幅，912，933，946，参见音
频 模数转换，925，948 解析几何，836
消声室，922 角频率，913，933 角动
量，872 角速度，867 动画，52，参见
动作状态
机器添加剂混合，769
连接点，800，808 混
合，755，763 相机，7
47
cel, 722 通道, 744, 7
47 剪辑, 734 约束, 8
06, 816 交叉淡入淡
出, 758, 759 差异剪
辑, 769 缓和曲线, 7
59 Endorphin, 43
欣快感, 43 事件触发器, 746
影片, 734, 736 平面加权平均
值, 788 浮点通道, 747 帧, 722,
737 全局时间轴, 739 Granny,
42, 507, 516, 783, 790 手绘, 722
Havok 动画, 42 空闲, 722 实
例化, 754 对象间注册, 808 关
节, 查看关节注视, 816 循环, 7
22, 734, 738 元通道, 746 变形
目标, 724 运动捕捉, 6 OrbisA
nim 库, 42 粒子, 711 每个顶点
, 724 相位, 738 播放速率, 739
后处理, 774 程序化, 775 重定
向, 748 刚性, 723 运行周期, 73
4 样本，737 骨骼，62
插座，查看动画，附加点同步，74
2 纹理，723 时间缩放，736，739
传统，722 过渡，758，759 更新，
1089 步行循环，734

albedo map, 640
algebra, 359
algebraic simplification, *see* compiler,
optimizations
aliasing, 683, 950
Alienbrain, *see* version control
alignment, *see* memory
allocation
chunky, 451, 514
delete operator (C++), 156
double-buffered, 435
dynamic, 155, 156, 426
heap memory, 156, 511
janitor, *see also* design pattern, 112
new operator (C++), 156
optimization, 426
pool, 430, 438, 512
resource chunk allocator, 514
single-frame allocator, 435
stack, 112
stack-based, 427, 438, 512
static, 155
alpha, 623, 635
blending function, 676
testing, 676
ALSA, *see* Advanced Linux Sound
Architecture
Altivec, *see* single instruction multiple
data
ALU, *see* arithmetic/logic unit
ambient lighting, *see* lighting
ambient occlusion, *see* lighting
Ambisonics, 964
amplifier, 926, 946, 982
amplitude, 912, 933, 946, *see also* audio
analog-to-digital conversion, 925, 948
analytical geometry, 836
anechoic chamber, 922
angular frequency, 913, 933
angular momentum, 872
angular velocity, 867
animation, 52, *see also* action state
machine
additive blending, 769
attach point, 800, 808
blending, 755, 763
camera, 747
cel, 722
channel, 744, 747
clip, 734
constraint, 806, 816
cross-fade, 758, 759
difference clip, 769
ease curve, 759
Endorphin, 43
Euphoria, 43
event trigger, 746
film, 734, 736
flat weighted average, 788
floating-point channel, 747
frame, 722, 737
global timeline, 739
Granny, 42, 507, 516, 783, 790
hand-drawn, 722
Havok Animation, 42
idle, 722
instancing, 754
interobject registration, 808
joint, *see* joint
look-at, 816
looping, 722, 734, 738
metachannel, 746
morph target, 724
motion capture, 6
OrbisAnim library, 42
particle, 711
per-vertex, 724
phase, 738
playback rate, 739
post-processing, 774
procedural, 775
retargeting, 748
rigid, 723
run cycle, 734
sample, 737
skeletal, 62
socket, *see* animation, attach point
synchronization, 742
texture, 723
time scaling, 736, 739
traditional, 722
transition, 758, 759
updating, 1089
walk cycle, 734

动画压缩，726，777 通道省略，77
8 基于曲线，783 键省略，782 量
化，778 采样频率，782 小波压缩
，783 各向异性，参见纹理，过滤
ANSI，462 反对换，372 抗锯齿，
48，683 API，参见应用程序编程

界面
应用程序二进制接口，176 应用程序编
程接口，40，
43, 975
近似分段线性，746，970 架构运行时
，38 存档文件，505 区域光，参见照
明 Argand 平面，934 参数（复数），9
34 算术/逻辑单元，166 结构数组，10
60 发音语音合成器，996 人工智能，5
8，538，597，822，

849, 901, 997, 1000, 1004,
1029, 1034, 1044, 1059, 1062,
1066, 1105, 1131, 1164
ASM，参见动作状态机汇编语言，105
断言，44，86，122，125-127，325编译
时，128，129无需锁定，325，1102静态
，128，129资产调节管道，59，61，499
，

501, 668, 671, 1036
关联，另请参阅缓存 数组，1139
属性，933 非对称多处理，228 异
步，请参阅多任务 原子，262，2
64，267，289 指令，292 操作，2
62，276，549，555

变量，314 ATRAC，参见音频
，文件格式附加点，参见动画
衰减

基于距离的, 956, 957 可听频带, 917 音频
，另请参阅声音, 54 振幅平移, 959 模拟,
942 拍频, 919 衰减, 921 密度, 921 漫反射
尾音, 920 扩散, 921 闪避, 989 早期反射, 9
20 排除, 968 文件格式, 951, 952 组, 989
实例限制, 990 插孔, 947 后期混响, 920
混音, 956, 962, 975, 980, 981, 988 阻塞, 96
8 遮挡, 968 平移, 956, 957, 975 恒定增益,
961 恒定功率, 961 聚焦, 963, 964 声像调
节器, 980 感知响度, 914 感知位置, 923
预延迟, 921 处理图, 976 渲染, 54, 911, 95
5 空间化, 956 更新, 1089 语音, 977 语音
窃取, 991 音频引擎, 975 Quake, 54

尖叫，54
虚幻，54
X3DAudio，993
XACT, 993
XAudio2, 54, 993 增强
现实, 27

animation compression, 726, 777
channel omission, 778
curve-based, 783
key omission, 782
quantization, 778
sampling frequency, 782
wavelet compression, 783
anisotropic, *see* texture, filtering
ANSI, 462
anti-commutative, 372
antialiasing, 48, 683
API, *see* application programming
interface
application binary interface, 176
application programming interface, 40,
43, 975
approximation
piecewise-linear, 746, 970
architecture
runtime, 38
archive file, 505
area light, *see* lighting
Argand plane, 934
argument (complex), 934
arithmetic/logic unit, 166
array of structs, 1060
articulatory speech synthesizer, 996
artificial intelligence, 58, 538, 597, 822,
849, 901, 997, 1000, 1004,
1029, 1034, 1044, 1059, 1062,
1066, 1105, 1131, 1164
ASM, *see* action state machine
assembly language, 105
assertion, 44, 86, 122, 125–127, 325
compile-time, 128, 129
lock-not-needed, 325, 1102
static, 128, 129
asset conditioning pipeline, 59, 61, 499,
501, 668, 671, 1036
associative, *see also* cache
array, 1139
property, 933
asymmetric multiprocessing, 228
asynchronous, *see* multitasking
atomic, 262, 264, 267, 289
instruction, 292
operation, 262, 276, 549, 555
variable, 314
ATRAC, *see* audio, file formats
attach point, *see* animation
attenuation
distance-based, 956, 957
audible frequency band, 917
audio, *see also* sound, 54
amplitude panning, 959
analog, 942
beating, 919
decay, 921
density, 921
diffuse tail, 920
diffusion, 921
ducking, 989
early reflections, 920
exclusion, 968
file formats, 951, 952
group, 989
instance limiting, 990
jack, 947
late reverberations, 920
mixing, 956, 962, 975, 980, 981, 988
obstruction, 968
occlusion, 968
pan, 956, 957, 975
constant gain, 961
constant power, 961
focus, 963, 964
pan pot, 980
perceived loudness, 914
perception of position, 923
pre-delay, 921
processing graph, 976
rendering, 54, 911, 955
spatialization, 956
updating, 1089
voice, 977
voice stealing, 991
audio engine, 975
Quake, 54
Scream, 54
Unreal, 54
X3DAudio, 993
XACT, 993
XAudio2, 54, 993
augmented reality, 27

辅助发送，979
平均的
　加权，765
AVX，请参阅高级矢量扩展
轴+角度旋转，404
方位角，959

带限，948，949
带宽，参见管道
银行，看到声音，银行
障碍
　编译器，303
　记忆，310
重心坐标，766
基数，另请参阅数字基数
基线，717
批量更新，1090
大逃杀，参见类型
殴打，见音频
bel，参见分贝
贝塞尔三角形，624
“大O”符号，445
大端字节序，参见 endian
双线性，见纹理，过滤
广告牌，49，711，714
二进制，参见数字基数，131
二叉堆，442
二叉搜索树，442
二进制信号量，276
二叉空间划分树，1018，参见
　树
Bink 视频，1162
生物力学模型，43
位深度，949
位运算符，176，346
双向量，373
漂白剂旁路，48
混合树，788、792 – 796、798、801、806
混合，另见动画
　音频，957，971
　舞台（图形），676
Blinn-Phong 照明模型，参见
　灯光
阻塞算法, 289
阻塞函数，245 – 247，251，268，
270, 273, 289, 1102, 1110, 1111 绽放，见
灯光
蓝图，请参阅虚幻引擎 4
蓝牙，562
波特图，939
Boost 库，40、45、253、254、447 – 449，
　1081
边界框轴对齐，411，831

　定向，411，832
边界球树，参见树
边界体积，688
分支依赖, 218
分支惩罚，218
分支预测，218，219
斗士，参见类型
BRDF，参见照明
断点，请参阅调试器
画笔几何，62，1018
BSP 树，参见树
BSS段，参见可执行文件
BSSRDF，见照明
气泡效应，1050
构建配置
　调试构建, 86
　开发构建, 86
　混合构建，86
　发布版本, 82
　船舶建造，86
构建规则, 502
子弹，824
子弹穿过纸张，见隧道
凹凸贴图，参见渲染
bus
　地址，174
　音频，947，978
　　模拟，983
　　数字，983
　　实施，983
　　延迟，984
　　主输出，982
　　预设，990
　　声音，978
　数据，174
忙等待，245，251，270，273，289，294，
　298, 328, 555
字节，132
字节码，1135，1145

auxiliary send, 979
average
　weighted, 765
AVX, *see* advanced vector extensions
axis+angle rotation, 404
azimuthal angle, 959

band-limited, 948, 949
bandwidth, *see* pipeline
bank, *see* sound, bank
barrier
　compiler, 303
　memory, 310
barycentric coordinates, 766
base, *see also* numeric base
baseline, 717
batched updating, 1090
battle royale, *see* genre
beating, *see* audio
bel, *see* decibel
Bézier triangle, 624
“big O” notation, 445
big-endian, *see* endian
bilinear, *see* texture, filtering
billboard, 49, 711, 714
binary, *see* numeric base, 131
binary heap, 442
binary search tree, 442
binary semaphore, 276
binary space partitioning tree, 1018, *see*
　tree
Bink Video, 1162
biomechanical models, 43
bit depth, 949
bitwise operator, 176, 346
bivector, 373
bleach bypass, 48
blend tree, 788, 792–796, 798, 801, 806
blending, *see also* animation
　audio, 957, 971
　stage (graphics), 676
Blinn-Phong lighting model, *see*
　lighting
blocking algorithm, 289
blocking function, 245–247, 251, 268,
　270, 273, 289, 1102, 1110, 1111
bloom, *see* lighting
Blueprints, *see* Unreal Engine 4
Bluetooth, 562
Bode plot, 939
Boost library, 40, 45, 253, 254, 447–449,
　1081
bounding box
　axis-aligned, 411, 831
　oriented, 411, 832
bounding sphere tree, *see* tree
bounding volume, 688
branch dependency, 218
branch penalty, 218
branch prediction, 218, 219
brawler, *see* genre
BRDF, *see* lighting
breakpoint, *see* debugger
brush geometry, 62, 1018
BSP tree, *see* tree
BSS segment, *see* executable file
BSSRDF, *see* lighting
bubble-up effect, 1050
build configuration
　debug build, 86
　development build, 86
　hybrid build, 86
　release build, 82
　ship build, 86
build rules, 502
Bullet, 824
bullet through paper, *see* tunneling
bump mapping, *see* rendering
bus
　address, 174
　audio, 947, 978
　　analog, 983
　　digital, 983
　　implementation, 983
　　latency, 984
　　master output, 982
　　preset, 990
　　voice, 978
　data, 174
busy-wait, 245, 251, 270, 273, 289, 294,
　298, 328, 555
byte, 132
byte code, 1135, 1145

C 标准库，43、89、412、465、486
– 488、539、589
C++，4
最佳实践，105
位运算符，176，346
班级，106
宣言，146-148，150
定义，146，148
删除，156
头文件，79
继承，106
初始化顺序，418
管理，489
多重继承，107
new，156
对象模型，1022
后增量，445
预增量，445
预处理器，79
私人，149
公众，149
源文件，79
标准库，40，89，114，269，
448
标准化，113
静态，149，157
翻译单元，79，144
用户定义文字，459
版本，113
虚拟继承，107
挥发性，参见挥发性
C4 引擎，参见墓碑引擎
C#、34、531、1063
代表，1115
反射，1064，1136
缓存，191
相干性，197，630，1092
相干域，308
一致性协议，307
抄回，196
直接映射，195
驱逐，195
友好的设计，1060
命中，191
命中率，196
线，192
MESI 协议，307
小姐，191，198
MOESI 协议，307
多层次，196
更换政策，196
集合关联性，195
回写，196
直写，196
调用堆栈，92，153，610
堆栈框架，153
回调函数，489，492，530
相机，47，622，655，1163
调试，605
成像矩形，655
Camtasia，607
笛卡尔坐标，360
德军总部3D，31，625
焦散，参见照明
CCD，参见连续碰撞检测
CD，参见光盘
赛璐珞动画，722
蜂窝宽带引擎，228
质心，857
中央仲裁者，285
中央处理器，参见 CPU
Cg，672
CgFX，682
责任链，1122
钱德拉-米斯拉，285
基础变更，388
频道，看动画
角色对话，997
字符集
ANSI，462
Unicode，462
作弊，606
循环等待，282
班级，106，156
聚合，1049，1051
构图，1049，1051
构造函数，520
耦合，参见耦合
图表，107
层次结构，1046
气泡效应，1050 单片
，1046
虚幻，1046
例如，106，156

C standard library, 43, 89, 412, 465,
486–488, 539, 589
C++, 4
best practices, 105
bitwise operator, 176, 346
class, 106
declaration, 146–148, 150
definition, 146, 148
delete, 156
header file, 79
inheritance, 106
initialization order, 418
managed, 489
multiple inheritance, 107
new, 156
object model, 1022
postincrement, 445
preincrement, 445
preprocessor, 79
private, 149
public, 149
source file, 79
standard library, 40, 89, 114, 269,
448
standardization, 113
static, 149, 157
translation unit, 79, 144
user-defined literals, 459
versions of, 113
virtual inheritance, 107
volatile, *see* volatile
C4 Engine, *see* Tombstone Engine
C#, 34, 531, 1063
delegate, 1115
reflection, 1064, 1136
cache, 191
coherency, 197, 630, 1092
coherency domain, 308
coherency protocol, 307
copy-back, 196
direct-mapped, 195
eviction, 195
friendly design, 1060
hit, 191
hit rate, 196
line, 192
MESI protocol, 307
miss, 191, 198
MOESI protocol, 307
multilevel, 196
replacement policy, 196
set associativity, 195
write-back, 196
write-through, 196
call stack, 92, 153, 610
stack frame, 153
callback function, 489, 492, 530
camera, 47, 622, 655, 1163
debug, 605
imaging rectangle, 655
Camtasia, 607
Cartesian coordinates, 360
Castle Wolfenstein 3D, 31, 625
caustics, *see* lighting
CCD, *see* continuous collision detection
CD, *see* compact disc
cel animation, 722
cell broadband engine, 228
center of mass, 857
central arbitor, 285
central processing unit, *see* CPU
Cg, 672
CgFX, 682
Chain of Responsibility, 1122
Chandra-Misra, 285
change of basis, 388
channel, *see* animation
character dialog, 997
character set
ANSI, 462
Unicode, 462
cheats, 606
circular wait, 282
class, 106, 156
aggregation, 1049, 1051
composition, 1049, 1051
constructor, 520
coupling, *see* coupling
diagram, 107
hierarchy, 1046
bubble-up effect, 1050
monolithic, 1046
Unreal, 1046
instance, 106, 156

