ELSEVIER

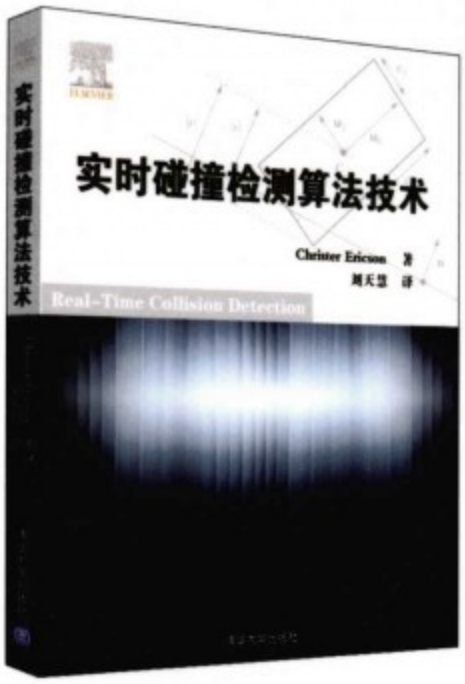
实时碰撞检测算法技术

Christer Ericson 著

刘天慧 译

Real-Time Collision Detection

清华大学出版社

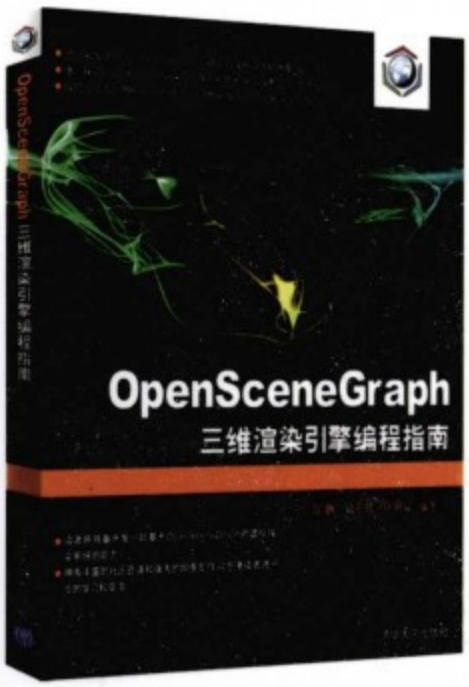


实时碰撞检测算法技术

ISBN:978-7-302-22411-2 定价:52.00元

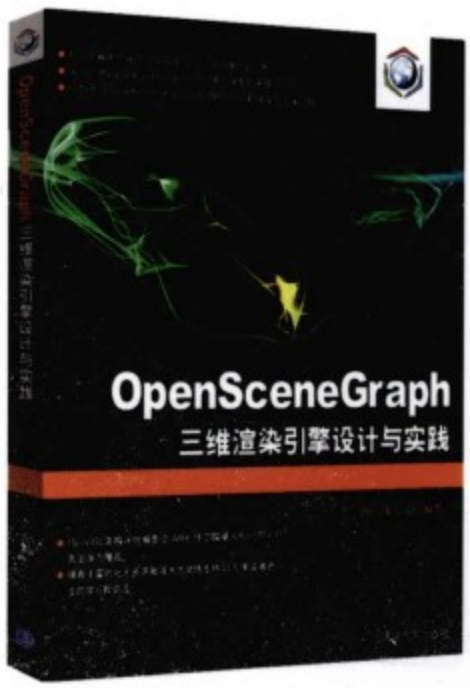
光线跟踪算法技术

即将出版



OpenSceneGraph三维渲染引擎编程指南

ISBN:978-7-302-21303-1 定价:49.80元



OpenSceneGraph三维渲染引擎设计与实践

ISBN:978-7-302-21223-2 定价:45.00元

Real-Time Collision Detection

ELSEVIER

ISBN 978-7-302-22411-2
9 787302 224112 >
定价：52.00元

实时碰撞检测算法技术

Christer Ericson 著

刘天慧 译

清华大学出版社

北 京

内 容 简 介

本书详细阐述了与碰撞检测问题相关的高效解决方案及相应的数据结构和算法，主要包括：碰撞检测系统中的设计问题、数学和几何学入门、包围体、基本图元测试、层次包围体技术、空间划分、BSP 树层次结构、凸体算法、基于 GPU 的碰撞检测、数值健壮性、几何健壮性以及优化操作。另外，本书还提供了相应的算法、代码以及伪代码，以帮助读者进一步理解计算方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。

Real-Time Collision Detection, 1e
Christer Ericson
ISBN: 9789812724120
Copyright © 2005 by Elsevier. All rights reserved.
Authorized Simplified Chinese translation edition published by Elsevier (Singapore) Pte Ltd Press and Tsinghua University Press.
ISBN: 978-1-55860-732-3
Copyright © 2010 by Elsevier (Singapore) Pte Ltd and Tsinghua University Press. All rights reserved.
Published in China by Tsinghua University Press under special arrangement with Elsevier (Singapore) Pte Ltd.. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.
本书简体中文版由 Elsevier (Singapore) Pte Ltd.授予清华大学出版社在中国大陆地区（不包括香港、澳门特别行政区以及台湾地区）出版与发行。未经许可之出口，视为违反著作权法，将受法律之制裁。

本书封底贴有 Elsevier 防伪标签，无标签者不得销售。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

实时碰撞检测算法技术/（美）埃里克森（Ericson，C.）著；刘天慧译．—北京：清华大学出版社，2010.6
书名原文：Real-Time Collision Detection

ISBN 978-7-302-22411-2

I. ①实…　II. ①埃…　②刘…　III. ①计算机仿真　IV. ①TP391.9

中国版本图书馆 CIP 数据核字（2010）第 065078 号

责任编辑：熊　健　郭　伟
封面设计：刘　超
版式设计：牛瑞瑞
责任校对：王　云
责任印制：李红英
出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市兴旺装订有限公司
经　销：全国新华书店
开　本：203×260　**印　张**：26.5　**字　数**：707 千字
版　次：2010 年 6 月第 1 版　　**印　次**：2010 年 6 月第1 次印刷
印　数：1～4000
定　价：52.00 元

产品编号：033422-01

译 者 序

目前，3D 游戏已经成为计算机游戏领域的主流，但不可否认的是，其复杂度也随之上升。相应地，在处理这类较大的数据集时，与碰撞检测相关的数据结构和算法也变得日趋复杂。无论是游戏，还是其他类型的模拟仿真应用程序，碰撞检测始终是程序开发的核心之一。有时，甚至可以用碰撞检测作为衡量游戏引擎是否完善的标准。因此，碰撞检测是让很多开发者棘手的一个难点，不仅算法复杂，而且容易出错。