内存布局，158 混合，107，1049 打包，159 纯虚函数，163 反射，1064 脚本化，1153 虚函数，162 经典力学，854 ClearCase，参见版本控制 剪辑，参见动画，声音 剪辑，411，659，674 剪辑平面，46 时钟 全局，741 本地，741 封闭哈希表，参见容器布料

渲染, 649, 670 模拟, 674, 818, 909 云计算, 225 云, 714

CLR，请参阅公共语言运行时
CM，参见重心 代码重用，38 代码段，参见可执行文件 编解码器，979 视频，1162 编码标准，118 恢复系数，877 科夫曼条件，282

COLLADA，682 集合，参见容器 共线向量，369 碰撞，另请参阅散列接触，830，849 检测，42，51，52，207，359，384，411，527，595，823，825 形状，829

扫描和修剪算法，847 更新，1089 颜色，622，633 通道，634 log-LUV，634，702 模型，634 RGB，634 空间，634 光谱，633

着色，719 命令行参数，473，477 公共语言运行时，34，489 交换律，368，372，933 光盘，948 比较交换指令，295 编译时断言，参见断言编译语言，1135 编译器，78

构建配置，81 调试信息，83 GNU 编译器，83 优化，81、84，另请参阅链接器 代数简化，85 代码内联，85 编译时与链接时，85 常量折叠，85 常量传播，85 死代码消除，85 指令重新排序，85 局部与全局，84 循环展开，85 运算符强度降低，85 窥视孔，84 配置文件引导，85 优化，316 项目配置教程，88 项目文件，80 解决方案文件，80 未解析的符号错误，144 警告，81 编译器屏障，请参阅屏障 编译器内在函数，294、295、333 复指数，934 复指令集，223 复数，934 乘法，935 旋转，935

合成，111，1049，1051，1052，1153压缩，另见动画
无损压缩，951

量化, 564, 778, 948, 949 纹理, 642 压缩器, 983 计算着色器, 348, 350, 351, 672 并发, 204, 205, 256, 544

memory layout, 158
mix-in, 107, 1049
packing, 159
pure virtual function, 163
reflection, 1064
scripted, 1153
virtual function, 162
classical mechanics, 854
ClearCase, *see* version control
clip, *see* animation, sound
clipping, 411, 659, 674
clipping plane, 46
clock
global, 741
local, 741
closed hash table, *see* container
cloth
rendering, 649, 670
simulation, 674, 818, 909
cloud computing, 225
clouds, 714
CLR, *see* Common Language Runtime
CM, *see* center of mass
code reuse, 38
code segment, *see* executable file
codec, 979
video, 1162
coding standards, 118
coefficient of restitution, 877
Coffman conditions, 282
COLLADA, 682
collection, *see* container
collinear vectors, 369
collision, *see also* hashing
contact, 830, 849
detection, 42, 51, 52, 207, 359, 384, 411, 527, 595, 823, 825
shape, 829
sweep and prune algorithm, 847
updating, 1089
color, 622, 633
channel, 634
log-LUV, 634, 702
model, 634
RGB, 634
space, 634
spectral, 633
colorization, 719
command line arguments, 473, 477
Common Language Runtime, 34, 489
commutative property, 368, 372, 933
compact disc, 948
compare-and-swap instruction, 295
compile-time assertion, *see* assertion
compiled language, 1135
compiler, 78
build configurations, 81
debugging information, 83
GNU compiler, 83
optimizations, 81, 84, *see also* linker
algebraic simplification, 85
code inlining, 85
compile-time vs. link-time, 85
constant folding, 85
constant propagation, 85
dead code elimination, 85
instruction reordering, 85
local vs. global, 84
loop unrolling, 85
operator strength reduction, 85
peep-hole, 84
profile-guided, 85
optimizing, 316
project configuration tutorial, 88
project files, 80
solution files, 80
unresolved symbol error, 144
warnings, 81
compiler barrier, *see* barrier
compiler intrinsic, 294, 295, 333
complex exponential, 934
complex instruction set, 223
complex number, 934
multiplication, 935
rotation, 935
composition, 111, 1049, 1051, 1052, 1153
compression, *see also* animation compression
lossless, 951
quantization, 564, 778, 948, 949
texture, 642
compressor, 983
compute shader, 348, 350, 351, 672
concurrency, 204, 205, 256, 544

消息传递，257
监视器，288
进步，289
共享内存，204，257
条件变量，273，278，548，555
配置文件, 471
配置空间，776
锥形声源，918
共识问题，300
保守派进步，845
一致性模型，请参阅以数据为中心的一致性模型
安慰
游戏平台，8
游戏中，50，604
常量折叠，参见编译器、优化

恒定功率平移定律，参见音频
常量传播，参见编译器，优化
常量寄存器，参见 GPU
约束，407，806，816，854，879，883
消耗内存顺序，查看内存排序语义
接触，参见碰撞
容器，40，441
数组，441
二叉堆，442
二叉搜索树，442，452
建筑定制，447
以及大约442
字典，442，452，508，1008，1065, 1141
动态数组，442，451
图表，443
哈希表，442，452
开放与关闭，452
列表，41，442，1141
地图，442
优先级队列，442
队列，442
设置，443
堆栈，442
树，442
矢量（STL），41，442
争论，287
上下文切换，553
连续性 C0、C1、C2、758
动议，758
连续碰撞检测，845
连续时间信号，参见信号
控制依赖性，218
控制器，参见人机接口设备
收敛，863
对话，1002
行动，1009
分支，1004
上下文相关，1009
标准，1008
独家，1004
规则，1008 说话者和听众，1007
凸多面体，412
卷积，719，930，932，933
合作多任务，1140，1154
坐标系，384
笛卡尔，360
圆柱形，360
层次结构，388
齐次剪辑空间，659
左撇子，361
光空间，655，703，704
模型空间，385，630，751
右撇子，361
屏幕空间，663，716
球形，360
切线空间，635
纹理空间，641
视野空间，387，656
世界空间，386，631
写时复制，457
回写缓存, 196
核心，参见 CPU
协程，204，554，1140
耦合，38，46，53，66，144，1020，1052，1095
CPU，166
寻址模式, 178, 180
核心，226，353
依赖 CPU 的游戏，536
微操作，172
乱序执行，171，217，222，224，225，239，291，

message passing, 257
monitor, 288
progress, 289
shared memory, 204, 257
condition variable, 273, 278, 548, 555
configuration file, 471
configuration space, 776
conical sound source, 918
consensus problem, 300
conservative advancement, 845
consistency model, *see* data-centric consistency model
console
gaming platform, 8
in-game, 50, 604
constant folding, *see* compiler, optimizations
constant power pan law, *see* audio
constant propagation, *see* compiler, optimizations
constant register, *see* GPU
constraint, 407, 806, 816, 854, 879, 883
consume memory order, *see* memory ordering semantics
contact, *see* collision
container, 40, 441
array, 441
binary heap, 442
binary search tree, 442, 452
building custom, 447
deque, 442
dictionary, 442, 452, 508, 1008, 1065, 1141
dynamic array, 442, 451
graph, 443
hash table, 442, 452
open & closed, 452
list, 41, 442, 1141
map, 442
priority queue, 442
queue, 442
set, 443
stack, 442
tree, 442
vector (STL), 41, 442
contention, 287
context switch, 553
continuity
C0, C1, C2, 758
motion, 758
continuous collision detection, 845
continuous-time signal, *see* signal
control dependency, 218
controller, *see* human interface device
convergence, 863
conversation, 1002
action, 1009
branching, 1004
context-sensitive, 1009
criterion, 1008
exclusivity, 1004
rule, 1008
speaker and listener, 1007
convex polyhedron, 412
convolution, 719, 930, 932, 933
cooperative multitasking, 1140, 1154
coordinate system, 384
Cartesian, 360
cylindrical, 360
hierarchy, 388
homogeneous clip space, 659
left-handed, 361
light space, 655, 703, 704
model space, 385, 630, 751
right-handed, 361
screen space, 663, 716
spherical, 360
tangent space, 635
texture space, 641
view space, 387, 656
world space, 386, 631
copy on write, 457
copy-back cache, 196
core, *see* CPU
coroutine, 204, 554, 1140
coupling, 38, 46, 53, 66, 144, 1020, 1052, 1095
CPU, 166
addressing mode, 178, 180
core, 226, 353
CPU-dependent game, 536
micro-operations, 172
out-of-order execution, 171, 217, 222, 224, 225, 239, 291,

300 – 302, 304, 309, 311, 315, 355
流水线，171，另请参阅流水线，211，213，217，221，225，445 寄存器，153 推测执行，239 阶段，212，另请参阅 GPU，阶段 超标量，211，221 利用率，1105 非常长的指令字，211，224

坠机报告，594
起严重事故
操作, 262, 264, 267, 289, 302, 303, 325, 327 比赛, 258 部分, 270 叉积，参见矢量 交叉淡入淡出, 758, 759, 786, 1001 人群建模, 996 水晶空间, 36

CSV 文件，467，614 立方体贴图，700 提示，参见声音剔除，15，47 反门户，689 背面，627 视锥，688，参见视锥遮挡，15，18，47，688 门户，15，689 潜在可见集，688 可见性确定，688 曲线，759 cvar，474 CVS，参见版本控制 圆柱坐标，360

D-cache，参见数据缓存
DAC，参见数模转换
DAG，参见有向无环图 阻尼，860 缓冲器，参见阻尼 数据缓存，196 数据定义语言，1135 数据依赖性，参见依赖性 数据并行性，参见并行性 数据路径，1132
数据竞争，205，259，281 数据段，请参阅可执行文件 数据类型，138，146 以数据为中心的一致性模型，266 数据驱动，12，1024 成本，1024 DCC，请参阅数字内容创建应用程序 死代码消除，请参阅编译器，

优化
死锁，275，281 调试构建，请参阅构建配置 调试绘图，50，595 调试器，50，另请参阅调用堆栈断点，92，95 – 97 调试优化构建，97 单步，92 监视窗口，93 贴花，48，712 衰减，请参阅音频分贝，914，915 十进制，请参阅数字基声明，请参阅 C++ 声明性语言，1136 解码，请参阅 CPU，阶段分解，请参阅任务分解深拷贝，1127 延迟渲染，709 定义，请参阅 C++ 可变形体，825，903，909 并行度，1105 自由度，406，856，883 Delaunay 三角剖分，766 延迟槽，215，225 委托（C#），1115 删除，请参阅C++ 解调，951 非规范化，参见浮点密度，参见音频依赖项，214，222 循环，38 数据，215 引擎子系统，1093 游戏对象，1093 线程，参见线程依赖关系图，1105 深度缓冲区，664，704 阴影映射，704

300–302, 304, 309, 311, 315, 355
pipelined, 171, *see also* pipeline, 211, 213, 217, 221, 225, 445
register, 153
speculative execution, 239
stage, 212, *see also* GPU, stage
superscalar, 211, 221
utilization, 1105
very long instruction word, 211, 224
crash report, 594
critical
operation, 262, 264, 267, 289, 302, 303, 325, 327
race, 258
section, 270
cross product, *see* vector
cross-fade, 758, 759, 786, 1001
crowd modeling, 996
Crystal Space, 36
CSV file, 467, 614
cube map, 700
cue, *see* sound
culling, 15, 47
antiportal, 689
back-face, 627
frustum, 688, *see* frustum
occlusion, 15, 18, 47, 688
portal, 15, 689
potentially visible set, 688
visibility determination, 688
curve, 759
cvar, 474
CVS, *see* version control
cylindrical coordinates, 360

D-cache, *see* data cache
DAC, *see* digital-to-analog conversion
DAG, *see* directed acyclic graph
damping, 860
dashpot, *see* damping
data cache, 196
data definition language, 1135
data dependency, *see* dependencies
data parallelism, *see* parallelism
data pathway, 1132
data race, 205, 259, 281
data segment, *see* executable file
data type, 138, 146
data-centric consistency model, 266
data-driven, 12, 1024
cost of, 1024
DCC, *see* digital content creation app
dead code elimination, *see* compiler, optimizations
deadlock, 275, 281
debug build, *see* build configuration
debug drawing, 50, 595
debugger, 50, *see also* call stack
breakpoint, 92, 95–97
debugging optimized builds, 97
single-step, 92
watch window, 93
decal, 48, 712
decay, *see* audio
decibel, 914, 915
decimal, *see* numeric base
declaration, *see* C++
declarative language, 1136
decode, *see* CPU, stage
decomposition, *see* task decomposition
deep copy, 1127
deferred rendering, 709
definition, *see* C++
deformable body, 825, 903, 909
degree of parallelism, 1105
degrees of freedom, 406, 856, 883
Delaunay triangulation, 766
delay slot, 215, 225
delegate (C#), 1115
`delete`, *see* C++
demodulation, 951
denormalized, *see* floating-point
density, *see* audio
dependencies, 214, 222
cycle, 38
data, 215
engine subsystems, 1093
game objects, 1093
thread, *see* thread
dependency graph, 1105
depth buffer, 664, 704
shadow mapping, 704

测试, 676, 716 z 测试, 675, 692, 693
景深模糊, 719 双端队列, 442 向量导数, 858 设计模式, 111 抽象工厂, 111, 1042 责任链, 1122 命令, 1117 工厂, 1065 迭代器, 111, 118, 430, 443, 445, 1149 看门人, 111, 112, 320, 327 RAII, 111, 125, 320, 327 单例, 111, 418 可破坏对象, 903, 1018 开发构建，请参阅构建

配置
开发套件，461，616 设备驱动程序，975 对话框操作，1009 对话框系统，997 钻石继承问题，1049 字典，参见容器差异剪辑，769 衍射，参见照明，917，970 漫反射

照明，参见照明尾部，920 纹理贴图，640 扩散，参见音频数字内容创作应用程序，59, 62, 501,

669
数字分子物质，825 数字信号处理，参见信号
加工
数模转换，951 哲学家就餐，285 直接照明，参见照明 直接内存访问控制器，984 有向无环图，61，443，1109 有向图，1109 方向向量，365 定向光，参见照明 定向声源，918 DirectX，41，46，91

透视投影矩阵，660
视图空间，387 狄利克雷条件，938 拆卸，98 离散时间信号，见信号盘操作系统，233 调度，224 位移映射，见渲染显示设备，622 距离场，见有符号距离场基于距离的衰减，见

衰减
分布式计算，229 分布式共享内存，257 分配属性，368，372，933 除以 w，381 DivX，1162

DMAC，参见直接内存访问控制器
DMM，参见数字分子物质
DOC，参见并行度
DOF，参见自由度、深度场地
杜比数字，查看音频、文件格式
毁灭战士，11，31，625，1025
DOP，参见并行度
多普勒效应，922、956、957、973 点积，参见矢量双调度，842 DRC，参见动态范围压缩 干声，参见声音 DSP，参见信号处理

DTS，请参阅音频，文件格式，请参阅环绕声
对偶性，940 鸭子类型，1141 闪避，参见音频 动态分配，参见分配 动态数组，442 动态链接库，80 动态范围压缩，983 动力学（物理），854

早期反射，920 缓和曲线，759 回声，920

Edge，请参阅 PlayStation Edge 库
效果文件，请参阅着色器

testing, 676, 716
z-testing, 675, 692, 693
depth of field blur, 719
deque, 442
derivative of a vector, 858
design pattern, 111
abstract factory, 111, 1042
Chain of Responsibility, 1122
command, 1117
factory, 1065
iterator, 111, 118, 430, 443, 445, 1149
janitor, 111, 112, 320, 327
RAII, 111, 125, 320, 327
singleton, 111, 418
destructible objects, 903, 1018
development build, *see* build configuration
development kit, 461, 616
device driver, 975
dialog action, 1009
dialog system, 997
diamond inheritance problem, 1049
dictionary, *see* container
difference clip, 769
diffraction, *see* lighting, 917, 970
diffuse
lighting, *see* lighting
tail, 920
texture map, 640
diffusion, *see* audio
digital content creation app, 59, 62, 501, 669
Digital Molecular Matter, 825
digital signal processing, *see* signal processing
digital-to-analog conversion, 951
dining philosophers, 285
direct lighting, *see* lighting
direct memory access controller, 984
directed acyclic graph, 61, 443, 1109
directed graph, 1109
direction vector, 365
directional light, *see* lighting
directional sound source, 918
DirectX, 41, 46, 91
perspective projection matrix, 660
view space, 387
Dirichlet conditions, 938
disassembly, 98
discrete-time signal, *see* signal
disk operating systems, 233
dispatching, 224
displacement mapping, *see* rendering
display device, 622
distance field, *see* signed distance field
distance-based attenuation, *see* attenuation
distributed computing, 229
distributed shared memory, 257
distributive property, 368, 372, 933
division by *w*, 381
DivX, 1162
DMAC, *see* direct memory access controller
DMM, *see* Digital Molecular Matter
DOC, *see* degree of parallelism
DOF, *see* degrees of freedom, depth of field
Dolby Digital, *see* audio, file formats
Doom, 11, 31, 625, 1025
DOP, *see* degree of parallelism
Doppler effect, 922, 956, 957, 973
dot product, *see* vector
double dispatch, 842
DRC, *see* dynamic range compression
dry sound, *see* sound
DSP, *see* signal processing
DTS, *see* audio, file formats, *see* surround sound
duality, 940
duck typing, 1141
ducking, *see* audio
dynamic allocation, *see* allocation
dynamic array, 442
dynamic link library, 80
dynamic range compression, 983
dynamics (physics), 854

early reflections, 920
ease curve, 759
echo, 920
Edge, *see* PlayStation Edge library
effect file, *see* shader

有效声压，见声音
ELF，请参阅可执行文件
省略，115
发射，见照明
封装，106
字节序，140
大，140
小，140，334
多字节数量，140
交换，141
内啡肽，43
能源，875
发动机，11
引擎和游戏之间的界限，11
可重用范围，12
发动机配置，470
食人魔，475
地震，474
神秘海域/TLOU，475
环境贴图，48，700，706
环境变量，236，473 EQ，参见均衡器等响度轮廓，916
均衡器，940，941，982
平衡，882
错误，另请参阅编译器
条件，119
数值方法，863
量化，949
欧拉角，386，403
欧拉公式，936，937
欣快感，43岁
事件，57，531，1040，1114
动画触发器，746
论点，1118
数据驱动，1131
调试，1128
驱动，37，57，1114，1122
转发，1116，1122
基于 GUI，1133
处理，1120，1151
内存分配，1128
对象，279
优先事项，1126
排队，1124
登记利益，1123
发送，1152
扳机，746
类型，1117
驱逐，查看缓存
异常处理，123
异常对象，123
交换指令，295
排除，见音频
可执行文件
BSS段，152
代码段，151
数据段，151
ELF 格式，151
文件，79，151
只读数据段，152
段旋转，152
段，151
文本段，151
显式并行性，206，225
爆炸，902
指数，参见浮点数
表达式树，792
F#，1136
面对，369
事实词典，1008
工厂，参见设计模式
跌落，看到声音，压力
扇形，见三角形
快速傅里叶变换，939
反馈，927
栅栏，另见屏障，见记忆排序语义
获取，查看 CPU，阶段
FFT，参见快速傅里叶变换
纤维，204，554
视野，46
格斗游戏，请参阅类型
文件输入/输出
异步，489，1072
缓冲，486
截止日期，492
同步，488
文件范围，请参阅范围
文件系统，482
路径，458，482，484
搜索路径，485
包装，487

effective sound pressure, *see* sound
ELF, *see* executable
elision, 115
emissive, *see* lighting
encapsulation, 106
endian, 140
big, 140
little, 140, 334
multibyte quantity, 140
swapping, 141
Endorphin, 43
energy, 875
engine, 11
line between engine and game, 11
reusability gamut, 12
engine configuration, 470
OGRE, 475
Quake, 474
Uncharted/TLOU, 475
environment map, 48, 700, 706
environment variable, 236, 473
EQ, *see* equalizer
equal-loudness contour, 916
equalizer, 940, 941, 982
equilibrium, 882
error, *see also* compiler
conditions, 119
numerical methods, 863
quantization, 949
Euler angles, 386, 403
Euler's formula, 936, 937
Euphoria, 43
event, 57, 531, 1040, 1114
animation trigger, 746
arguments, 1118
data-driven, 1131
debugging, 1128
driven, 37, 57, 1114, 1122
forwarding, 1116, 1122
GUI-based, 1133
handling, 1120, 1151
memory allocation, 1128
object, 279
priorities, 1126
queuing, 1124
registering interest, 1123
sending, 1152
trigger, 746
types, 1117
evict, *see* cache
exception handling, 123
exception object, 123
exchange instruction, 295
exclusion, *see* audio
executable
BSS segment, 152
code segment, 151
data segment, 151
ELF format, 151
file, 79, 151
read-only data segment, 152
rodata segment, 152
segment, 151
text segment, 151
explicit parallelism, 206, 225
explosions, 902
exponent, *see* floating-point
expression tree, 792
F#, 1136
facing, 369
fact dictionary, 1008
factory, *see* design pattern
fall-off, *see* sound, pressure
fan, *see* triangle
fast Fourier transform, 939
feedback, 927
fence, *see also* barrier, *see* memory ordering semantics
fetch, *see* CPU, stage
FFT, *see* fast Fourier transform
fiber, 204, 554
field of view, 46
fighting game, *see* genre
file I/O
asynchronous, 489, 1072
buffered, 486
deadline, 492
synchronous, 488
file scope, *see* scope
file system, 482
path, 458, 482, 484
search path, 485
wrapping, 487

过滤器，940，979
梳子，920
低通，940
缺口，940
通带，940
发送后，980
预发送，980
阻带，940
纹理，查看纹理，过滤
有限状态机，786，1042，1087，1154，1157
第一人称射击游戏，请参阅类型
固定功能管道，672
定点数，参见整数，133
手电筒，见照明
平面加权平均值，见动画
浮点数，133
非规范化，136
指数，134
无穷大，135
机器 epsilon，136
震级，134
尾数，134
非数字，135
精度，134
有效数字，134，135
低于正常水平，136
最后一个单位，137
浮点单元，166
流量控制，1157
FLT_MAX，135
流体动力学，715，910，924
流体模拟
基于位置，910
冲洗，218 Flynn 分类法，207 FMOD 工作室，995
FMV，观看全动态视频
FName，459
FO 最小值和最大值，查看声音，衰减
专注，看音频
愚蠢的图书馆，40，45，450
字体渲染，请参阅文本渲染
脚滑动，813，815
力量，373，859
作为一个函数，860
正向运动学，775
傅里叶变换，933，938，940
FPS，查看类型，查看帧速率
FPU，参见浮点单元
片段，664
帧，见动画
帧缓冲区，663
后台缓冲区，538
双缓冲，663
前缓冲器，538
交换，538
三重缓冲，664
帧速率，10，534，736
治理，537，538
帧间连贯性，参见时间连贯性

框架，529
Fraps，607
自由存储，另见分配，156
频率，913
角度，913
域，933，938
基本，938
回应，920，940
摩擦，880
件，47
FSM，参见有限状态机
全围栏，310，参见内存排序语义
全动态视频，49，735，1019，1161
功能
签名，146
功能语言，参见语言
融合，37
futex，271
Fx作曲家，670

G缓冲区，709
G因子，1015
增益，926，946，979，983
游戏
定义，8
游戏控制器，参见人机界面设备
游戏开发团队
艺术家，6
作曲家，6
工程师，5

filter, 940, 979
comb, 920
low-pass, 940
notch, 940
passband, 940
post-send, 980
pre-send, 980
stopband, 940
texture, *see* texture, filtering
finite state machine, 786, 1042, 1087, 1154, 1157
first-person shooter, *see* genre
fixed-function pipeline, 672
fixed-point, *see* integer, 133
flashlight, *see* lighting
flat weighted average, *see* animation
floating-point, 133
denormalized, 136
exponent, 134
infinity, 135
machine epsilon, 136
magnitude, 134
mantissa, 134
not-a-number, 135
precision, 134
significant digits, 134, 135
subnormal, 136
unit in the last place, 137
floating-point unit, 166
flow control, 1157
FLT_MAX, 135
fluid dynamics, 715, 910, 924
fluid simulation
position based, 910
flushed, 218
Flynn's taxonomy, 207
FMOD Studio, 995
FMV, *see* full-motion video
FName, 459
FO min and max, *see* sound, fall-off
focus, *see* audio
Folly library, 40, 45, 450
font rendering, *see* text rendering
foot sliding, 813, 815
force, 373, 859
as a function, 860
forward kinematics, 775
Fourier transform, 933, 938, 940
FPS, *see* genre, *see* frame rate
FPU, *see* floating-point unit
fragment, 664
frame, *see* animation
frame buffer, 663
back buffer, 538
double buffering, 663
front buffer, 538
swapping, 538
triple buffering, 664
frame rate, 10, 534, 736
governing, 537, 538
frame-to-frame coherency, *see* temporal coherency
framework, 529
Fraps, 607
free store, *see also* allocation, 156
frequency, 913
angular, 913
domain, 933, 938
fundamental, 938
response, 920, 940
friction, 880
frustum, 47
FSM, *see* finite state machine
full fence, 310, *see* memory ordering semantics
full-motion video, 49, 735, 1019, 1161
function
signature, 146
functional language, *see* language
Fusion, 37
futex, 271
Fx Composer, 670

G-buffer, 709
G-factor, 1015
gain, 926, 946, 979, 983
game
definition of, 8
game controller, *see* human interface device
game development team
artist, 6
composer, 6
engineer, 5

游戏设计师, 7 制作人, 7 出版商, 8
声音设计师, 6 配音演员, 6 编剧, 7
游戏引擎，参见引擎 游戏流控制,
1157 游戏循环, 526, 544, 899 并行化
, 545 暂停, 605 Pong , 527 单步执行,
532, 605 睡眠, 537 慢动作, 605 游戏
对象, 1021 异步更新, 1102 属性, 102
1 批量更新, 1090 行为, 1021, 1041
依赖项, 1093 实例, 1021 链接到引
擎, 1041 持久性, 1042 查询, 1042 查
询, 1085 引用, 1150 引用, 1042, 1079
生成, 1040 状态, 1087, 1100 状态缓
存, 1100 时间戳, 1101 类型, 1021 类
型, 1041
游戏玩法, 1015 流程, 1015, 1019, 1040
基础系统, 55, 1039 中心, 1019 线性, 10
19 目标, 1015, 1019, 1040 开放世界, 101
9 玩家机制, 55, 1015, 1162 区域, 1033
任务, 1019 水, 715 GameSalad, 37 伽马
校正, 718 响应曲线, 718 色域，参见照
明垃圾收集, 1081 高斯消元法, 378 gcc
，参见 GNU C/C++ 编译器通用 GP
U 编程,

参见 GPGPU
类型，13 大逃杀，24 格斗，19 格
斗，17 第一人称射击，14 大型多
人在线游戏，9，13，23 平台游戏
，15 竞速，19 实时战略，21 模拟
，31 策略，21 回合制战略，21 几
何，另请参阅网格画笔，请参阅画
笔几何，万向节锁，403，404，87
1 git，请参阅版本控制 GJK 算法
，839

唯一 ID，458，1041，1079，10
86 更新，527，1040，1086，109
3，1095
游戏对象模型，参见对象模型游戏状态
，1016游戏世界，9，56，64，1016，101
8区块，1019，1029，1062编辑器，59，6
4，714，1021，1023，1025，1030 – 1033
，1035，1043 Hammer，64，1025
辐射，64，1025
沙盒，1025
虚幻编辑器，64，66，1025

滑翔，41
全局照明模型，参见照明全局命名空
间，118 全局优化，参见编译器，
优化
全局唯一标识符，458，507，517 光泽
图，699 字形图集，716

game designer, 7
producer, 7
publisher, 8
sound designer, 6
voice actor, 6
writer, 7
game engine, *see* engine
game flow control, 1157
game loop, 526, 544, 899
parallelizing, 545
pausing, 605
Pong, 527
single-stepping, 532, 605
sleep, 537
slow motion, 605
game object, 1021
asynchronous updating, 1102
attribute, 1021
batched updating, 1090
behavior, 1021, 1041
dependencies, 1093
instance, 1021
linkage to engine, 1041
persistence, 1042
queries, 1042
query, 1085
reference, 1150
references, 1042, 1079
spawning, 1040
state, 1087, 1100
state caching, 1100
time-stamping, 1101
type, 1021
types, 1041
unique id, 458, 1041, 1079, 1086
updating, 527, 1040, 1086, 1093, 1095
game object model, *see* object model
game state, 1016
game world, 9, 56, 64, 1016, 1018
chunk, 1019, 1029, 1062
editor, 59, 64, 714, 1021, 1023, 1025, 1030–1033, 1035, 1043
Hammer, 64, 1025
Radiant, 64, 1025
Sandbox, 1025
UnrealEd, 64, 66, 1025
gameplay, 1015
flow, 1015, 1019, 1040
foundation system, 55, 1039
hub, 1019
linear, 1019
objectives, 1015, 1019, 1040
open world, 1019
player mechanics, 55, 1015, 1162
region, 1033
task, 1019
water, 715
GameSalad, 37
gamma
correction, 718
response curve, 718
gamut, *see* lighting
garbage collection, 1081
Gaussian elimination, 378
gcc, *see* GNU C/C++ compiler
general-purpose GPU programming, *see* GPGPU
genre, 13
battle royale, 24
brawler, 19
fighting, 17
first-person shooter, 14
massively multiplayer online game, 9, 13, 23
platformer, 15
racing, 19
real-time strategy, 21
simulation, 31
strategy, 21
turn-based strategy, 21
geometry, *see also* mesh
brush, *see* brush geometry
gimbal lock, 403, 404, 871
git, *see* version control
GJK algorithm, 839
Glide, 41
global illumination model, *see* lighting
global namespace, 118
global optimizations, *see* compiler, optimizations
globally unique identifier, 458, 507, 517
gloss map, 699
glyph atlas, 716

GNU C/C++编译器，78，336
高洛德着色，637
GPGPU，41，348，672
GPU，672 命令列表，692 计算单元，353，354 内核，350 锁定步骤执行，355 寄存器，678 SIMD 单元，350，354 阶段，355 线程，354，355 线程块，353 线程组，353，354 扭曲，355 波前，355 语法，792 Granny，42，507，516，783，790 图形着色语言，670 图形用户界面，49 图形处理单元，参见 GPU 重力，856，875，888 手榴弹物理，902 组，参见音频 GUI，参见图形用户界面

GUID，请参阅全局唯一标识符

H.26 x , 1162
Hadamard 积，364，676 头发，647，674，710，909 半条命 2，32，33
锤子，参见游戏世界编辑器手柄，440，554，1042，1079，1082声音，987陈旧，1083硬件T&L，41，672谐波，938

时间域中的缩放，974 散列，另请参阅容器，453 链接，453 碰撞，452，455，459 线性探测，455 开放寻址，453 二次探测，455 罗宾汉散列，456

哈斯克尔，1136年
Havok，42，824
Havok动画，42岁
HDMI, 954
头部相关传递函数，924 头文件，参见 C++ 耳机，944 净空高度，962 平视显示器，49 堆内存，156 高度场，参见地形高度图，参见纹理
HeroEngine，35

赫兹，534
异构系统架构, 678 十六进制编辑器, 102

十六进制，参见数字基数，132 HID，参见人机界面设备层次结构，另请参阅
坐标系，参见
也联合
高动态范围，参见照明高水位线，451，616命中率，参见缓存保持和等待，282齐次坐标，379水平添加，参见单条指令

多个数据
胡迪尼，64岁
HRTF，参见头部相关传递函数
HSA，参见异构系统
建筑学
HTML，1136
HUD，参见平视显示器 hUMA，参见异构系统
建筑学
人机接口设备，53，559 抽象控制索引，584 加速度计，566，567 执行器，570 模拟轴，564 模拟输入，564 和游戏循环，527 音频，570 按钮，563，564 和弦，575

上下文相关控件，585 控件映射，584 跨平台，582 死区，571

GNU C/C++ compiler, 78, 336
Gouraud shading, 637
GPGPU, 41, 348, 672
GPU, 672
command list, 692
compute unit, 353, 354
kernel, 350
lock step execution, 355
register, 678
SIMD unit, 350, 354
stage, 355
thread, 354, 355
thread block, 353
thread group, 353, 354
warp, 355
wavefront, 355
grammar, 792
Granny, 42, 507, 516, 783, 790
graphical shading language, 670
graphical user interface, 49
graphics processing unit, *see* GPU
gravity, 856, 875, 888
grenade physics, 902

group, *see* audio
GUI, *see* graphical user interface
GUID, *see* globally unique identifier