针对这一问题，本书详细阐述了与碰撞检测问题相关的高效解决方案及其相应的数据结构和算法。碰撞检测问题涵盖了丰富的内容，主要包括：包围体、基本图元测试、层次包围体技术、空间划分、BSP 树层次结构、凸体算法、基于 GPU 的碰撞检测、数值健壮性、几何健壮性以及优化操作。值得一提的是，本书并非是一本纯理论书籍，除了对相关内容进行了全面、系统的讲解以外，其设计思想、数据结构和算法均辅以对应的代码示例，以帮助读者进一步理解计算方案的实现过程。

本书深入碰撞检测这一核心问题的内部，内容涉及数学、物理、数据结构、算法、计算机图形学、计算机体系结构以及优化操作等知识。这里是我们具体应用所学知识的地方，不再有 API 调用，一切都是真实的——这的确令人激动！正如本书作者所言：作为一名读者，此刻，你值得拥有！

在本书的翻译过程中，除刘天慧外，刘鹏、张博、刘璋、刘祎、周建娟、李莉、马琳琳、祁宏亮、付钰、李素梅等也参与了部分翻译和校对工作。

限于译者的水平，译文中难免有错误和不妥之处，恳请广大读者批评指正。

译　　者

前　言

Preface

1978 年，孩提时代的我就和朋友编写开发了第一款属于我们自己的计算机游戏——我仍然清楚地记得，那是一款采用 BASIC 语言编写的猜谜类游戏，玩家将被问及与非洲动物相关的一些问题。恰巧，“太空入侵者”游戏也于同年发布。诚然，与“太空入侵者”相比，我们的作品多少有些黯然失色，但大家仍沉湎于其中，乐此不疲。不久，我们即在自己的家用计算机上开始模仿“太空入侵者”以及当年流行的街机游戏，并设计自己的原创游戏。时光荏苒，岁月蹉跎，昔日对游戏设计的爱好引领我成为一名业界中人，而游戏领域业已发展壮大为一项价值达数十亿美元的产业，并一直在图形硬件以及 CPU 的发展过程中扮演着催化剂的角色。

言归正传，动作类游戏设计的挑战之一便是如何正确地处理碰撞检测问题，即判断两个对象间以及对象与场景间的相交问题。鉴于游戏最初源于 2D 场景，因而碰撞检测主要集中于如何在屏幕空间内高效地计算对象间的相交状态。时至今日，虽然硬件的计算速度已提高了不止 1000 倍，但碰撞检测仍被视为一类核心问题，这一点确实令人深思。目前，3D 游戏已经成为主流，但其复杂度也随之上升——动辄千万个多边形的计算量已屡见不鲜。相应地，在处理这一类较大的数据集时，与碰撞检测相关的数据结构和算法也变得日趋复杂，皆因这一类计算具有显著的实时特征。这里需要指出的是，游戏设计只是复杂实时碰撞检测应用的一个分支，在诸如 CAD/CAM 系统以及 3D 建模设计中也面临着同样的问题。

针对游戏设计以及其他实时应用程序，本书详细阐述了与碰撞检测问题相关的高效解决方案，包括相应的数据结构和算法。阅读本书需要读者具备良好的数学知识，相关内容也会在书中进行深入的探讨。同时，为了帮助读者进一步理解计算方案的实现过程，本书还提供了相应的算法、代码以及伪代码。

综上所述，碰撞检测问题涵盖了丰富的内容，甚至书中的各章均可独立成书。鉴于此，本书内容力争做到有的放矢，以便为读者今后的学习打下坚实的基础。

关于作者

本书作者 Christer Ericson 毕业于瑞典 Umeå 大学并获得计算机科学硕士学位。随后，在该校任教多年并于 1996 年移居美国。1999 年，他曾在索尼电脑娱乐公司美国分公司担任高级主程序员、高级软件配置管理工程师和首席软件工程师。此前，他还曾在 Neversoft 娱乐公司担任主程序员。2002 年以来，他一直在“Full Sail 游戏设计”和项目开发部担任顾问委员会委员。Christer Ericson 涉猎广泛，目前他将主要精力集中在程序优化管理方面，并在游戏开发者大会上就此问题进行了演讲。

致谢

下列读者对本书内容提出了相关的指正及修改建议，在此表示衷心的感谢，他们是：Ian Ashdown,Gino van den Bergen,David Eberly,George Innis,Neil Kirby,Eric Larsen,Thomas Larsson, Amit Patel, Jamie Siglar,Steven Woodcock 以及一位低调的匿名读者。

另外，我的感谢名单还包括：Ville Miettinen，感谢他慷慨地提供了源代码，这使得与包围体构造相关的章节看起来更加出色；感谢 Matt Pharr，他的建议令我受益匪浅并极大地丰富了第 13 章中的内容；当然，还要感谢 Tim Moss（因为他，我可以按时下班回家并致力于本书的创作）和 Bob Soper（感谢他对初稿提出的建议并就本书的构思与我进行愉快的交流）。

在此，还要感谢编辑 Tim Cox，他对本书的创作赋予了极大的耐心；感谢助理编辑 Richard Camp 和 Stacie Pierce，以及为本书出版付出艰辛劳动的 Morgan Kaufmann 出版社的全体工作人员。

最后，请允许我向 Kim,Ellinor,Tekla 和 Maja 献上我最真挚的谢意，感谢他们 4 年来对我的支持，并在本书的创作过程中陪伴我度过无数个不眠之夜。

作为一名读者，此刻，你值得拥有！

Christer Ericson

目 录

Contents

第 1 章 概述 1
1.1 内容概览 1
1.1.1 第 2 章：碰撞检测系统中的设计问题 2
1.1.2 第 3 章：数学和几何学入门 2
1.1.3 第 4 章：包围体 2
1.1.4 第 5 章：基本图元测试 2
1.1.5 第 6 章：层次包围体技术 2
1.1.6 第 7 章：空间划分 2
1.1.7 第 8 章：BSP 树层次结构 3
1.1.8 第 9 章：凸体算法 3
1.1.9 第 10 章：基于 GPU 的碰撞检测 3
1.1.10 第 11 章：数值健壮性 3
1.1.11 第 12 章：几何健壮性 3
1.1.12 第 13 章：优化操作 3
1.2 关于本书的代码 4

第 2 章 碰撞检测系统中的设计问题 5
2.1 碰撞算法的设计因素 5
2.2 应用程序中对象的表达方式 5
2.2.1 对象的表达方式 6
2.2.2 碰撞与几何渲染 7
2.2.3 特定的碰撞检测算法 8
2.3 查询类型 9
2.4 环境模拟参数 9
2.4.1 物体对象的数量 9
2.4.2 顺序移动和同步移动 10
2.4.3 不连续移动与连续移动 11
2.5 性能 12
2.5.1 优化概览 12
2.6 健壮性 13
2.7 实现与使用的简洁性 13
2.7.1 碰撞检测系统的调试 14
2.8 小结 14

第 3 章 数学和几何学入门 15
3.1 矩阵 15
3.1.1 矩阵运算 16
3.1.2 矩阵的几何代数符号 17
3.1.3 行列式 17
3.1.4 利用克莱姆法则计算线性方程组 19
3.1.5 2×2 矩阵和 3×3 矩阵的逆矩阵 20
3.1.6 行列式断言 21
3.2 坐标系统和顶点 23
3.3 向量 23
3.3.1 向量运算 24
3.3.2 向量的代数恒等式 25
3.3.3 点积 26
3.3.4 点积的代数恒等式 27
3.3.5 叉积 27
3.3.6 叉积的代数恒等式 29
3.3.7 标量三重积 29
3.3.8 标量三重积的代数恒等式 30
3.4 质心坐标 30
3.5 直线、光线和线段 35
3.6 平面和半空间 36
3.7 多边形 37
3.7.1 多边形凸性测试 39
3.8 多面体 41
3.8.1 凸体测试 43
3.9 凸包计算 43
3.9.1 Andrew 算法 43
3.9.2 Quickhull 算法 44

3.10 域 46
3.11 Minkowski 和与 Minkowski 差 47
3.12 小结 48

第 4 章 包围体 49
4.1 BV 期望特征 49
4.2 轴对齐包围盒 50
4.2.1 AABB 间的相交测试 51
4.2.2 AABB 的计算与更新 53
4.2.3 基于包围球的 AABB 54
4.2.4 基于原点的 AABB 重构 54
4.2.5 利用爬山法构造 AABB 55
4.2.6 旋转 AABB 后的重计算 56
4.3 Spheres 球体 58
4.3.1 其他相交测试 58
4.3.2 计算包围球 58
4.3.3 最大离散方向上的包围球 60
4.3.4 采用迭代修正的包围球 66
4.3.5 最小包围球 66
4.4 方向包围盒 68
4.4.1 相交测试 68
4.4.2 健壮的分离轴测试 72
4.4.3 计算紧凑的 OBB 72
4.4.4 基于 PCA 的 OBB 优化 74
4.4.5 蛮力法实现 OBB 的拟合 76
4.5 球扫掠体 76
4.5.1 球扫掠体的相交测试 77
4.5.2 球体扫掠体包围体的计算 78
4.6 半空间相交体 78
4.6.1 Kay-Kajiya 平行平面空间包围体 79
4.6.2 离散有向多面体（*k*-DOP） 79
4.6.3 *k*-DOP—*k*-DOP 相交测试 81
4.6.4 *k*-DOP 的计算与重对齐 81
4.6.5 近似凸体相交测试 83
4.7 其他类型的包围体 83
4.8 小结 84

第 5 章 基本图元测试 85
5.1 最近点计算 85
5.1.1 点到面的最近点 85
5.1.2 点至线段的最近点 86
5.1.3 点至 AABB 的最近点 88
5.1.4 点至 OBB 的最近点 90
5.1.5 点至三角形的最近点 93
5.1.6 点到四面体的最近点 97
5.1.7 点到凸多面体的最近点 99
5.1.8 两条直线间的最近点 100
5.1.9 两线段上的最近点 101
5.1.10 线段和三角形最近点 105
5.1.11 两个三角形之间的最近点计算 106
5.2 图元测试 107
5.2.1 分离轴测试 107
5.2.2 球体与平面间的测试 110
5.2.3 盒体与平面间的测试 111
5.2.4 锥体与平面间的测试 113
5.2.5 球体与 AABB 之间的测试 113
5.2.6 球体与 OBB 之间的测试 114
5.2.7 球体与三角形之间的测试 115
5.2.8 球体与多边形之间的测试 115
5.2.9 AABB 与三角形之间的测试 116
5.2.10 三角形之间的测试 119
5.3 直线、光线和有向线段的相交测试 120
5.3.1 线段与平面的相交测试 120
5.3.2 光线或线段与球体的相交测试 122
5.3.3 光线或线段与盒体的相交测试 123
5.3.4 直线与三角形之间的相交测试 127
5.3.5 直线与四边形之间的相交测试 130
5.3.6 光线或线段与三角形之间的相交测试 131
5.3.7 光线或线段与圆柱体之间的相交测试 135
5.3.8 光线或线段与凸多面体之间的相交测试 137
5.4 其他类型的测试 139
5.4.1 点与多边形之间的测试 139
5.4.2 点与三角形之间的测试 141
5.4.3 点与多面体之间的测试 143
5.4.4 两个平面间的相交测试 144
5.4.5 3 个平面间的相交测试 146
5.5 动态相交测试 148

5.5.1 运动物体的区间半分法相交测试 149
5.5.2 运动凸体对象的分离轴测试 152
5.5.3 运动球体与平面间的相交测试 152
5.5.4 运动 AABB 与平面间的相交测试 154
5.5.5 运动球体与球体之间的相交测试 155
5.5.6 运动球体与三角形以及多边形之间的相交测试 157
5.5.7 运动球体与 AABB 之间的相交测试 158
5.5.8 运动 AABB 之间的测试 160
5.6 小结 162