H.26*x*, 1162
Hadamard product, 364, 676
hair, 647, 674, 710, 909
Half-Life 2, 32, 33
Hammer, *see* game world editor
handle, 440, 554, 1042, 1079, 1082
sound, 987
stale, 1083
hardware T&L, 41, 672
harmonic, 938
scaling in time domain, 974
hashing, *see also* container, 453
chaining, 453
collision, 452, 455, 459
linear probing, 455
open addressing, 453
quadratic probing, 455
Robin Hood hashing, 456
Haskell, 1136
Havok, 42, 824
Havok Animation, 42
HDMI, 954
head-related transfer function, 924
header file, *see* C++
headphones, 944
headroom, 962
heads-up display, 49
heap memory, 156
height field, *see* terrain
heightmap, *see* texture
HeroEngine, 35
Hertz, 534
heterogeneous system architecture, 678
hex editor, 102
hexadecimal, *see* numeric base, 132
HID, *see* human interface device
hierarchy, *see also* coordinate system, *see also* joint
high dynamic range, *see* lighting
high water mark, 451, 616
hit rate, *see* cache
hold and wait, 282
homogeneous coordinates, 379
horizontal add, *see* single instruction multiple data
Houdini, 64
HRTF, *see* head-related transfer function
HSA, *see* heterogeneous system architecture
HTML, 1136
HUD, *see* heads-up display
hUMA, *see* heterogeneous system architecture
human interface device, 53, 559
abstract control indices, 584
accelerometer, 566, 567
actuator, 570
analog axis, 564
analog input, 564
and game loop, 527
audio, 570
button, 563, 564
chord, 575
context-sensitive controls, 585
control mapping, 584
cross-platform, 582
dead zone, 571

禁用, 586 DualShock, 562,
566, 567 力反馈, 53, 570 手
势, 577 红外传感器, 567
输入事件, 574 中断, 562
键盘, 54 鼠标, 566 多人游
戏, 582 PlayStation Eye, 56
8 轮询, 561 相对轴, 566 隆
隆声, 569 序列, 577 信号
过滤, 572 系统要求, 571 W
iimote, 559, 566, 567

XInput、562、563 混合构建，请参阅
构建配置 Hydro Thunder、1044 超线
程、225 滞后、971

I-cache，参见指令缓存
我碰撞，52
IDE，参见集成开发
环境
IGC，观看游戏内电影
IID，参见双耳强度差异
IK，参见逆运动学图像

位图，634 每像素位数，634 程序，1
51 采样，683 基于图像的照明，参见
照明 虚数，934 命令式语言，1136 隐
式并行，206，211 脉冲物理，876，8
77 响应，928，932，966 单位，参见
单位脉冲 脉冲响应，931 游戏内过场
动画，49，735，991，1019 自变量，
924

索引缓冲区，628，630 间接照明，参
见照明惯性张量，870 继承，106，16
2，1022，1067，1153 致命钻石，107
钻石问题，1049 多重，107，162，104
9 虚拟，107 初始化顺序（C ++），4
18 内联函数，83，参见编译器，

优化, 148
内积，367 输入寄存器，参见 GPU
实例限制，参见音频实例化几何体
，1018 实例化，参见动画瞬时声压
，参见

声音
指令缓存，196 指令重新排序，参见
编译器，
优化
指令流, 354 指令级并行, 214 Insure++,
50, 102 整数, 132 定点, 133 符号和幅度,
132 符号位, 132 有符号, 132 二进制补
码, 132 无符号, 132 集成开发环境,

78
一体化
显式欧拉，366，535 数值，10，535
强度，参见照明 对象间通信，参见事
件

系统
双耳
强度差异，959 时间差异，923 互连总
线，308 接口，118 干扰，919，944 建
设性和破坏性，919 插值

disabling, 586
DualShock, 562, 566, 567
force-feedback, 53, 570
gesture, 577
infrared sensor, 567
input event, 574
interrupt, 562
keyboard, 54
mouse, 566
multiplayer, 582
PlayStation Eye, 568
polling, 561
relative axis, 566
rumble, 569
sequence, 577
signal filtering, 572
system requirements, 571
Wiimote, 559, 566, 567
XInput, 562, 563
hybrid build, *see* build configuration
Hydro Thunder, 1044
hyperthreading, 225
hysteresis, 971

I-cache, *see* instruction cache
I-Collide, 52
IDE, *see* integrated development
environment
IGC, *see* in-game cinematic
IID, *see* interaural intensity difference
IK, *see* inverse kinematics
image
bitmapped, 634
bits per pixel, 634
of program, 151
sampling, 683
image-based lighting, *see* lighting
imaginary number, 934
imperative language, 1136
implicit parallelism, 206, 211
impulse
physics, 876, 877
response, 928, 932, 966
unit, *see* unit impulse
impulse response, 931
in-game cinematic, 49, 735, 991, 1019
independent variable, 924

index buffer, 628, 630
indirect lighting, *see* lighting
inertia tensor, 870
inheritance, 106, 162, 1022, 1067, 1153
deadly diamond, 107
diamond problem, 1049
multiple, 107, 162, 1049
virtual, 107
initialization order (C++), 418
inline function, 83, *see* compiler,
optimizations, 148
inner product, 367
input register, *see* GPU
instance limiting, *see* audio
instanced geometry, 1018
instancing, *see* animation
instantaneous acoustic pressure, *see*
sound
instruction cache, 196
instruction reordering, *see* compiler,
optimizations
instruction stream, 354
instruction-level parallelism, 214
Insure++, 50, 102
integer, 132
fixed-point, 133
sign and magnitude, 132
sign bit, 132
signed, 132
two's complement, 132
unsigned, 132
integrated development environment,
78
integration
explicit Euler, 366, 535
numerical, 10, 535
intensity, *see* lighting
inter-object communication, *see* event
system
interaural
intensity difference, 959
time difference, 923
interconnect bus, 308
interface, 118
interference, 919, 944
constructive and destructive, 919
interpolation

线性，375，400，546，755 球面线性插值，401 时间，757 顶点属性，635 解释型语言，1135 中断，95，186，232，234，247，292 服务例程，177，562 中断

人物对话，1001，1003，1004

关键操作，261，264，291 相交，另见碰撞，829 AABB 与 AABB，838 点与球，836 球与球，836 内在的，见编译器内在的逆运动学，775，811 调用，262 Irrlicht，36

ISR，参见中断、服务程序
ITD，参见双耳时间差迭代器，参见设计模式

看门人，参见设计模式 Java，1063 作业，549 计数器，554 声明，549 踢，549 同步，555 系统，545，549，1102 连接，参见线程 关节，参见动画 坐标，730 索引，728 名称，728 非均匀缩放，731 父级，728 根，728 缩放，731

操纵杆，参见人机接口设备

k d-tree, 47, 695
内核，参见 GPU 内核调用，267 字距调整，717 关键帧，736

键值对，452，508，1031，1032，1065, 1119, 1139
键盘，参见人机界面设备 踢出作业，参见作业 Killzone 2，709 运动学，785 末端执行器，775 正向，775 反向，775，811 Kismet，参见虚幻引擎 4，蓝图

Kynapse，见人工智能

L1 缓存，参见缓存 lambda，1144
车道，参见单指令多数据语言，106 编译型，1135 声明式，1136 函数式，1101，1136 命令式，1136 解释式，1135 面向对象，106，1136 过程式，1136 反射，1022，1042，1052，1064，1065，1136

后期函数绑定，1115 后期混响，920 延迟音频总线，984 内存访问，188 管道，214 Lead Werks 引擎，35 左手法则，371 细节级别，625、714、715 词汇范围，参见范围 LFE，参见低频效果 libgcm，41 库，79、529 生命周期

调试原语，597 资源，503
光照贴图，48，653，672 照明，633，647 吸收，633 环境，649，654 环境光遮蔽，705 区域光，655

linear, 375, 400, 546, 755
spherical linear interpolation, 401
temporal, 757
vertex attributes, 635
interpreted language, 1135
interrupt, 95, 186, 232, 234, 247, 292
service routine, 177, 562
interruption
of character dialog, 1001, 1003, 1004
of critical operation, 261, 264, 291
intersection, *see also* collision, 829
AABB versus AABB, 838
point versus sphere, 836
sphere versus sphere, 836
intrinsic, *see* compiler intrinsic
inverse kinematics, 775, 811
invocation, 262
Irrlicht, 36
ISR, *see* interrupt, service routine
ITD, *see* interaural time difference
iterator, *see* design pattern

janitor, *see* design pattern
Java, 1063
job, 549
counter, 554
declaration, 549
kicking, 549
synchronization, 555
system, 545, 549, 1102
join, *see* thread
joint, *see also* animation
coordinate, 730
index, 728
name, 728
nonuniform scale, 731
parent, 728
root, 728
scale, 731
joystick, *see* human interface device

*k*d-tree, 47, 695
kernel, *see* GPU
kernel call, 267
kerning, 717
key frame, 736
key-value pair, 452, 508, 1031, 1032, 1065, 1119, 1139
keyboard, *see* human interface device
kicking a job, *see* job
Killzone 2, 709
kinematics, 785
end effector, 775
forward, 775
inverse, 775, 811
Kismet, *see* Unreal Engine 4, Blueprints
Kynapse, *see* artificial intelligence

L1 cache, *see* cache
lambda, 1144
lane, *see* single instruction multiple data
language, 106
compiled, 1135
declarative, 1136
functional, 1101, 1136
imperative, 1136
interpreted, 1135
object-oriented, 106, 1136
procedural, 1136
reflection, 1022, 1042, 1052, 1064, 1065, 1136
late function binding, 1115
late reverberations, 920
latency
audio bus, 984
memory access, 188
pipeline, 214
LeadWerks Engine, 35
left-hand rule, 371
level of detail, 625, 714, 715
lexically scoped, *see* scope
LFE, *see* low-frequency effects
libgcm, 41
library, 79, 529
lifetime
debug primitive, 597
resource, 503
light map, 48, 653, 672
lighting, 633, 647
absorption, 633
ambient, 649, 654
ambient occlusion, 705
area light, 655

Blinn-Phong 光照模型, 652 光晕, 48, 655, 702 BRDF, 652

BSSRDF, 707 焦
散, 706 衍射, 63
3 漫反射, 649
直接, 647 定向,
654 发射, 655
闪光灯, 655 色
域, 633

全局光照模型，648，702

高动态范围, 634, 655, 702 基于图像, 698
间接, 648, 702 强度, 633 与物质的相互
作用, 633 局部照明模型, 647 介质, 633
逐像素着色, 637 冯氏反射模型, 649 点
光源, 654 预计算辐射传递, 708 辐射度,
648 光线追踪, 648 反射, 633, 706 折射, 6
33 散射, 634, 707 光源, 622, 653 镜面贴
图, 699 镜面反射, 649 镜面反射功率图,
670, 699 聚光灯, 654 静态, 653, 672 透射,
633 传输模型, 622, 647 观察方向, 650 波
长, 633光波，669线，408

视线，595，849，1103 线
级，945 线性，926 代数，
359

近似值，625，745 动量，859 探
测，参见散列 时不变系统，926
速度，858 线性化，266 链接寄
存器，297 链接，149 外部，149
内部，149，151，153 链接器，
78 优化，85 Lisp，477，797，11
44 列表，参见三角形，参见容
器监听器，956 小端序，参见字
节序 活锁，283

加载链接/存储条件，297 局部照明模
型，参见照明局部优化，参见编译器
，
优化
局部变量，参见变量本地化，45
6，462，466 位置标签，1009 基
于位置的娱乐，30 定位器，747

无锁算法，267，287，290，328，
1102
无需锁定断言，请参阅断言运动周期，
722 噪声，773 关键，761 目标，761 LO
D，请参阅详细程度对数，915 日志记
录，589 通道，592 到文件，593 详细程
度，591 Loki 库，40，450 查找表，517
，521，522，640，674 循环展开，请参
阅编译器，
优化
循环，参见动画无损压缩，参见压缩
响度，参见音频

Blinn-Phong lighting model, 652
bloom, 48, 655, 702
BRDF, 652
BSSRDF, 707
caustics, 706
diffraction, 633
diffuse, 649
direct, 647
directional, 654
emissive, 655
flashlight, 655
gamut, 633
global illumination model, 648, 702
high dynamic range, 634, 655, 702
image-based, 698
indirect, 648, 702
intensity, 633
interactions with matter, 633
local illumination model, 647
medium, 633
per-pixel shading, 637
Phong reflection model, 649
point light, 654
precomputed radiance transfer, 708
radiosity, 648
ray tracing, 648
reflection, 633, 706
refraction, 633
scattering, 634, 707
source, 622, 653
spec map, 699
specular, 649
specular power map, 670, 699
spot light, 654
static, 653, 672
transmission, 633
transport model, 622, 647
viewing direction, 650
wavelength, 633
Lightwave, 669
line, 408
line of sight, 595, 849, 1103
line-level, 945
linear, 926
algebra, 359
approximation, 625, 745
momentum, 859
probing, *see* hashing
time-invariant system, 926
velocity, 858
linearizability, 266
link register, 297
linkage, 149
external, 149
internal, 149, 151, 153
linker, 78
optimizations, 85
Lisp, 477, 797, 1144
list, *see* triangle, *see* container
listener, 956
little-endian, *see* endian
livelock, 283
load linked/store conditional, 297
local illumination model, *see* lighting
local optimizations, *see* compiler, optimizations
local variable, *see* variable
localization, 456, 462, 466
location tag, 1009
location-based entertainment, 30
locator, 747
lock-free algorithm, 267, 287, 290, 328, 1102
lock-not-needed assertion, *see* assertion
locomotion
cycle, 722
noise, 773
pivotal, 761
targeted, 761
LOD, *see* level of detail
logarithmic, 915
logging, 589
channel, 592
to file, 593
verbosity, 591
Loki library, 40, 450
look-up table, 517, 521, 522, 640, 674
loop unrolling, *see* compiler, optimizations
looping, *see* animation
lossless compression, *see* compression
loudness, *see* audio

低频效果，944、959、982 低通滤波器，572 LPCM，参见脉冲编码调制
LTI 系统，参见线性时不变系统
LU分解，378
星期一，1139
木材场，36
LUT，参见查找表

机器 epsilon，参见浮点宏，333 Macromedia Fusion，37 madd 指令，340 幅度，参见浮点复数，参见复数向量，参见向量 make，86 托管 C++，489 管理器，418 清单常量，152 尾数，参见浮点多核，参见 GPU，210 质量，857

大型多人在线游戏，参见类型
材料，682 编辑器，670 系统，46 视觉，646，670 数学库，44 矩阵，375 3×3，380，404
4 × 3, 384
4 × 4, 343, 363, 376, 380, 382,□□ 390, 430, 730, 756, 769 仿射, 376 相机到世界, 46 列矩阵, 377 转换为四元数, 399 恒等式, 378
内存表示，392 逆，378 各向同性，376 关节到模型，752 模型到世界，369，390，631，754 正交投影，662
正交，376 透视投影矩阵，660 乘积，376 纯旋转，404 行矩阵，377 特殊正交，376 转置，379 视图到世界，656 世界到视图，656 矩阵调色板，753 Maya，62，669 力学经典，854 介质，另请参阅照明 MEL 语言，502，1134