第 6 章 层次包围体技术 163
6.1 层次结构设计问题 164
6.1.1 BVH 的期望特征 164
6.1.2 性能函数 164
6.1.3 树的度数 165
6.2 层次结构的构建策略 165
6.2.1 自顶向下的构造方法 166
6.2.2 自底向上的构造方法 170
6.2.3 扩充（插入）构造策略 174
6.3 层次结构的遍历 176
6.3.1 下降规则 177
6.3.2 通用的启发式深度优先遍历 178
6.3.3 同步深度优先遍历 180
6.3.4 优化的有向叶节点深度优先遍历 181
6.4 包围体层次结构示例 182
6.4.1 OBB TreesOBB 树 182
6.4.2 AABB 树和盒体树 183
6.4.3 采用 8 叉树子划分的球体树 184
6.4.4 采用球体覆盖表面的球体树 184
6.4.5 生成-修剪球体覆盖 184
6.4.6 *k*-DOP 树 185
6.5 合并包围体 186
6.5.1 合并两个 AABB 186
6.5.2 合并两个球体 187
6.5.3 合并两个 OBB 188
6.5.4 合并两个 *k*-DOP 188
6.6 高效的树型表达方式及遍历 189
6.6.1 数组表达方式 189
6.6.2 前序遍历 190
6.6.3 采用偏移量而非指针 191
6.6.4 采用缓存友好的结构（非二叉树） 192
6.6.5 树节点和图元排序 192
6.6.6 递归遍历 193
6.6.7 分组查询 194
6.7 通过缓存机制改善查询 196
6.7.1 表面缓存：缓存相交图元 196
6.7.2 前界面追踪 198
6.8 小结 199

第 7 章 空间划分 200
7.1 均匀网格 200
7.1.1 网格单元的尺寸 200
7.1.2 采用链表数组表示的网格 201
7.1.3 哈希存储与无限网格 202
7.1.4 静态数据存储 204
7.1.5 隐式网格 204
7.1.6 使用均匀网格的对象间的测试 207
7.1.7 网格的其他注意事项 210
7.2 层次网格 211
7.2.1 基本的层次网格实现方式 212
7.2.2 其他类型的层次网格表达方式 216
7.2.3 其他层次网格 216
7.3 树 217
7.3.1 8 叉树（以及 4 叉树） 217
7.3.2 8 叉树对象的分配 218
7.3.3 位置码和 8 分体的定位 221
7.3.4 基于哈希存储的线性树 222
7.3.5 计算 Morton 键 223
7.3.6 松散 8 叉树 224
7.3.7 *k*-d 树 225
7.3.8 混合方案 227
7.4 光线和有向线段的遍历 228
7.4.1 *k*-d 树相交测试 228
7.4.2 均匀网格的相交测试 229
7.5 排序扫掠算法 233
7.5.1 排序链表实现方案 234
7.5.2 基于数组的排序 238

7.6 网格单元和伪入口 240
7.7 避免重复测试 242
7.7.1 位标志 242
7.7.2 时间戳 243
7.7.3 分时清除时间戳 244
7.8 小结 246

第 8 章 BSP 树层次结构 248
8.1 BSP 树 248
8.2 BSP 树的类型 249
8.2.1 采用节点存储的 BSP 树 249
8.2.2 采用叶节点存储的 BSP 树 251
8.2.3 实体叶节点 BSP 树 251
8.3 构造 BSP 树 252
8.3.1 分割面的选择 254
8.3.2 分割面的评估 256
8.3.3 基于分割面的多边形分类 258
8.3.4 多边形分割计算 260
8.3.5 更多讨论 264
8.3.6 BSP 树的性能调试 265
8.4 BSP 树的应用 266
8.4.1 点与实体叶节点 BSP 树间的测试 266
8.4.2 光线与实体叶节点 BSP 树间的相交测试 267
8.4.3 基于实体叶节点 BSP 树的多面体查询 269
8.5 小结 272

第 9 章 凸体算法 273
9.1 基于边界的碰撞检测 273
9.2 最近特征算法 274
9.2.1 V-Clip 算法 274
9.3 层次多面体表达形式 276
9.3.1 Dobkin-Kirkpatrick 层次结构 277
9.4 线性规划和二次规划 278
9.4.1 线性规划 279
9.4.2 二次规划 283
9.5 Gilbert-Johnson-Keerthi 算法 284
9.5.1 算法概述 284
9.5.2 计算单形体内的最小范数顶点 286
9.5.3 GJK 算法、最近点以及接触流形 287
9.5.4 利用爬山法计算极值顶点 288
9.5.5 与顶点缓存相关的一致性问题 289
9.5.6 旋转对象的优化 290
9.5.7 移动对象的 GJK 算法 290
9.6 Chung-Wang 分离向量算法 291
9.7 小结 292

第 10 章 基于 GPU 的碰撞检测 294
10.1 GPU 接口 294
10.1.1 缓冲区读取 295
10.1.2 遮挡查询 295
10.2 凸体对象间的测试 296
10.3 测试凹体对象 298
10.4 基于 GPU 的碰撞过滤 301
10.5 小结 303

第 11 章 数值健壮性 304
11.1 健壮性问题的分类 304
11.2 实数表示法 305
11.2.1 IEEE-754 浮点格式 307
11.2.2 无穷运算 309
11.2.3 浮点误差源 311
11.3 健壮的浮点数用法 313
11.3.1 浮点值的误差容值比较 313
11.3.2 采用厚平面实现算法的健壮性 315
11.3.3 采用共享计算实现算法的健壮性 316
11.3.4 厚对象的健壮性 318
11.4 区间计算 319
11.4.1 区间计算实例 319
11.4.2 碰撞检测中的区间计算 320
11.5 精确计算和近似计算 321
11.5.1 采用整型数据实现精确计算 321
11.5.2 整型除法 324
11.5.3 采用整型运算处理线段相交问题 325
11.6 提高数值健壮性的进一步讨论 327
11.7 小结 328

第 12 章 几何健壮性 329
12.1 顶点焊接 330

12.2 计算邻接信息 335
12.2.1 计算顶点-面表 338
12.2.2 计算边-面表 339
12.2.3 连通性测试 341
12.3 孔、缝隙、间隙以及 t-连接 343
12.4 共面数据面的合并操作 345
12.4.1 测试多边形的共面性 346
12.4.2 多边形的共面测试 347
12.5 三角形剖分和凸划分 350
12.5.1 耳式剪裁实现三角剖分 350
12.5.2 多边形的凸剖分 353
12.5.3 多面体的凸剖分 355
12.5.4 不可剖分的凹几何体 357
12.6 采用欧拉公式的一致性测试 358
12.7 小结 360

第 13 章 优化操作 361
13.1 CPU 缓存 362
13.2 指令缓存优化 363
13.3 数据缓存优化 365
13.3.1 结构优化 366
13.3.2 顶点数据的量化操作和压缩操作 369
13.3.3 预取和预载操作 369
13.4 基于缓存感知的数据结构和算法 371
13.4.1 紧凑型静态 *k*-d 树 371
13.4.2 紧凑型 AABB 树 374
13.4.3 缓存参数无关性 375
13.5 软件缓存 376
13.5.1 缓存线性化操作实例 376
13.5.2 基于分摊机制的预测线性化缓存 379
13.6 数据别名 380
13.6.1 基于类型的别名分析 381
13.6.2 restrict 指针 382
13.6.3 避免别名问题 384
13.7 采用 SIMD 优化的并行操作 385
13.7.1 4 球体-4 球体 SIMD 测试 386
13.7.2 4 球体-4AABB SIMD 测试 386
13.7.3 4AABB-4AABB SIMD 测试 387
13.8 分支结构 388
13.9 小结 390

参考文献 392

第1章 概 述

本书将探讨碰撞检测这一主题，即处理如下看似简单的问题：检测两个（或多个）物体是否相交。特别地，碰撞检测将确定两个物体是否、何时以及在何处形成碰撞。“是否碰撞”将产生一个布尔值，并计算两个物体是否相交。“何时碰撞”将额外测定碰撞发生的时刻。“何处碰撞”将解释物体如何形成碰撞。一般来讲，以上述给定顺序分析这3类问题是一个比较复杂的过程。

“何时碰撞”和“何处碰撞”（以及布尔碰撞结果值）常被解释为碰撞检测。术语“相交检测”和“冲突检测”有时也与“碰撞检测”具有相同的含义。

碰撞检测计算往往是各种相关应用程序的基本原理，包括计算机游戏、物理仿真（如计算机动画）、机器人技术、虚拟样机技术以及工程仿真。

在计算机游戏中，碰撞检测将保证真实世界的正确虚拟化（例如，禁止人物穿越墙壁，或者防止人物坠落至地板之下）。同时，碰撞检测还提供了某种视线查询，通知敌人是否发现玩家并发动相应的攻击。

在计算机动画（例如布料的实现）中，碰撞检测常用于约束布料的物理模拟，确保布料以更加真实的方式运动。同时，还可以防止人物运动时产生“脱离”现象。在机器人模拟程序中，碰撞检测还可以用于路径规划，从而使机器人能够避开障碍物。在虚拟样机技术中，物理模型进行批量生产之前，碰撞检测常用于修正计算。另外，碰撞检测还可用于撞击测试和其他类型的工程仿真。

某些应用程序，例如路径规划和动画渲染，并不需要其碰撞检测系统的实时特性；而有些应用程序，特别是计算机游戏，则十分强调碰撞检测系统的实时性能。一些基于计算机或游戏机平台的动作类游戏所涉及的模拟计算数据量，则要求以每秒30帧～60帧的帧速率进行计算。在这种实时状态以及游戏和物理引擎中碰撞检测占优的状态下，碰撞检测系统将占用游戏一帧中的大量时间消耗。因此，在计算机游戏中，拙劣的碰撞系统设计将导致其瓶颈的产生。

本书内容并非只是泛泛阐述碰撞检测这一概念，同时，还将强调实时应用程序中与碰撞检测相关的数据结构和算法的高效实现方法。碰撞检测常被用于游戏领域并加以演示，然而，其他一些非游戏类应用程序也存在类似（或更加苛刻）的性能需求，包括：触觉系统（力反馈）、粒子系统的模拟、医疗外科模拟系统，以及其他一些仿真系统。本书所阐述的相关方案同样适用于这一类应用程序。

这里所讨论的解决方案不仅仅适用于碰撞检测系统。例如，在第6章和第8章中所讨论的内容，同样适用于光线跟踪计算和光线投射计算（如计算场景光照）以及地理信息系统。一些计算机图形学领域中的问题，也可以利用碰撞检测加以解决。例如，视见体的剔除问题就可以采用第6章和第7章所描述的方法实现。

1.1 内容概览

本节将描述本书中各个章节的内容提要。

1.1.1 第 2 章：碰撞检测系统中的设计问题

本章阐述构造碰撞检测系统时应考虑的各种问题，以及影响系统设计的各种因素。包括：物体如何显示、物体的数量、物体的移动方式，以及用户所触发的碰撞查询类型。本章还将介绍贯穿本书的一些相关术语。

1.1.2 第 3 章：数学和几何学入门

任何一种复杂的碰撞检测系统都需要大量的、基于几何定位的数学知识，用于解决碰撞计算的 if、when 和 where 等问题。本章介绍了后续章节中将要用到的数学和几何学概念。

1.1.3 第 4 章：包围体

为了加速碰撞查询的过程，一些简单的几何体对象（如球体和包围盒）常用于表达具有复杂特征的物体。只有当这些“简单的”包围体（该包围体足够大，用于包围具有复杂几何特征的物体）发生了碰撞，复杂几何体的测试计算才得以执行。本章将介绍一些常用的包围体类型、作用于包围体之上的相交测试，以及如何确定一个具有复杂几何特征的物体的包围体。

1.1.4 第 5 章：基本图元测试

在前述章节中介绍相交测试之后，本章将详细阐述不同类型物体之间的相交状态及其空间距离的计算，包括：直线、光线、线段、平面、三角形、多边形、球体、包围盒、圆柱体和多面体。同时，本章还将分别针对静态和动态物体加以测试。

1.1.5 第 6 章：层次包围体技术

对于包含众多物体对象的对象集而言，可以通过构建相应的包围体层次结构来获取高效的查询性能。在碰撞检测过程中，这种层次结构提供了碰撞对象，抑或单一对象中未碰撞部分的快速确认方案，并允许查询限定在少量对象或者对象的某一部分中。本章将讨论包围体的层次结构特征、构造方案，以及作用于其上的查询方法。同时，本章还将探讨实现这一层次结构的高效方案。

1.1.6 第 7 章：空间划分

当碰撞测试计算涉及大量的物体对象时，多个物体需要划分为子集并用于快速测试（避免测试时出现 n^2 级别的运算）。在第 6 章中讨论的包围体层次结构给出了实现该划分的方法。

1.1.7　第 8 章：BSP 树层次结构

一种比较通用的、用于表达碰撞检测系统数据结构的方法是二分空间划分树，即 BSP 树。BSP 树可以从空间对象中单独划分出子空间，也可以用于从对象所处空间中划分其边界线，因此能够高效地实现相应物体对象的体空间表示方法。本章将讨论如何构造健壮的 BSP 树系统，以及如何在该系统中执行相关的测试。

1.1.8　第 9 章：凸体算法

本章将考查凸体对象碰撞测试时的其他高级解决方案，并探讨凸体的特征、层次表达方式、V-Clip 最近特征算法、线性和二次方程程序设计的数学优化方法、高效的 GJK 算法，以及 Chung-Wang 分离向量算法。