Meltdown 漏洞利用，239 内存访问周期，293 访问模式，426 对齐，159，333，431 缓存，参见缓存卡，570

控制器, 160, 166, 176, 184, 212, 227, 291, 356 损坏, 101 调试内存, 461 碎片整理, 439, 1077 围栏, 304，另请参阅屏障, 316 碎片, 437 游戏内统计数据, 615 泄漏, 101, 615 管理, 44, 426, 511 管理单元，请参阅内存、控制器排序错误, 291 排序语义, 305, 311, 315 获取, 311, 312, 316 消耗, 316 完全围栏, 311, 316 放松, 315 释放, 311, 312, 316 重定位, 439, 987, 1077 共享，请参阅并发小内存分配器, 1076 摇杆、487 虚拟、437、615 菜单

low-frequency effects, 944, 959, 982
low-pass filter, 572
LPCM, *see* pulse-code modulation
LTI system, *see* linear time-invariant system
LU decomposition, 378
Lua, 1139
Lumberyard, 36
LUT, *see* look-up table

machine epsilon, *see* floating-point
macro, 333
Macromedia Fusion, 37
madd instruction, 340
magnitude, *see also* floating-point
    complex, *see* complex number
    vector, *see* vector
make, 86
managed C++, 489
manager, 418
manifest constants, 152
mantissa, *see* floating-point
manycore, *see also* GPU, 210
mass, 857
massively multiplayer online game, *see* genre
material, 682
    editor, 670
    system, 46
    visual, 646, 670
math library, 44
matrix, 375
    3 × 3, 380, 404
    4 × 3, 384
    4 × 4, 343, 363, 376, 380, 382, 390, 430, 730, 756, 769
    affine, 376
    camera-to-world, 46
    column matrix, 377
    conversion to quaternion, 399
    identity, 378
    in-memory representation, 392
    inverse, 378
    isotropic, 376
    joint-to-model, 752
    model-to-world, 369, 390, 631, 754
    orthographic projection, 662
    orthonormal, 376
    perspective projection matrix, 660
    product, 376
    pure rotation, 404
    row matrix, 377
    special orthogonal, 376
    transpose, 379
    view-to-world, 656
    world-to-view, 656
matrix palette, 753
Maya, 62, 669
mechanics
    classical, 854
medium, *see also* lighting
MEL language, 502, 1134
Meltdown exploit, 239
memory
    access cycle, 293
    access patterns, 426
    alignment, 159, 333, 431
    cache, *see* cache
    card, 570
    controller, 160, 166, 176, 184, 212, 227, 291, 356
    corruption, 101
    debug memory, 461
    defragmentation, 439, 1077
    fence, 304, *see also* barrier, 316
    fragmentation, 437
    in-game statistics, 615
    leak, 101, 615
    management, 44, 426, 511
    management unit, *see* memory, controller
    ordering bugs, 291
    ordering semantics, 305, 311, 315
        acquire, 311, 312, 316
        consume, 316
        full fence, 311, 316
        relaxed, 315
        release, 311, 312, 316
    relocation, 439, 987, 1077
    shared, *see* concurrency
    small memory allocator, 1076
    stick, 487
    virtual, 437, 615
menu

游戏中, 50, 475, 601 网格,
61, 121, 625 构建, 627 实例,
631, 1090 渐进式, 626 静
态, 1018, 1029 子网格, 646
MESI, 197

MESIF，197 消息，另请参阅
事件映射，1121 传递，请参
阅并发泵，46，529 公制，请
参阅 SI 单位 mic-level，945 微
操作，172 麦克风，942 Micro
soft Excel，467，614

Microsoft Visual Studio，70，78
迈尔斯音响系统，995
MIMD，参见多指令
多个数据
Minkowski 和/差，839 mipmapping，
643，679 MISD，参见多指令单
数据
混合快照，990 混合类，1
049 混合现实，27 混合，
参见音频 MKS 单位制，8
56
MMO，请参阅类型
MMX，参见流式 SIMD 扩展 mod 社
区，11 模型 3D，参见网格分析，10
闭式，10 数学，9 数值，10 建模器 (3
D)，6 调制，940 MOESI，197 莫尔条
纹，643 惯性矩，867 角动量，872
线性，859 监视器，参见并发性
MonoGame，34 单片类层次结
构，1046 运动模糊，719 运动捕
捉，6 电影捕捉，607 电影播放
器□□，1161 移动平均，537，572
MP3，参见音频，文件格式

MPEG，1162，另请参阅音频、文件格
式多字节数量，请参阅字节序多核，21
0、226、437多级缓存，请参阅缓存多
播放器，55

人机接口设备, 582 网络化, 55, 1162 复制
, 1042 分屏, 55 多重继承, 1049 多指令多
数据, 207,

348
多指令单数据，207 乘法作为复杂旋转
，935 乘法定义符号错误，146 多任务
，540，545 异步编程，1102 合作，234
，250，554，1140，1154 GPU，672

与游戏对象更新接口，1102
作业，545

抢占式，40，230，234，247，250，554
，1140 pthreads，545 睡眠，492，537
，546 SPURS，545 线程，487，492，54
5 音乐，1010

互斥锁，112，245，264，267 – 269，271，273，
282, 292, 555
互斥，参见 mutex

命名空间, 118 自然
数, 132 NDEBUG , 8
2

in-game, 50, 475, 601
mesh, 61, 121, 625
constructing, 627
instance, 631, 1090
progressive, 626
static, 1018, 1029
submesh, 646
MESI, 197
MESIF, 197
message, *see also* event
map, 1121
passing, *see* concurrency
pump, 46, 529
metric, *see* SI units
mic-level, 945
micro-operations, 172
microphone, 942
Microsoft Excel, 467, 614
Microsoft Visual Studio, 70, 78
Miles Sound System, 995
MIMD, *see* multiple instruction
multiple data
Minkowski sum/difference, 839
mipmapping, 643, 679
MISD, *see* multiple instruction single
data
mix snapshot, 990
mix-in class, 1049
mixed reality, 27
mixing, *see* audio
MKS system of units, 856
MMO, *see* genre
MMX, *see* streaming SIMD extensions
mod community, 11
model
3D, *see* mesh
analytic, 10
closed-form, 10
mathematical, 9
numerical, 10
modeler (3D), 6
modulation, 940
MOESI, 197
moiré pattern, 643
moment of inertia, 867
momentum
angular, 872
linear, 859
monitor, *see* concurrency
MonoGame, 34
monolithic class hierarchy, 1046
motion blur, 719
motion capture, 6
movie capture, 607
movie player, 1161
moving average, 537, 572
MP3, *see* audio, file formats
MPEG, 1162, *see also* audio, file formats
multibyte quantity, *see* endian
multicore, 210, 226, 437
multilevel cache, *see* cache
multiplayer, 55
human interface devices, 582
networked, 55, 1162
replication, 1042
split-screen, 55
multiple inheritance, 1049
multiple instruction multiple data, 207,
348
multiple instruction single data, 207
multiplication as complex rotation, 935
multiply defined symbol error, 146
multitasking, 540, 545
asynchronous programming, 1102
cooperative, 234, 250, 554, 1140,
1154
GPU, 672
interfacing with game object
update, 1102
job, 545
preemptive, 40, 230, 234, 247, 250,
554, 1140
pthreads, 545
sleep, 492, 537, 546
SPURS, 545
thread, 487, 492, 545
music, 1010
mutex, 112, 245, 264, 267–269, 271, 273,
282, 292, 555
mutual exclusion, *see* mutex

namespace, 118
natural number, 132
NDEBUG, 82

最近邻，查看纹理，过滤
负强化，919
new, *see* C++
牛顿恢复定律，876
牛顿运动定律，859
牛顿力学，854
非阻塞算法，289，290
　无锁，289
　无障碍，289
　无需等待，289
非阻塞函数，268，另请参阅
　阻塞函数，1102
非玩家角色，15，56，734，742
非交互序列，735
非均匀有理 B 样条，624
法线，另见矢量
　地图，查看纹理
　平面，369
Novodex，参见 PhysX
NPC，参见非玩家角色
NTSC、534、542、663
数字基数
　二进制，131
　十进制，131
　十六进制，132
数值方法，863
NURBS，参见非均匀有理数
　B样条
奈奎斯特频率，949

OBB，参见边界框
目标文件, 79
对象模型，另请参阅游戏对象模型
　C++，1022
　组件，1052，1055
　组件创建，1053
　组件所有权，1053
　Excel，1022游戏，5
　6，1022，1040
　水雷，1044
　接口，1022
　多任务处理，1107
　以对象为中心，1043，1044
　OMT，1022
　以财产为中心，1043，1060
　运行时，1023，1043
　软件，1022
　工具侧，1023
　唯一 ID，1056
面向对象，参见语言
阻塞，见音频
无障碍算法，290
Ocaml，1136
遮挡，见音频
八进制，参见数字基数
八叉树，47，694
ODE，请参阅开放动力学引擎
奥格 · 沃比斯，953
食人魔, 36, 45, 47, 91, 139, 140, 789
一帧关闭错误，1099
OOO 执行，参见 CPU
不透明度，623
开放寻址，参见哈希
开放 Dynamics 引擎，42，823
打开哈希表，查看容器
开源软件，34
OpenAL，994
OpenGL，41，46
　透视投影矩阵，660
　视图空间，387
　查看量，659
OpenTissue，825
操作系统, 40, 43, 79, 80,
　151 – 153, 156
　DOS，483
　Mac OS，483 Microsoft
　Windows，483
　UNIX，483
运算符重载，445
操作员强度降低，参见
　编译器、优化
光纤音频连接器，954
优化，另请参阅编译器，
　优化，99，165，216，317
　，447，608
OrbisAnim 库，查看动画
常微分方程，860，869，873

正交，363
OS，请参阅操作系统
乱序执行，参见 CPU
外积，367，370
输出寄存器，参见 GPU
透支，692

nearest neighbor, *see* texture, filtering
negative reinforcement, 919
new, *see* C++
Newton's law of restitution, 876
Newton's laws of motion, 859
Newtonian mechanics, 854
non-blocking algorithm, 289, 290
　lock-free, 289
　obstruction-free, 289
　wait-free, 289
non-blocking function, 268, *see also*
　blocking function, 1102
non-player character, 15, 56, 734, 742
noninteractive sequence, 735
nonuniform rational B-spline, 624
normal, *see also* vector
　map, *see* texture
　of plane, 369
Novodex, *see* PhysX
NPC, *see* non-player character
NTSC, 534, 542, 663
numeric base
　binary, 131
　decimal, 131
　hexadecimal, 132
numerical methods, 863
NURBS, *see* nonuniform rational
　B-spline
Nyquist frequency, 949

OBB, *see* bounding box
object file, 79
object model, *see also* game object model
　C++, 1022
　component, 1052, 1055
　component creation, 1053
　component ownership, 1053
　Excel, 1022
　game, 56, 1022, 1040
　Hydro Thunder, 1044
　interface, 1022
　multitasking, 1107
　object-centric, 1043, 1044
　OMT, 1022
　property-centric, 1043, 1060
　runtime, 1023, 1043
　software, 1022
　tool-side, 1023
　unique ids, 1056
object-oriented, *see* language
obstruction, *see* audio
obstruction-free algorithm, 290
Ocaml, 1136
occlusion, *see* audio
octal, *see* numeric base
octree, 47, 694
ODE, *see* Open Dynamics Engine
Ogg Vorbis, 953
OGRE, 36, 45, 47, 91, 139, 140, 789
one-frame-off bug, 1099
OOO execution, *see* CPU
opacity, 623
open addressing, *see* hashing
Open Dynamics Engine, 42, 823
open hash table, *see* container
open-source software, 34
OpenAL, 994
OpenGL, 41, 46
　perspective projection matrix, 660
　view space, 387
　view volume, 659
OpenTissue, 825
operating system, 40, 43, 79, 80,
　151–153, 156
　DOS, 483
　Mac OS, 483
　Microsoft Windows, 483
　UNIX, 483
operator overloading, 445
operator strength reduction, *see*
　compiler, optimizations
optical audio connector, 954
optimization, *see also* compiler,
　optimizations, 99, 165, 216,
　317, 447, 608
OrbisAnim library, *see* animation
ordinary differential equation, 860, 869,
　873
orthogonal, 363
OS, *see* operating system
out-of-order execution, *see* CPU
outer product, 367, 370
output register, *see* GPU
overdraw, 692

覆盖，49，657，716，参见渲染图形用户界面，32，80，111平视显示器，657

包文件，497，499，506 页面错误，186，239 画家算法，664 PAL，534，542，663 平移，参见音频 Panda3D，36，1143 视差遮挡映射，参见

渲染
并行性, 203, 205 数据, 207, 348, 544, 546 显式, 206, 225 隐式, 206, 211 任务, 207, 544, 545 平行四边形面积, 371 参数方程, 408, 848 表面, 624 帕累托原则，参见 80/20 规则 粒子系统, 46, 48, 64, 711, 1089 通过

读取和写入，310 patch，624 Bézier，624 bicubic，624 N-patch，624

非均匀有理B样条，624 路径追踪，970 Pawn，1141

PCM，参见脉冲编码调制窥孔优化，参见编译器，
优化
半影，655，703 每像素着色，请参阅每用户照明选项，474 感知响度，请参阅音频位置感知，请参阅音频感知编码，952 Perforce，请参阅版本控制周期性的，913，938 环绕声，964 垂直轴，360，837 距离，410，658，688

矢量, 363, 369, 370, 392, 635, 638 波, 917 持久性, 1042 透视, 657 幻影图像, 959 相位，另请参阅复数动画, 934 移位, 913, 919 声码器, 974 Phong 反射模型，请参阅光照真实感, 622 PhyreEngine, 34

基于物理的声音合成，996 物理，另见碰撞，51，527，674 和乐趣，819 Havok，52 库，42 Havok，42

开放动力学引擎, 42
PhysX，42
开放动力学引擎, 52
PhysX，52
刚体动力学，51，726，854 模拟，51 更新，1089 水，715 世界，828 物理抽象层，825

PhysX，42，824
分段线性近似，746，
970，见近似值
翅膀，924
管道，213，667，672，976，1092 资产调节，参见资产调节管道
带宽，214 延迟，214，667 吞吐量，214，667 工具，参见资产调节管道

间距, 386 像素, 622, 655 放置新, 520 普通旧数据结构, 114, 520 平面图, 657 平面, 369, 409, 658 到原点的距离, 409

overlay, 49, 657, 716, *see* rendering
graphical user interface, 32, 80, 111
heads-up display, 657

package file, 497, 499, 506
page fault, 186, 239
painter's algorithm, 664
PAL, 534, 542, 663
pan, *see* audio
Panda3D, 36, 1143
parallax occlusion mapping, *see* rendering
parallelism, 203, 205
data, 207, 348, 544, 546
explicit, 206, 225
implicit, 206, 211
task, 207, 544, 545
parallelogram
area of, 371
parametric equation, 408, 848
surface, 624
Pareto principle, *see* 80/20 rule
particle system, 46, 48, 64, 711, 1089
passing
of reads and writes, 310
patch, 624
Bézier, 624
bicubic, 624
N-patch, 624
nonuniform rational B-spline, 624
path tracing, 970
Pawn, 1141
PCM, *see* pulse-code modulation
peep-hole optimization, *see* compiler, optimizations
penumbra, 655, 703
per-pixel shading, *see* lighting
per-user options, 474
perceived loudness, *see* audio
perception of position, *see* audio
perceptual coding, 952
Perforce, *see* version control
periodic, 913, 938
periphonic, 964
perpendicular
axes, 360, 837
distance, 410, 658, 688
vectors, 363, 369, 370, 392, 635, 638
wave, 917
persistence, 1042
perspective, 657
phantom image, 959
phase, *see also* animation
of complex number, 934
shift, 913, 919
vocoder, 974
Phong reflection model, *see* lighting
photorealism, 622
PhyreEngine, 34
physically based sound synthesis, 996
physics, *see also* collision, 51, 527, 674
and fun, 819
Havok, 52
library, 42
Havok, 42
Open Dynamics Engine, 42
PhysX, 42
Open Dynamics Engine, 52
PhysX, 52
rigid body dynamics, 51, 726, 854
simulation, 51
updating, 1089
water, 715
world, 828
Physics Abstraction Layer, 825
PhysX, 42, 824
piecewise-linear approximation, 746, 970, *see* approximation
pinnae, 924
pipeline, 213, 667, 672, 976, 1092
asset conditioning, *see* asset conditioning pipeline
bandwidth, 214
latency, 214, 667
throughput, 214, 667
tools, *see* asset conditioning pipeline
pitch, 386
pixel, 622, 655
placement new, 520
plain old data structure, 114, 520
plan view, 657
plane, 369, 409, 658
distance to origin, 409

点范式，409，658 平台独立层，43 平台游戏，参见类型 播放器 I/O，参见人机界面设备 播放列表，1011 PlayStation Edge 库，41，42

PlayStation Network，481，487
POD，参见普通旧数据结构点，360 算术，365 点光，参见照明指针修复，519 极性图案，942 极化，917 多边形

渲染, 624, 625 汤, 854 多面体凸, 412 多态性, 109, 162 复音, 977 便携式网络图形, 642 门户, 47 剔除, 689 声音, 972 姿势

作为基础的变化，732
绑定姿势，728，729，751 当前，729 当前姿势，751 全局，733

内存表示，732 插值，736 局部，730 T姿势，729

基于位置的流体模拟，910 正强化，919 后效应，48，719 后加载初始化，521 后增量（C++），445 潜在可见集，47 电位器，947 功率，961

电源处理单元，参见 PPU 前置放大器，982 预延迟，参见音频精度，参见浮点

预计算辐射传输，参见灯光
谓词，221，346 向量，344，346，347 抢占，参见多任务
预增量（C++），445 预处理器，参见 C++ 压力，参见声音，912 原语