1.1.9　第 10 章：基于 GPU 的碰撞检测

与个人计算机的核心之一 CPU 相比，当前民用级别的图形显示卡已经整合了更强大的计算能力，这种改进激发了图形显示卡的外围计算能力。本章将考查如何利用图形硬件来执行碰撞检测测试。

1.1.10　第 11 章：数值健壮性

碰撞检测系统中很小的误差都将导致灾难的发生，例如，物体未曾与几何场景发生正确的碰撞，从而脱离场景。本章将讨论浮点算术的健壮性问题，以及解决这些问题的相关建议。

1.1.11　第 12 章：几何健壮性

第 11 章考查了计算系统的健壮性问题，本章则分析如何将任意的多边形集合转换为具有良好格式的几何元素，并将其正确地提供给碰撞检测系统，包括：顶点的焊接、缝隙的去除、共面合并，以及将物体分解为凸面（三角形）。

1.1.12　第 13 章：优化操作

针对特殊的硬件平台以及相关的代码调试，本章将考查如何利用前述高效的数据结构和算法，并在此基础上获取更优的性能。可以通过优化代码并利用存储结构（缓存），以及代码和数据的并行计算方式，从而获取巨大的性能提升。

1.2 关于本书的代码

本书并非是一本纯理论书籍，其相关的设计思想均辅以对应的代码示例。不同于其他书籍所采用的伪代码算法描述，本书中的绝大多数代码均采用 C++语言加以实现。首先，代码提供了理解（以及实现）算法问题所必需的细节；其次，读者可以运行相关代码并查看运行过程中变量值的状态以加深对该问题的理解，这对于不熟悉算法数学思想的读者尤为重要。本书仅有少量代码采用了伪代码实现，皆因其整体实现方案缺乏相应的可行性。

虽然本书的代码示例采用了 C++语言，但需要强调的是，本书的重点并非是 C++语言。这里，C++语言仅提供了相关概念的一种具体的可行性描述，其他计算机语言均可完成这一任务。本书选择 C++作为描述语言包括下列因素：该语言应用范围较广并可对几何实体的底层计算提供简洁的抽象操作。例如，可使用类和（重载后的）中缀操作符对顶点和向量实施有效的计算。考虑到代码的受众范围（如仅通晓 C 语言或 Java 语言的程序员），本书将会弱化 C++语言中的某些特性操作，例如模板和 STL（标准模板库）。因而，C++纯粹主义者请保持淡定。

同样，本书也并非是一本介绍软件工程的专业性书籍。为了更好地引入相关基础知识点，书中代码尽可能地保持短小精悍。另外，为了避免读者陷入 C++语法的繁文缛节中，书中代码也有相应的调整。例如，尽量保持简约风格的类定义（或不使用类）；将局部变量替换为全局变量；避免将指针声明为 const 类型（或 restrict 类型）；采取固定尺寸的数组而非动态数组。最后，代码还将对变量名的长度加以限制，以使其符合排版要求。

需注意的是，本书中的示例代码并非最终版本。例如，代码中一般不包含“除 0 错”测试，以使读者不至于过早地陷入细节问题中，从而干扰对整体算法的理解。同样，某些代码测试需要添加相应的容值以满足算法的健壮性要求。同时，第 11 章还将对健壮性问题进行详细讨论，并就实现健壮的代码这一问题彰显其中的不同之处。

另外，输入变量通常采取值传递，输出变量则一般通过引用返回，从而进一步区分二者间的差别。在某些情况下，按引用传递出入变量则更加高效，这一部分内容将作为练习留给读者完成。

在本书中，注释将采用手写体表示，代码则使用黑体标注。另外，函数名、类名、结构名以及用户自定义复合类型均以大写字母开头，变量则以小写字母开头。同时，变量的命名尽量与文中所提及的内容一致。但在某些情况下，上述规则也并非一成不变。例如，文中所描述的顶点一般采用大写字母，代码中则使用小写字母。

本书中相关示例代码已上传至 www.tup.com.cn，请读者自行下载。

第 2 章　碰撞检测系统中的设计问题

高效的碰撞检测系统设计过程有一些类似于拼图游戏：在见识一幅完整图画的庐山真面目之前，需要将每一个零散部件拼接起来。类似地，本书的主要工作即是将碰撞检测系统中的不同解决方案所形成的各个组件加以整合。通过本书的学习，读者将逐步地了解碰撞检测系统的全貌。本章将给出在选择诸多解决方案并考虑其相互关系时所涉及的相关问题。同时，本章还将介绍一些相关的术语，其定义和详细的解释将在后续章节中加以说明，其更深层次的内容解析亦将在后续章节中进行阐述。

2.1　碰撞算法的设计因素

在设计碰撞检测系统时，多种因素将会影响设计者的选择，包括以下几类：

（1）应用程序中对象的表达方式。场景以及物体的几何表达方式将直接影响到相关算法的选择，表达方式的约束条件越少，则碰撞检测的解决方案也就越通用，同时，也能够满足既定的运行性能。

（2）查询类型的多样化。通常，查询及其相应的结果越详细，需要的计算量就越大。另外，也许还需要额外的数据结构对某一类特定的查询加以支持。最后，某些对象的几何表达方式还可能不存在相应的查询类型与之对应。

（3）环境模拟参数。模拟环境自身亦包含了某些参数，并对碰撞检测系统产生直接的影响。例如，对象的数量、尺寸和位置、是否移动以及如何移动、物体间是否允许相互穿透、物体是刚体还是可变形物体。

（4）系统的性能。实时碰撞检测系统对时间要求苛刻，并受到物体尺寸的约束。由于时间和空间的可交换性，为了满足既定的性能需求，某些特性通常应均衡加以考虑。

（5）系统的健壮性。应用程序的物理仿真级别是存在差异的。例如，考查一个堆砌的砖垛，与一个球场上跳动的篮球相比，其碰撞检测系统则要复杂得多。篮球弹射初期或以较大角度弹射时，其状态可以忽略不计。但是，在计算砖块之间的接触点时，即使是很小的误差，都将导致砖块逐渐地彼此嵌入或脱离。

（6）系统实现与应用的简易性。多数项目设计都存在其自身的开发周期，若系统不能按时完成并交付使用，那么项目的时间规划将是一句空谈。因此，在考查采用何种方案时，“系统实现的简洁性”这一决策将扮演重要的角色。