调试绘图，597 几何，46，646 网格材料对，646 提交，691 原始数据类型，146 printf 调试，589 优先级，另请参阅线程对话框，1001 反转，284 语音，991 私人，请参阅 C++ 过程语言，1136 处理器利用率，545，1105 生产者-消费者问题，271 配置文件陷阱，557 配置文件引导的优化，请参阅编译器、优化
分析工具, 50, 99 分层, 609 游戏内, 608 检测, 100, 611 统计, 100 程序顺序, 262 程序堆栈, 参见调用堆栈进度, 参见并发投影, 368, 657 正交, 21, 657, 662, 716, 1029

透视, 22, 657, 660, 661, 1029 透视缩短, 657, 661

Prolog，1136 传播建模，965
使用 LTI 系统，966 属性类，1059 属性网格，1031 属性对象，1059 与组件相比，1059

point-normal form, 409, 658
platform independence layer, 43
platformer, *see* genre
player I/O, *see* human interface device
playlist, 1011
PlayStation Edge library, 41, 42
PlayStation Network, 481, 487
POD, *see* plain old data structure
point, 360
arithmetic, 365
point light, *see* lighting
pointer fix-up, 519
polar pattern, 942
polarization, 917
polygon
rendering, 624, 625
soup, 854
polyhedron
convex, 412
polymorphism, 109, 162
polyphony, 977
Portable Network Graphics, 642
portal, 47
culling, 689
sound, 972
pose
as change of basis, 732
bind pose, 728, 729, 751
current, 729
current pose, 751
global, 733
in-memory representation, 732
interpolation, 736
local, 730
T-pose, 729
position based fluid simulation, 910
positive reinforcement, 919
post effect, 48, 719
post-load initialization, 521
postincrement (C++), 445
potentially visible set, 47
potentiometer, 947
power, 961
power processing unit, *see* PPU
pre-amp, 982
pre-delay, *see* audio
precision, *see* floating-point
precomputed radiance transfer, *see* lighting
predication, 221, 346
vector, 344, 346, 347
preemption, *see* multitasking
preincrement (C++), 445
preprocessor, *see* C++
pressure, *see also* sound, 912
primitive
debug drawing, 597
geometric, 46, 646
mesh-material pair, 646
submission, 691
primitive data type, 146
printf debugging, 589
priority, *see also* thread
dialog, 1001
inversion, 284
voice, 991
private, *see* C++
procedural language, 1136
processor utilization, 545, 1105
producer-consumer problem, 271
profile trap, 557
profile-guided optimizations, *see* compiler, optimizations
profiling tools, 50, 99
hierarchical, 609
in-game, 608
instrumenting, 100, 611
statistical, 100
program order, 262
program stack, *see* call stack
progress, *see* concurrency
projection, 368, 657
orthographic, 21, 657, 662, 716, 1029
perspective, 22, 657, 660, 661, 1029
perspective foreshortening, 657, 661
Prolog, 1136
propagation modeling, 965
with LTI system, 966
property class, 1059
property grid, 1031
property object, 1059
versus component, 1059

以属性为中心，参见对象模型伪向量，362 PUBG，参见类型，大逃杀公开，参见 C++ 脉冲编码调制，948 双关，参见类型双关纯组件模型，1056 Purify，50 推锁，324 PVS，参见潜在可见集

Python，1134，1140方法表，1144

二次探测，参见哈希四叉树，47，694 Quake C，1138

地震引擎，11，31，35，45，64，1018，1025
量化，50
量化, 136, 参见压缩四元数, 394, 405 连接, 398 共轭, 397 转换为矩阵, 399 对偶, 406 逆, 396 乘积, 396 旋转矢量, 397 队列, 442 快速时间事件, 735

种族，参见数据种族竞争条件，205，258 赛车游戏，参见类型 Radiant，参见游戏世界编辑器辐射度，参见照明布娃娃，777，886 RAGE，参见 Rockstar Advanced Game 引擎
RAII，参见设计模式随机数，412 Diehard 测试，413
KISS99、414 线性同余、412 梅森扭转器、413
万物之母，413
PCG, 414
Xorshift，414
RAPID，52 快速迭代，1024 稀疏化，912 光栅化，664，675 Rational Purify，101 射线，408 射线投射，1103 射线追踪，参见照明 RC 滤波器，参见滤波器

RCA 插孔，参见音频
RCS，参见版本控制
RCU，参见读-复制-更新 读-获取，311，312，319 读-复制-更新，324 读-修改-写入，259 只读数据段，参见可执行文件 文件，151
读写锁，324 实时策略，参见类型现实，参见虚拟现实记录和回放，50，538 矩形失效，525 Redis，67

精简指令集，223 引用计数，510，1080 参照完整性，495，498，503，516 反射，另请参阅 照明，1042，另请参阅
语言各向异性，634 漫反射，634 反射率，650 镜面反射，634 波，917 折射，参见照明，917 寄存器，另请参阅 CPU 注册表（Microsoft Windows），473 相对速度，参见速度 宽松内存顺序，参见内存 排序语义
发布构建，请参阅构建配置发布围栏，请参阅内存排序 语义
释放内存顺序，参见memory 排序语义
浮雕映射，参见渲染循环，526

property-centric, *see* object model
pseudovector, 362
PUBG, *see* genre, battle royale
public, *see* C++
pulse-code modulation, 948
punning, *see* type punning
pure component model, 1056
Purify, 50
push lock, 324
PVS, *see* potentially visible set
Python, 1134, 1140
method table, 1144

quadratic probing, *see* hashing
quadtree, 47, 694
Quake C, 1138
Quake Engine, 11, 31, 35, 45, 64, 1018, 1025
Quantify, 50
quantization, 136, *see* compression
quaternion, 394, 405
concatenation, 398
conjugate, 397
conversion to matrix, 399
dual, 406
inverse, 396
product, 396
rotating vectors with, 397
queue, 442
quick time event, 735

race, *see* data race
race condition, 205, 258
racing game, *see* genre
Radiant, *see* game world editor
radiosity, *see* lighting
rag doll, 777, 886
RAGE, *see* Rockstar Advanced Game Engine
RAII, *see* design pattern
random number, 412
Diehard tests, 413
KISS99, 414
linear congruential, 412
Mersenne Twister, 413
Mother of All, 413
PCG, 414
Xorshift, 414
RAPID, 52
rapid iteration, 1024
rarefaction, 912
rasterization, 664, 675
Rational Purify, 101
ray, 408
ray cast, 1103
ray tracing, *see* lighting
RC filter, *see* filter
RCA jack, *see* audio
RCS, *see* version control
RCU, *see* read-copy-update
read-acquire, 311, 312, 319
read-copy-update, 324
read-modify-write, 259
read-only data segment, *see* executable file, 151
readers-writer lock, 324
real-time strategy, *see* genre
reality, *see* virtual reality
record and play back, 50, 538
rectangle invalidation, 525
Redis, 67
reduced instruction set, 223
reference counting, 510, 1080
referential integrity, 495, 498, 503, 516
reflection, *see also* lighting, 1042, *see also* language
anisotropic, 634
diffuse, 634
reflectivity, 650
specular, 634
wave, 917
refraction, *see* lighting, 917
register, *see also* CPU
registry (Microsoft Windows), 473
relative velocity, *see* velocity
relaxed memory order, *see* memory ordering semantics
release build, *see* build configuration
release fence, *see* memory ordering semantics
release memory order, *see* memory ordering semantics
relief mapping, *see* rendering
render loop, 526

渲染状态，691，692
泄漏，691
渲染目标，664
渲染
音频，参见音频
广告牌，见广告牌
凹凸贴图，699
延期，709
位移贴图，698
G缓冲区，709
视差遮挡映射，698
地形测绘，698
渲染引擎, 45
图形设备接口, 46
低级渲染器，46，47
渲染包, 46
场景图，47，622，693，697
渲染方程，622，649
渲染管道
申请阶段，668，687
资产调节阶段，668，671
数据转换, 669
几何处理阶段，668
GPU管道，672
合并，676
光栅化阶段，668
流输出，674
工具阶段，668
三角形设置，675
三角形遍历，675
替换策略，见缓存
存储库，69
资源，59
二进制，520
编译器，502
复合材料，516，521
数据库，494，1035 依
赖项，502，516
目录组织，504
出口，499，502
文件格式，507
GUID，507
连接器，502
记忆，511
元数据，494
注册表，508
分段文件，516
源资产，59，495，501
资源获取即初始化，参见设计模式、RA
II
资源依赖性, 222
资源交换文件格式，952
资源管理器，45，481，493
食人魔，501
运行时，503
神秘海域/TLOU，498
虚幻，496
XNA，501
回应，262
恢复，见牛顿定律
重新定位，参见动画
回邮地址，153
混响，920，922，975
地区，967
坦克，979
RIFF，请参阅资源交换文件
格式
右手定则，371，395
刚体动力学，参见物理学
环形缓冲区，983
RMS，参见均方根
RMW，参见读-修改-写
罗宾汉哈希，参见哈希
Rockstar 高级游戏引擎，33
rodata 段，参见可执行文件
卷，386
均方根，914
绳索模拟，818
RTS，请参阅类型
RTTI，1042
运行周期，734
运行时脚本语言，1135
运行时类型识别，1042

S/PDIF，954
采样，683，另见动画，737，
925, 948
深度转换，981
汇率转换，981
沙盒，参见游戏世界编辑器
保存的游戏，1042，1078
标量，221，362
分散/聚集，546，547
散射，见照明

render state, 691, 692
leak, 691
render target, 664
rendering
audio, *see* audio
billboard, *see* billboard
bump mapping, 699
deferred, 709
displacement mapping, 698
G-buffer, 709
parallax occlusion mapping, 698
relief mapping, 698
rendering engine, 45
graphics device interface, 46
low-level renderer, 46, 47
render packet, 46
scene graph, 47, 622, 693, 697
rendering equation, 622, 649
rendering pipeline
application stage, 668, 687
asset conditioning stage, 668, 671
data transformation, 669
geometry processing stage, 668
GPU pipeline, 672
merging, 676
rasterization stage, 668
stream output, 674
tools stage, 668
triangle set-up, 675
triangle traversal, 675
replacement policy, *see* cache
repository, 69
resource, 59
binary, 520
compiler, 502
composite, 516, 521
database, 494, 1035
dependencies, 502, 516
directory organization, 504
exporting, 499, 502
file formats, 507
GUID, 507
linker, 502
memory, 511
metadata, 494
registry, 508
sectioned files, 516
source asset, 59, 495, 501
resource acquisition is initialization, *see*
design pattern, RAII
resource dependency, 222
resource interchange file format, 952
resource manager, 45, 481, 493
OGRE, 501
runtime, 503
Uncharted/TLOU, 498
Unreal, 496
XNA, 501
response, 262
restitution, *see* Newton's law of
retargeting, *see* animation
return address, 153
reverb, 920, 922, 975
region, 967
tank, 979
RIFF, *see* resource interchange file
format
right-hand rule, 371, 395
rigid body dynamics, *see* physics
ring buffer, 983
RMS, *see* root mean square
RMW, *see* read-modify-write
Robin Hood hashing, *see* hashing
Rockstar Advanced Game Engine, 33
rodata segment, *see* executable file
roll, 386
root mean square, 914
rope simulation, 818
RTS, *see* genre
RTTI, 1042
run cycle, 734
runtime scripting language, 1135
runtime type identification, 1042

S/PDIF, 954
sampling, 683, *see also* animation, 737,
925, 948
depth conversion, 981
rate conversion, 981
Sandbox, *see* game world editor
saved games, 1042, 1078
scalar, 221, 362
scatter/gather, 546, 547
scattering, *see* lighting

SCCS，参见版本控制
场景图，参见渲染引擎
模式，1066
继承，1067
方案，477，797，1144
范围，152，153
文件，152
词汇，153
范围锁，320
尖叫，看看音频引擎
屏幕
长宽比，659，663
映射，663，675
决议，659
屏幕截图，606
脚本语言，58，1040，1134类，1153
数据定义，477，1135
与母语的接口，1144
λ，1144
星期一，1139
多任务处理，1154
面向对象，1152
典当，1141
Python，502，1140
地震 C，11，1138
运行时，1135
UnrealScript，1138
SDK，参见软件开发套件
SECAM，534，542，663
段，请参阅可执行文件选择，221
信号量, 275, 278, 492, 548, 555
二进制，276
分离轴定理，837
分离载体，830
顺序锁，324
顺序编程，204，544
串行计算机，233
序列化，1063
设置关联性，参见缓存
着色器，672
建筑，677
Cg, 680
效果文件，670，682
几何，674
高级着色语言，677
内存访问，678
OpenGL着色器语言，680
通过，682
像素，675
像素着色器，46
登记册，678
语义，680
着色器模型 4.0, 677
技术，682
纹理访问，679
纹理采样器，681
统一声明，680
顶点，629，673
着色器资源表，678
阴影，633，647
阴影语言，351
阴影，48，655，703
联系方式，705
映射，703，704
卷，703
香农-奈奎斯特采样定理，948，949
形状，另见碰撞
球体，408
塑造，参见文本渲染
共享内存，参见并发
共享排他锁，324
船舶建造，请参阅建造配置
随机播放，参见 SIMD 指令
国际单位制，912
侧链输入，990
符号位，参见整数，另见浮点数，134
信号，912，924
线程之间，276
连续时间，925，948
离散时间，925，948
操纵，925
周期性，913，938
加工理论，783，924
信号内核对象，268，276
有符号距离场，717
有效数字，参见浮点数
轮廓边缘，625，689，703

SCCS, *see* version control
scene graph, *see* rendering engine
schema, 1066
inheritance, 1067
Scheme, 477, 797, 1144
scope, 152, 153
file, 152
lexical, 153
scoped lock, 320
Scream, *see* audio engine
screen
aspect ratio, 659, 663
mapping, 663, 675
resolution, 659
screenshot, 606
scripting language, 58, 1040, 1134
class, 1153
data definition, 477, 1135
interface to native language, 1144
lambda, 1144
Lua, 1139
multitasking, 1154
object-oriented, 1152
Pawn, 1141
Python, 502, 1140
Quake C, 11, 1138
runtime, 1135
UnrealScript, 1138
SDK, *see* software development kit
SECAM, 534, 542, 663
segment, *see* executable file
select, 221
semaphore, 275, 278, 492, 548, 555
binary, 276
separating axis theorem, 837
separating vector, 830
sequential lock, 324
sequential programming, 204, 544
serial computer, 233
serialization, 1063
set associativity, *see* cache
shader, 672
architecture, 677
Cg, 680
effect file, 670, 682
geometry, 674
High-Level Shading Language, 677
memory access, 678
OpenGL shader language, 680
pass, 682
pixel, 675
pixel shader, 46
register, 678
semantic, 680
shader model 4.0, 677
technique, 682
texture access, 679
texture sampler, 681
uniform declaration, 680
vertex, 629, 673
shader resource table, 678
shading, 633, 647
shading language, 351
shadow, 48, 655, 703
contact, 705
mapping, 703, 704
volume, 703
Shannon-Nyquist sampling theorem, 948, 949
shape, *see also* collision
sphere, 408
shaping, *see* text rendering
shared memory, *see* concurrency
shared-exclusive lock, 324
ship build, *see* build configuration
shuffle, *see* SIMD instruction
SI units, 912
side-chain input, 990
sign bit, *see* integer, *see also* floating-point, 134
signal, 912, 924
between threads, 276
continuous-time, 925, 948
discrete-time, 925, 948
manipulating, 925
periodic, 913, 938
processing theory, 783, 924
signaled kernel object, 268, 276
signed distance field, 717
significant digits, *see* floating-point
silhouette edge, 625, 689, 703

SIMD，参见单指令多数据
数据
SIMD 单元，参见 GPU
单工，840
SIMT，参见单指令多
线
模拟, 9, 51
基于代理，9，1041
离散事件，1087
游戏类型，请参阅类型
硬实时，10
互动，9，525
物理学，10
实时，9，525，532，1041
软实时，10
颞叶，9
单指令多数据，160, 168, 170, 207, 221, 331, 336,
341, 348, 355, 393, 408, 413, 431, 548, 672, 678, 831
__m128 , 332, 350
AltiVec，336
水平添加，338
指令，334
车道，343
随机播放，341，342
矢量浮点数，336，350
VF32, 139
单指令多线程，209，
331, 348, 355
单指令单数据，207
单步执行，参见调试器
单例，参见设计模式
奇异函数，928
正弦曲线，933
SISD，参见单指令单数据
骨骼动画，参见动画
骨架，670
在记忆中，728
SketchUp，669
剥皮，62，670，750
至多个关节，754
重量，635，725，750
天空，713
盒子，714
圆顶，714
小内存分配器，1076
智能指针，440，1042，1079，1080
平滑，639
SMP，参见对称多处理
快照，1112
SoC，参见片上系统
插座，见动画
Softimage/XSI，62，669
软件开发套件，40
软件对象模型，参见对象模型
立体角，963
排序，443
用于渲染，692
声音，另见音频，912
2D, 955
3D, 955
银行，98
7 卡，974
夹子，955，985，998
提示，986
干燥，920
压力下降，917，923，957
门户网站，972
压力，912，914
压力水平，915，961
辐射模式，918
合成，955，975，978，995
波浪，913
湿的，921
Sound Forge，63
源引擎, 33
源文件，参见 C++
SourceSafe，请参阅版本控制
空间哈希，693，846，1086
空间分区，693
空间化，参见音频
产卵器，1065
优点和缺点，1069
扬声器，943
演讲者圈子，959
光谱图，633
Spectre 漏洞，239
镜面反射，参见照明
镜面照明，参见照明
推测执行，218，参见 CPU
演讲，997
速度，535 声音，
913

SIMD, *see* single instruction multiple
data
SIMD unit, *see* GPU
simplex, 840
SIMT, *see* single instruction multiple
thread
simulation, 9, 51
agent-based, 9, 1041
discrete event, 1087
game genre, *see* genre
hard real-time, 10
interactive, 9, 525
physics, 10
real-time, 9, 525, 532, 1041
soft real-time, 10
temporal, 9
single instruction multiple data, 160,
168, 170, 207, 221, 331, 336,
341, 348, 355, 393, 408, 413,
431, 548, 672, 678, 831
`__m128`, 332, 350
AltiVec, 336
horizontal add, 338
instruction, 334
lanes, 343
shuffle, 341, 342
`vector float`, 336, 350
`VF32`, 139
single instruction multiple thread, 209,
331, 348, 355
single instruction single data, 207
single-step, *see* debugger
singleton, *see* design pattern
singularity function, 928
sinusoid, 933
SISD, *see* single instruction single data
skeletal animation, *see* animation
skeleton, 670
in memory, 728
SketchUp, 669
skinning, 62, 670, 750
to multiple joints, 754
weights, 635, 725, 750
sky, 713
box, 714
dome, 714
small memory allocator, 1076
smart pointer, 440, 1042, 1079, 1080
smoothing, 639
SMP, *see* symmetric multiprocessing
snapshot, 1112
SoC, *see* system on a chip
socket, *see* animation
Softimage/XSI, 62, 669
software development kit, 40
software object model, *see* object model
solid angle, 963
sorting, 443
for rendering, 692
sound, *see also* audio, 912
2D, 955
3D, 955
bank, 987
card, 974
clip, 955, 985, 998
cue, 986
dry, 920
fall-off of pressure, 917, 923, 957
portals, 972
pressure, 912, 914
pressure level, 915, 961
radiation patterns, 918
synthesis, 955, 975, 978, 995
wave, 913
wet, 921
Sound Forge, 63
Source engine, 33
source file, *see* C++
SourceSafe, *see* version control
spatial hash, 693, 846, 1086
spatial partitioning, 693
spatialization, *see* audio
spawner, 1065
pros and cons, 1069
speakers, 943
speaker circle, 959
spectral plot, 633
Spectre exploit, 239
specular, *see* lighting
specular lighting, *see* lighting
speculative execution, 218, *see* CPU
speech, 997
speed, 535
of sound, 913

球体层次结构，参见树形球体图
，700 球面坐标，360 谐波基函数
，708 自旋锁，271、290、294、55
5 自旋等待，参见忙等待 SPL，参
见声音、压力水平 样条线，624、
1033 B 样条线，624、783

贝塞尔曲线，624分屏
，995，997聚光灯，
见照明弹簧，884 SPU
，545
SQL 服务器，498
SQT变换，405
SRT，另请参阅着色器资源表变换
，405，729 SSE，请参阅流式 SIMD
扩展编译器内在函数，333 稳定性（
数值），863 堆栈（数据结构），44
2 堆栈帧，请参阅调用堆栈阶段，请
参阅 CPU，阶段停顿，215 姿态变
化，772 标准 C 库，请参阅 C 标准

图书馆
标准 C++ 库，参见 C++ 标准模板库
、40、41、442、
448 std::list , 41 std::string , 457, 4
65 std::vector , 41 启动和关闭按
需构造, 419 引擎子系统, 417 手
动, 420 OGRE, 422

神秘海域/TLOU，424 个
饥饿，284 个状态缓存，1
100 个静态
分配，请参阅分配断言
，请参阅断言灯，请参
阅照明
变量，149，157 模板
缓冲区，664，703 测
试，676 立体声，944
毒刺，1011
STL，参见标准模板库随机传播建模
，972 策略游戏，参见流派流派

音频流，988 对话流，998 级别
流，1040 流式 SIMD 扩展，331
严格别名，143 字符串，456 类
，457，465 数据库，467 字符串
ID，459，1138，1151 条带，参
见三角形数组结构，1060 结构化
绑定，115 工作室，5

第一方开发者，8 细分表面，624 亚
正规化，参见浮点 Subversion，参见
版本控制 低音炮，参见低频效果 叠
加，919，926 超标量，参见 CPU 表
面，622，623 视觉属性，622，632
环绕声，944 扫描和修剪算法，847 S
WIFT，52 符号链接，494 对称多处
理，227 同步点，1106，1113 协同处
理单元，参见 SPU 语法树，792 合成
器，995

系统，参见线性时不变系统片上系统，6
78

T&L，参见硬件 T&L 表（数据
结构），1139 标记图像文件格
式，642
塔尔加，642

sphere hierarchy, *see* tree
sphere map, 700
spherical
    coordinates, 360
    harmonic basis function, 708
spin lock, 271, 290, 294, 555
spin-wait, *see* busy-wait
SPL, *see* sound, pressure level
spline, 624, 1033
    B-spline, 624, 783
    Bézier, 624
split-screen, 995, 997
spot light, *see* lighting
spring, 884
SPU, 545
SQL Server, 498
SQT transform, 405
SRT, *see also* shader resource table
    transform, 405, 729
SSE, *see* streaming SIMD extensions
    compiler intrinsics, 333
stability (numerical), 863
stack (data structure), 442
stack frame, *see* call stack
stage, *see* CPU, stage
stall, 215
stance variation, 772
standard C library, *see* C standard
        library
standard C++ library, *see* C++
standard template library, 40, 41, 442,
        448
    `std::list`, 41
    `std::string`, 457, 465
    `std::vector`, 41
start-up and shut-down
    construct on demand, 419
    engine subsystem, 417
    manual, 420
    OGRE, 422
    Uncharted/TLOU, 424
starvation, 284
state caching, 1100
static
    allocation, *see* allocation
    assertion, *see* assertion
    lights, *see* lighting
    variable, 149, 157
stencil buffer, 664, 703
    testing, 676
stereo, 944
stinger, 1011
STL, *see* standard template library
stocastic propagation modeling, 972
strategy game, *see* genre
streaming
    audio streaming, 988
    dialog streaming, 998
    level streaming, 1040
streaming SIMD extensions, 331
strict aliasing, 143
string, 456
    class, 457, 465
    database, 467
string id, 459, 1138, 1151
strip, *see* triangle
struct of arrays, 1060
structured bindings, 115
studio, 5
    first-party developer, 8
subdivision surface, 624
subnormal, *see* floating-point
Subversion, *see* version control
subwoofer, *see* low-frequency effects
superposition, 919, 926
superscalar, *see* CPU
surface, 622, 623
    visual properties, 622, 632
surround sound, 944
sweep and prune algorithm, 847
SWIFT, 52
symbolic link, 494
symmetric multiprocessing, 227
synchronization point, 1106, 1113
synergistic processing unit, *see* SPU
syntax tree, 792
synthesizer, 995
system, *see* linear time-invariant system
system on a chip, 678

T&L, *see* hardware T&L
table (data structure), 1139
Tagged Image File Format, 642
Targa, 642

目标硬件, 38 目标管
理器, 590
任务分解，544 任务并行，参见
并行性
分类学，1048
撕裂，537，538，663
技术要求清单，587
模板元编程, 118
时间相干性，537，846
时间多线程，234
临时寄存器，参见 GPU
地形，714
　高度场，60，714，1029
镶嵌，625，674
　动态，626，714
测试和设置指令，294
可测试性，87
TeX，1136
特克塞尔，639
　密度，643，644
文本渲染，另见有符号距离场，716
　塑造，717
文本段，参见可执行文件
纹理, 46, 121, 639, 670, 678, 679
1D, 641
3D, 701
寻址模式，641
反照率图，640
动画，52，723
压缩，642
坐标，641
立方体贴图，700
漫反射贴图，640
环境贴图，48，700，706
过滤，645，679
格式，642
光泽图，699
高度图，698
光照贴图，48，653，672
法线贴图，698
渲染至，680，706
滚动，691
阴影贴图，703
镜面贴图，699
镜面反射功率图，699
球体地图，700
线程，204，另请参阅 GPU
　亲和力，227，248 依赖
　性，548，1105
　组，请参阅 GPU
　加入，243，554，557
　优先级，248
　安全，288，321，1102
　同步原语，265，267，555
　用户级，253，1102，1110
吞吐量，参见管道
音色，920
时间
　抽象时间轴，532
　时钟，532
　时钟漂移，540
　时钟变量，540
　三角洲，535 – 537，572
　域，924，938，974
　浮点数，541
　游戏，525，532
　高分辨率定时器，539
　索引，735
　本地，533
　测量，539，543
　影响，845
　缩放，739
　转变，925
　单位，540
时不变，926
时间切片，另见多任务处理，208，
　234, 356, 545
时间线
　全球，739
　本地，735
TLB，翻译后备缓冲区
TOI，查看影响时间
Tokamak, 825
墓碑发动机，35
色调映射，702
工具，59，61，669
工具管道，请参阅资产调节管道
扭矩，373，867
　三维空间，873
扭矩发动机，36
托斯林克，954

target hardware, 38
Target Manager, 590
task decomposition, 544
task parallelism, *see* parallelism
taxonomy, 1048
tearing, 537, 538, 663
technical requirements checklist, 587
template metaprogramming, 118
temporal coherency, 537, 846
temporal multithreading, 234
temporary register, *see* GPU
terrain, 714
　height field, 60, 714, 1029
tessellation, 625, 674
　dynamic, 626, 714
test-and-set instruction, 294
testability, 87
TeX, 1136
texel, 639
　density, 643, 644
text rendering, *see also* signed distance
　field, 716
　shaping, 717
text segment, *see* executable file
texture, 46, 121, 639, 670, 678, 679
1D, 641
3D, 701
addressing modes, 641
albedo map, 640
animated, 52, 723
compression, 642
coordinates, 641
cube map, 700
diffuse map, 640
environment map, 48, 700, 706
filtering, 645, 679
formats, 642
gloss map, 699
heightmap, 698
light map, 48, 653, 672
normal map, 698
rendering to, 680, 706
scrolling, 691
shadow map, 703
specular map, 699
specular power map, 699
sphere map, 700
thread, 204, *see also* GPU
　affinity, 227, 248
　dependencies, 548, 1105
　group, *see* GPU
　join, 243, 554, 557
　priority, 248
　safety, 288, 321, 1102
　synchronization primitive, 265,
　267, 555
　user-level, 253, 1102, 1110
throughput, *see* pipeline
timbre, 920
time
　abstract timeline, 532
　clock, 532
　clock drift, 540
　clock variable, 540
　delta, 535–537, 572
　domain, 924, 938, 974
　floating-point, 541
　game, 525, 532
　high-resolution timer, 539
　index, 735
　local, 533
　measuring, 539, 543
　of impact, 845
　scaling, 739
　shift, 925
　units, 540
time-invariant, 926
time-slicing, *see also* multitasking, 208,
　234, 356, 545
timeline
　global, 739
　local, 735
TLB, *see* translation lookaside buffer
TOI, *see* time of impact
Tokamak, 825
Tombstone Engine, 35
tone mapping, 702
tool, 59, 61, 669
tools pipeline, *see* asset conditioning
　pipeline
torque, 373, 867
　in three dimensions, 873
Torque Engine, 36
TOSLINK, 954

事务，287，328 提交，328 转换，376
和照明，672 坐标轴，384，390 矩阵，
376 旋转，376，382，403 缩放，364，
376，383 平移，376，382 晶体管，946
过渡，请参阅动画翻译单元，请参阅 C
++，148 半透明，623 透射，请参阅照
明透明度，623 传输模型，请参阅照明
转置矩阵，339 树，442，另请参阅混
合树二叉空间分区树，47，695 边界球
，695，846 k d 树，47，695 八叉树，4
7，694 四叉树，47，694 球体层次结构
，47 三角形，625 面积，371 面法线，
627 扇形，629，630 索引列表，628 列
表，628，630 网格，参见网格遮挡，6
64 条带，629，630 缠绕顺序，627 三角
剖分，625 三线性，参见纹理，过滤 T
RS 插孔，参见音频 TrueAxis，824，82
5 TTY，590 隧道，843 涡轮按钮，536
回合制策略，参见类型 二进制补码，
参见整数类型双关，143

UAA，参见通用音频架构 UDN，11，
参见虚幻开发者网络 ULP，参见浮点本
影，655 UNC，483 Unicode，462 UTF-1
6，463 UTF-32，463 UTF-8，463 统一建
模语言，107 单位脉冲，928 单位在最后
，参见浮点 Unity，35

通用音频架构，992 通用串行总线，请
参阅 USB 虚幻开发者网络，33 虚幻引
擎，11、32、35、45，另请参阅音频引
擎蓝图，32、1133 虚幻竞技场 2004，3
2 UnrealEd，496、1035，请参阅游戏
世界编辑器 UnrealScript，1138 未解析
的符号错误，请参阅编译器，146 USB
，570、954 用户定义文字，请参阅 C+
+ 用户级线程，请参阅线程 UTFx ，请
参阅 Unicode V-Collide，52 VAG，请
参阅音频、文件格式 Valgrind，50、10
2 变量类静态，157 全局，151 本地，1
53 成员，156 静态，151 变体，1008、1
118、1145 矢量，362，另请参阅垂直
加法， 364 算术, 365 基, 363, 390 共线,
369 叉积, 367, 370

transaction, 287, 328
committing, 328
transformation, 376
and lighting, 672
coordinate axes, 384, 390
matrix, 376
rotation, 376, 382, 403
scale, 364, 376, 383
translation, 376, 382
transistor, 946
transition, *see* animation
translation unit, *see* C++, 148
translucency, 623
transmission, *see* lighting
transparency, 623
transport model, *see* lighting
transpose
matrix, 339
tree, 442, *see also* blend tree
binary space partitioning tree, 47, 695
bounding sphere, 695, 846
*k*d-tree, 47, 695
octree, 47, 694
quadtree, 47, 694
sphere hierarchy, 47
triangle, 625
area of, 371
face normal, 627
fan, 629, 630
indexed list, 628
list, 628, 630
mesh, *see* mesh
occlusion, 664
strip, 629, 630
winding order, 627
triangulation, 625
trilinear, *see* texture, filtering
TRS jack, *see* audio
TrueAxis, 824, 825
TTY, 590
tunneling, 843
turbo button, 536
turn-based strategy, *see* genre
two's complement, *see* integer
type punning, 143

UAA, *see* Universal Audio Architecture
UDN, 11, *see* Unreal Developer Network
ULP, *see* floating-point
umbra, 655
UNC, 483
Unicode, 462
UTF-16, 463
UTF-32, 463
UTF-8, 463
Unified Modeling Language, 107
unit impulse, 928
unit in the last place, *see* floating-point
Unity, 35
Universal Audio Architecture, 992
universal serial bus, *see* USB
Unreal Developer Network, 33
Unreal Engine, 11, 32, 35, 45, *see also* audio engine
Blueprints, 32, 1133
Unreal Tournament 2004, 32
UnrealEd, 496, 1035, *see* game world editor
UnrealScript, 1138
unresolved symbol error, *see* compiler, 146
USB, 570, 954
user-defined literals, *see* C++
user-level thread, *see* thread
UTF-*x*, *see* Unicode