上述内容所涉及的问题将在后续章节中详细讨论。

2.2　应用程序中对象的表达方式

在选择合适的碰撞检测算法时，要考虑到场景和物体的几何表达方式，这一点是十分重要的。本

小节将简要地讨论各种物体对象的几何表达方式，并探讨如何使用简化后的几何体，以及如何使特定的方法应用于通用的解决方案之中。

2.2.1 对象的表达方式

当前多数计算机图形硬件均使用三角形作为基本的渲染图元。因此，多边形这一表达方式对于场景和场景中的物体，及其相应的碰撞几何体，将是一个自然的选择。而最为通用的对象表达方式则是“多边形汤”：一组无序的多边形构成的集合，并且不包含确定多边形相互关系的连接信息。由于没有过多的表达限制，对于美术师和关卡设计者来讲，多边形汤往往是一种较具吸引力的对象表达方式。多边形汤的相关算法虽然可以适用于任意多边形集合，但是与那些具备额外信息的多边形集合相比较而言，其执行效率低下并且算法的健壮性也较差。例如，多边形汤并没有考虑到物体对象的“内部”信息，因此，这将较难确定两个对象是否错误地重叠在一起。而所谓额外的数据往往包含一条边上的两个顶点的信息、两个面的连接信息、物体对象是否为一个封闭实体，以及物体对象是否为凸体抑或凹体。

多边形通过公共边可以连接成一个大型的多边形平面，称作多边形网格。如图 2.1 所示，当构造集合模型时，采用多边形网格集合是最常用的方法。

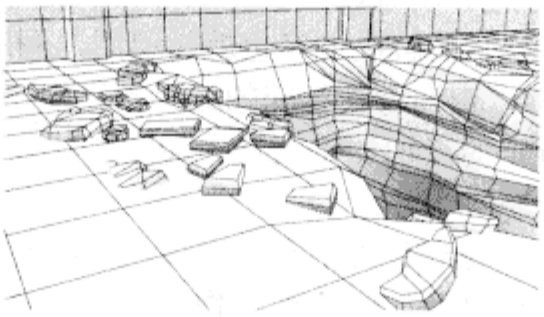


图 2.1　采用多边形网格集合构建的集合模型

多边形物体按照顶点、边和面加以定义。依据该方法构造的物体对象常被称为“具备显式的表达方式”。而“隐式表达方式”的物体对象一般指球体、圆锥体、椭圆、圆环以及其他非显式定义的、但可以间接地通过数学表达式表示的几何图元。隐式物体对象常描述为 3D 空间至实数的函数映射，$f:\mathbb{R}^3 \rightarrow \mathbb{R}$。其中，$f(x,y,z)<0$ 代表物体的内部点集；$f(x,y,z)=0$ 代表物体边界上的点集；$f(x,y,z)>0$ 代表物体的外部点集，如图 2.2 所示。通过隐式函数定义的物体对象边界常称作隐式表面。隐式物体对象常用于对场景中的物体进行粗略的快速剔除；在光线跟踪计算中，也可用于对直线和光线执行快速相交测试。隐式测试的一些具体实例将在第 5 章中加以分析。

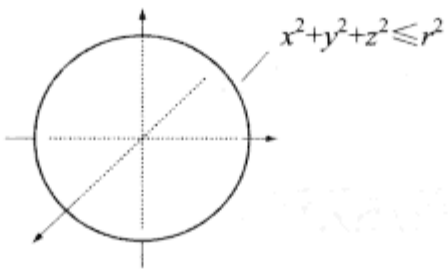


图 2.2　一个显式定义的球体（包括球体边界和内部区域）

凸体多边形对象也称作半空间的交集。例如，立方体可描述为 6 个半空间的交集，其中，每一个半空间均去除了立方体面的部分外部空间。半空间和半空间体的相关内容将在第 3 章加以详细解释。

球体、包围盒和圆柱体也可称为结构几何实体（CSG）对象的构造块。CSG 物体对象可以在基础的几何形状或其他 CSG 物体上，利用集合理论（如并集、交集、差集）实现递归构造，这将构造出任意形状的复杂几何对象。因此，CSG 对象可以表达为一棵（二叉）树，内部节点表达为集合操作；叶节点为几何图元，如图 2.3 所示。CSG 物体属于隐式对象，其中，顶点、边和面均无法直接获取。

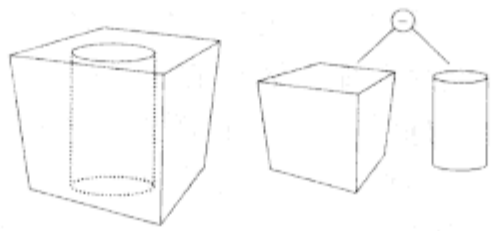


图 2.3　（a）一个包含柱状孔的立方体。（b）相应的 CSG 树型结构（圆柱体从立方体中被析取）

CSG 模型的强度是指：物体对象通常是有效的，即不存在任何影响多边形表达的某些问题（如缝隙）。CSG 对象也是一种空间体的表达方式，它使得判断一点是否位于 CSG 物体之内变得十分容易。例如，CSG 多面体对象可以通过[Laidlaw86]和[Thibault87]中所描述的步骤加以实现。然而，由于计算过程中的数字误差，其健壮性难以实现。

2.2.2　碰撞与几何渲染

虽然几何渲染可以直接置入碰撞检测系统，但最好将渲染系统与其分离开来，理由如下：

（1）当前，图形平台之上的几何体渲染过程十分复杂，并且不适合用于碰撞检测和物理运算。另外，对于碰撞的精确度也存在着某种限制。因此，不建议采用供渲染的几何对象执行碰撞检测，而是采用同一位置的替代对象。例如，在游戏中，常利用球体、包围盒来表达游戏物体，从而可忽略物体的复杂度。如果该替代对象发生了碰撞，则真实物体才暂且假定为也发生了碰撞。上述这些简单的几何形状或包围体忽略了物体对象的真实表示，它们常常用于加速碰撞检测。一般来讲，包围体通常全方位包围原几何体。关于包围体的详细讨论可参见第 4 章。

（2）对于当前的硬件环境来讲，几何图元趋于一种特定的格式并加以表达。例如，三角带和索引顶点缓存，从而提高了自身的渲染速度。但这对碰撞检测系统并不适用。因此，无须分解上述几何结构（即使它们是为了实现复用性并已加以缓存），提供某种特定的碰撞几何体通常会使碰撞检测更高效。对于碰撞几何体，某些非三角形图元的使用有时会带来性能上的提升。