V-Collide, 52
VAG, *see* audio, file formats
Valgrind, 50, 102
variable
class static, 157
global, 151
local, 153
member, 156
static, 151
variant, 1008, 1118, 1145
vector, 362, *see also* perpendicular
addition, 364
arithmetic, 365
basis, 363, 390
collinear, 369
cross product, 367, 370

方向，363，381
点积，367
前左上，385，631
震级，365
乘以标量，364
正常，367，369，392，638
规范化，367
垂直，635
位置，363
投影，368
四元数形式，397
平方幅度，367
STL，442
减法，365
单元，363，367
矢量处理单元，168，208，224，参见还有 GPU、SIMD 单元、350、672
矢量单元（PS2），224
矢量化，168，337，344，348，548
车辆发动机声音，996
速度，366，528，535
角度，867
动画，758
线性，858
亲戚，923，957，973
屏幕空间，719
版本控制, 69
异形大脑，70，494
资产，493
入住和退房，75
ClearCase，70
承诺，75
CVS，70
删除文件, 78
差异，75
专属退房，76
git，70
历史，74
托管，72
锁定，76
合并，76
多次签出，76 Perforce
，70，493 RCS，70
重新定基，70
存储库，72
资源，493
SCCS，70
SourceSafe，70
提交，75
颠覆，70
三路合并，76，102
TortoiseSVN，73
更新, 74
顶点，635
属性插值，637，662，675
属性，635
双常态，635
双切线，635
缓存优化，629，679
格式，636
正常，635
切线，635
顶点缓冲区，628
垂直消隐间隔，538，663
查看量，658，674
远平面，658
件，411，658，674
近平面，658
观察方向，看灯光
视口，46
小插图，719
虚拟机, 317, 1135, 1144
虚拟现实，27，35
能见度测定，47，157
Visual Basic，1134
视觉效果，48，64
VLIW，参见CPU
语音，参见音频
网络电话，570
挥发性 ，302，318，326
音量控制，946，979，983
巫毒教，672
VPU，参见矢量处理单元
VR，见识虚拟现实
VTune，50
VU0（PS2），224
伏尔甘，41，46，692

w 缓冲区，664，666，704
无等待算法，290
无等待共识问题，参见共识问题

direction, 363, 381
dot product, 367
front, left and up, 385, 631
magnitude, 365
multiplication by scalar, 364
normal, 367, 369, 392, 638
normalization, 367
perpendicular, 635
position, 363
projection, 368
quaternion form, 397
squared magnitude, 367
STL, 442
subtraction, 365
unit, 363, 367
vector processing unit, 168, 208, 224, *see also* GPU, SIMD unit, 350, 672
vector unit (PS2), 224
vectorization, 168, 337, 344, 348, 548
vehicle engine sound, 996
velocity, 366, 528, 535
angular, 867
animation, 758
linear, 858
relative, 923, 957, 973
screen space, 719
version control, 69
Alienbrain, 70, 494
assets, 493
checking in and out, 75
ClearCase, 70
committing, 75
CVS, 70
deleting files, 78
diff, 75
exclusive check-out, 76
git, 70
history, 74
hosting, 72
locking, 76
merging, 76
multiple check-out, 76
Perforce, 70, 493
RCS, 70
rebasing, 70
repository, 72
resources, 493
SCCS, 70
SourceSafe, 70
submitting, 75
Subversion, 70
three-way merge, 76, 102
TortoiseSVN, 73
updating, 74
vertex, 635
attribute interpolation, 637, 662, 675
attributes, 635
binormal, 635
bitangent, 635
cache optimization, 629, 679
formats, 636
normal, 635
tangent, 635
vertex buffer, 628
vertical blanking interval, 538, 663
view volume, 658, 674
far plane, 658
frustum, 411, 658, 674
near plane, 658
viewing direction, *see* lighting
viewport, 46
vignette, 719
virtual machine, 317, 1135, 1144
virtual reality, 27, 35
visibility determination, 47, 157
Visual Basic, 1134
visual effects, 48, 64
VLIW, *see* CPU
voice, *see* audio
VoIP, 570
`volatile`, 302, 318, 326
volume control, 946, 979, 983
Voodoo, 672
VPU, *see* vector processing unit
VR, *see* virtual reality
VTune, 50
VU0 (PS2), 224
Vulkan, 41, 46, 692

w buffer, 664, 666, 704
wait-free algorithm, 290
wait-free consensus problem, *see* consensus problem

步行循环，734 扭曲，参见 GPU WAS
API，参见 Windows 音频会话 API 水
，626、632、647、648、673、706、71
0、715、910、1017、1029、1044 WAV
，参见音频，文件格式 波纵波，917
传播，917 横波，917 波前，参见 GPU
波长，633，另请参阅照明，914 WD
M，参见 Windows 驱动程序模型加权
平均值，375、635、676、765、788 湿
声，参见声音 整数，132 Wiimote，53
Windows 音频会话 API，993 位图，64
2 驱动程序模型，992 媒体视频 (WMV)
，1162 WMA，参见音频，文件格式 W
MV，1162 低音扬声器，参见低频效果
工作，875世界，参见游戏世界世界空
间纹素密度，644写入

后退，196，另请参阅 CPU，阶段，306
，307，309 释放，311，312，319 通过
，196，306 Wwise，995 X3DAudio，请
参阅音频引擎 XACT，请参阅音频引擎
XAudio2，请参阅音频引擎 Xbox Live
，34，473，481，487 XML，1136 XNA
Game Studio，34，993 Yake，36 偏航
，386 屈服

协程，253，554，1110 光纤，250，2
51，554 线程的时间片，234，241，2
44，246，250 z 偏差，712 z 缓冲区，
704 z 战斗，666，712 ZBrush，62 ZIP
存档，请参阅存档文件

walk cycle, 734
warp, *see* GPU
WASAPI, *see* Windows Audio Session API
water, 626, 632, 647, 648, 673, 706, 710, 715, 910, 1017, 1029, 1044
WAV, *see* audio, file formats
wave
  longitudinal, 917
  propagation, 917
  transverse, 917
wavefront, *see* GPU
wavelength, 633, *see also* lighting, 914
WDM, *see* Windows Driver Model
weighted average, 375, 635, 676, 765, 788
wet sound, *see* sound
whole number, 132
Wiimote, 53
Windows
  Audio Session API, 993
  Bitmap, 642
  Driver Model, 992
  Media Video (WMV), 1162
WMA, *see* audio, file formats
WMV, 1162
woofer, *see* low-frequency effects
work, 875
world, *see* game world
world-space texel density, 644
write
  back, 196, *see also* CPU, stage, 306, 307, 309
  release, 311, 312, 319
  through, 196, 306
Wwise, 995

X3DAudio, *see* audio engine
XACT, *see* audio engine
XAudio2, *see* audio engine
Xbox Live, 34, 473, 481, 487
XML, 1136
XNA Game Studio, 34, 993

Yake, 36
yaw, 386
yielding
  coroutine, 253, 554, 1110
  fiber, 250, 251, 554
  thread's timeslice, 234, 241, 244, 246, 250

z bias, 712
z buffer, 704
z fighting, 666, 712
ZBrush, 62
ZIP archive, *see* archive file

图一：暴雪娱乐出品的《守望先锋》（Xbox One、PlayStation 4、Windows）。（参见第14页图1.2。）

Plate I. *Overwatch* by Blizzard Entertainment (Xbox One, PlayStation 4, Windows). (See Figure 1.2 on page 14.)

图二。顽皮狗的《杰克二世》（《杰克、达斯特》、《杰克与达斯特》和《杰克二世》© 2003、2013/TM SIE）。由顽皮狗创作和开发 Dog、PlayStation 2）。（参见第 16 页的图 1.3。）

Plate II. *Jak II* by Naughty Dog (Jak, Daxter, Jak and Daxter, and Jak II © 2003, 2013/™ SIE. Created and developed by Naughty Dog, PlayStation 2). (See Figure 1.3 on page 16.)

图三。The Coalition 出品的《战争机器 4》（Xbox One）。（参见第 17 页图 1.4。）

Plate III. *Gears of War 4* by The Coalition (Xbox One). (See Figure 1.4 on page 17.)

图四。Namco 出品的《铁拳 3》（PlayStation）。（参见第 18 页图 1.5。）

Plate IV. *Tekken 3* by Namco (PlayStation). (See Figure 1.5 on page 18.)

NetherRealm Studios 出品的《Plate V. Injustice 2》（PlayStation 4、Xbox One、安卓、iOS、Microsoft Windows）。（参见第 19 页图 1.6。）

Plate V. *Injustice 2* by NetherRealm Studios (PlayStation 4, Xbox One, Android, iOS, Microsoft Windows). (See Figure 1.6 on page 19.)

图六。Polyphony Digital 出品的《Gran Turismo Sport》（PlayStation 4）。（参见第 20 页图 1.7。）

Plate VI. *Gran Turismo Sport* by Polyphony Digital (PlayStation 4). (See Figure 1.7 on page 20.)

图 VII。Ensemble Studios 出品的《帝国时代》（Windows 系统）。（参见第 21 页图 1.8。）

Plate VII. *Age of Empires* by Ensemble Studios (Windows). (See Figure 1.8 on page 21.)

图八。Creative Assembly 出品的《全面战争：战锤 2》（Windows 系统）。（参见第 22 页图 1.9。）

Plate VIII. *Total War: Warhammer 2* by Creative Assembly (Windows). (See Figure 1.9 on page 22.)

图 IX. 暴雪娱乐出品的《魔兽世界》（Windows、MacOS）。（参见第 23 页图 1.10。）

Plate IX. *World of Warcraft* by Blizzard Entertainment (Windows, MacOS). (See Figure 1.10 on page 23.)

板X。《命运2》由Bungie出品，© 2018 Bungie Inc.（Xbox One、PlayStation 4、PC）。（参见第24页图1.11。）

Plate X. *Destiny* 2 by Bungie, © 2018 Bungie Inc. (Xbox One, PlayStation 4, PC). (See Figure 1.11 on page 24.)

图 XI。LittleBigPlanetTM 2 由 Media Molecule 出品，© 2014 Sony Interactive Entertainment (PlayStation 3)。（参见图 1.12 第 25 页。

Plate XI. *LittleBigPlanet™ 2* by Media Molecule, © 2014 Sony Interactive Entertainment (PlayStation 3). (See Figure 1.12 on page 25.)

图十二。《梦境》，Media Molecule 作品，© 2017 Sony Interactive Entertainment（PlayStation 4）。（参见第 26 页图 1.13。）

Plate XII. *Dreams* by Media Molecule, © 2017 Sony Interactive Entertainment (PlayStation 4). (See Figure 1.13 on page 26.)

图十三。Markus “Notch” Persson / Mojang AB 的 Minecraft（Windows、MacOS、Xbox 360、PlayStation 3、PlayStation Vita、iOS）。
（参见第 26 页的图 1.14。）

Plate XIII. *Minecraft* by Markus “Notch” Persson / Mojang AB (Windows, MacOS, Xbox 360, PlayStation 3, PlayStation Vita, iOS).
(See Figure 1.14 on page 26.)

图十四。Squanchtendo 和 Crows Crows Crows (HTC Vive) 的《会计》。（参见第 29 页图 1.15。）

Plate XIV. *Accounting* by Squanchtendo and Crows Crows Crows (HTC Vive). (See Figure 1.15 on page 29.)

图十五。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的一个场景，渲染时未使用纹理。（参见第 647 页的图 11.20。）

Plate XV. A scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) rendered without textures. (See Figure 11.20 on page 647.)

图 16。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的同一场景，仅应用了漫反射纹理。（参见第 648 页的图 11.21。）

Plate XVI. The same scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) with only diffuse textures applied. (See Figure 11.21 on page 648.)

图十七。《最后生还者：重制版》场景（© 2014/TM SIE。顽皮狗创作并开发，PlayStation 4平台），全光照。（参见第648页图11.22。）

Plate XVII. Scene from *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) with full lighting. (See Figure 11.22 on page 648.)

图十八。任天堂（Wii）游戏《路易吉洋楼》中的手电筒由多种视觉效果组成，包括用于光束的半透明几何锥形、用于将光线投射到场景中的动态聚光灯、镜头上的自发光纹理以及用于产生镜头光晕的面向相机的卡片。（参见第 656 页的图 11.30。）

Plate XVIII. The flashlight in *Luigi's Mansion* by Nintendo (Wii) is composed of numerous visual effects, including a cone of translucent geometry for the beam, a dynamic spot light to cast light into the scene, an emissive texture on the lens and camera-facing cards for the lens flare. (See Figure 11.30 on page 656.)

图 XIX。左：无抗锯齿。中：4 × MSAA。右：Nvidia 的 FXAA，预设 3。图片来自 Nvidia 的 FXAA 白皮书，作者：蒂莫西 · 洛特斯（http://bit.ly/1mIzCTv）。（参见第 683 页图 11.45。）

Plate XIX. Left: No antialiasing. Center: 4× MSAA. Right: Nvidia's FXAA, preset 3. Image from Nvidia's FXAA white paper by Timothy Lottes (http://bit.ly/1mIzCTv). (See Figure 11.45 on page 683.)

图 XX。这张来自 EA 的《Fight Night Round 3》的截图展示了如何使用光泽贴图来控制应用于表面每个纹素的镜面反射程度。（参见第 701 页的图 11.55。）

Plate XX. This screenshot from EA's *Fight Night Round 3* shows how a gloss map can be used to control the degree of specular reflection that should be applied to each texel of a surface. (See Figure 11.55 on page 701.)

图二十一。《最后生还者：重制版》（© 2014/TM SIE。由顽皮狗创作和开发，PlayStation 4）中的镜面反射效果，通过将场景渲染到纹理上，然后应用到镜面表面来实现。（参见第 707 页的图 11.59。）

Plate XXI. Mirror reflections in *The Last of Us: Remastered* (© 2014/™ SIE. Created and developed by Naughty Dog, PlayStation 4) implemented by rendering the scene to a texture that is subsequently applied to the mirror's surface. (See Figure 11.59 on page 707.)

图 XXII。Guerrilla Games 出品的《杀戮地带 2》的截图，展示了延迟渲染中使用的 G 缓冲区的一些典型组件。上图显示最终渲染的图像。下方从左上角开始顺时针依次为反照率（漫反射）颜色、深度、视图空间法线、屏幕空间二维运动矢量（用于运动模糊）、镜面反射功率和镜面反射强度。（参见第 710 页的图 11.62。）

Plate XXII. Screenshots from *Killzone 2* by Guerrilla Games, showing some of the typical components of the G-buffer used in deferred rendering. The upper image shows the final rendered image. Below it, clockwise from the upper left, are the albedo (diffuse) color, depth, view-space normal, screen-space 2D motion vector (for motion blurring), specular power and specular intensity. (See Figure 11.62 on page 710.)

图二十三。《神秘海域3：德雷克的诡计》中的火焰、烟雾和子弹曳光粒子效果（© 2011/TM SIE。由顽皮狗创作和开发，PlayStation 3）。（参见第712页图11.63。）

Plate XXIII. Flame, smoke and bullet tracer particle effects in *Uncharted 3: Drake's Deception* (© 2011/™ SIE. Created and developed by Naughty Dog, PlayStation 3). (See Figure 11.63 on page 712.)

图 XXIV。《神秘海域 3：德雷克的诡计》中的视差贴花（© 2011/TM SIE。由 Naughty 创建和开发）Dog，PlayStation 3）。（参见第 713 页的图 11.64。）

Plate XXIV. Parallax-mapped decals from *Uncharted 3: Drake's Deception* (© 2011/™ SIE. Created and developed by Naughty Dog, PlayStation 3). (See Figure 11.64 on page 713.)

图 XXV。CRYENGINE 的沙盒编辑器。（参见第 1028 页的图 15.6。）

Plate XXV. The Sandbox editor for CRYENGINE. (See Figure 15.6 on page 1028.)

图 XXVI。虚幻引擎 4 动画蓝图编辑器。（参见第 801 页的图 12.58。）

Plate XXVI. The Unreal Engine 4 animation Blueprints editor. (See Figure 12.58 on page 801.)