（3）在数据组织方面，渲染几何体与碰撞检测几何体二者之间存在着显著的区别：静态渲染数据可以按照材质的特征加以分类排序，而碰撞数据一般按照空间特征加以组织。渲染几何体需要内嵌诸如材质信息、顶点颜色和纹理坐标等数据信息，而碰撞几何体则关联其面属性。分离两者的数据格式并只保持碰撞相关信息，将有利于碰撞数据的最小化。反之，这种较好的数据缓存一致性也将有助于执行效率的改善。

（4）有时，通过一些设计手法，也可以对碰撞几何体和渲染几何体加以区别。例如，在一款滑雪游戏中，如果雪层的深度到达膝盖，则建模过程中，可以将碰撞表面设定为雪层渲染表面以下 2 英尺

处。在刚没过脚踝的、摇曳的草丛中前行，或者在齐腰深的浑浊水域中跋涉，也可以采用相同的手法加以处理。甚至，当渲染几何体也用于碰撞体时，仍需要将渲染几何体从碰撞几何体数据集合中分离出来。

（5）为了实现正确的模拟，即使渲染数据透明不可见，其碰撞数据也需要加以保留。只有当碰撞区域小于对应的渲染几何体时，其碰撞标记才可以适度减少。

（6）原始的几何体可以通过多边形汤或网格形式给定，而模拟过程通常需要以实体方式加以显示。在这种情况下，计算实体的替代几何体将会更加简便，而非实体化原始的几何显示数据。

然而，采用分离的碰撞几何体仍然存在如下潜在的缺点：

（1）数据冗余（主要是顶点数据的重复）将导致额外的内存占用。这一问题，可以通过从实时渲染的几何体中创建部分或全部碰撞几何体这一方式加以缓解，其具体内容将在 13.5 节中加以描述。

（2）生成并维护这两类相似几何体集合还需要一些额外的步骤。手工构造替代几何体将会减缓设计者的开发进度。若该替代几何体是通过某种工具加以构造的，那么在碰撞系统的构造过程中，该工具应该是数据可写入的；另外，如果用户手工修改该工具的数据输出值，那么该变化状态也应该能够通知该工具和原始数据集合。

（3）如果构造过程和维护过程分开加以实现，那么渲染几何体和碰撞几何体之间可以不实现完全匹配。因此，当碰撞几何体不完全填充其对应的渲染几何体空间时，渲染物体之间可能会出现遮挡的现象。

（4）对象状态和其他逻辑问题也可能产生于上述两种几何体之中。当渲染几何体发生变化时，碰撞几何体一定要随之重新构造吗？碰撞几何体与渲染几何体之间，哪一个首先创建？当一种几何体发生变化时，另一种又应该如何随之更新呢？

采用与真实尺寸接近的替代几何体在游戏中亦运行良好。从直观角度上来讲，用户并不擅长判断真实的碰撞是否刚好发生，模拟过程中，所涉及的物体越多，其移动速度越快，玩家也就越难以把握；同时，用户也难以预测碰撞的最终结果，从而采取相应的反馈措施。在游戏中，碰撞检测和相应的反馈总是遵循“如果看上去是正确的，它就是正确的”这一法则。而其他一些应用程序，则需要比较严格的精确度。

2.2.3 特定的碰撞检测算法

不建议设计并采用那种“全功能”的碰撞检测系统，为特定方案提供某种专用的碰撞检测系统往往是更加明智的选择。一个与特定算法相关的例子是粒子系统。通常，粒子往往以粒子云的方式发射，从而使得碰撞系统能够较好地接收并处理它们，而不是采用逐个粒子发射的方式。其中，粒子云可以依据当前环境加以构造并实现重组。而在某些碰撞可以忽略的情况下，粒子甚至可以不参与碰撞检测。

另一个例子是：为物体-物体间的碰撞与物体-场景间的碰撞分别使用不同的算法。物体-物体之间的碰撞甚至可以采用更加特殊的算法，从而使玩家和快速运动的发射物区别于其他对象的处理。例如，将全部物体与对象发生碰撞作为硬编码加以测试，通常可以获得较好的控制效果，而不是将玩家与某一通用的碰撞系统进行单独测试。

下面具体考查大型场景的模拟过程。首先，对于小型的场景，碰撞数据可以一直存储于内存之中；而对于大型的、无缝连接的场景世界，其加载、卸载过程是随着场景的遍历而执行的。对于后者，令

对象与场景结构分离是一个不错的选择，因此，对象不会随着场景结构的变化而变化。但是，令物体与场景的结构加以分离，将会导致一个潜在的缺陷：查询过程将会遍历这两类数据结构，而非一次遍历。

2.3　查询类型

最直接的碰撞查询称为“冲突检测或相交测试”，即回答两个（静态）物体在其给定的位置和方向上是否产生重叠这一布尔问题。布尔相交测试查询快速并易于实现，因此被广泛地采用。然而，有时仅知道碰撞的布尔结果值是远远不够的，还需要获取相交处的具体数据信息。相交数据的计算过程一般是比较困难的一个步骤，涉及计算一个或多个碰撞点。

对于某些应用程序，计算物体之间的一个碰撞点往往已经足够；然而，在某些程序中，例如刚体的模拟，则需要计算碰撞点的集合。健壮的碰撞点集的计算通常是比较困难的。综上所述，与精确计算查询相比，近似计算查询往往更加易于处理。当然，后者所需的精确度取决于相应的误差范围。游戏中一般采用近似计算查询。另外，在游戏中，碰撞计算通常需要给出某些特定的属性值，该值一般为某物体或其表面的、已定义好的属性值。例如，道路表面的防滑系数，或者墙壁表面的坡度。

如果物体之间彼此穿越，某些程序还需要计算该穿越深度。这里所说的“穿越深度”通常定义为最小位移距离：分离两物体的最短位移向量的长度值。但是计算该位移向量往往比较困难。两个不相交物体之间的“间隔距离”则定义为：物体 A 上的点与物体 B 上的点之间的最小距离值。当该距离值为 0 时，则两物体相交。两物体间的距离计算对于预测下一次碰撞的发生时刻是非常有用的。一个比较常见的问题是计算两物体间的最近碰撞点：即两物体的间隔距离。注意，最近碰撞点并不一定是唯一的，也可能存在无限多个。对于动态物体，计算下一次碰撞的发生时刻常称作“预计到达时刻（ETA）计算”，或称作“碰撞时刻（TOI）计算”。在刚体模拟中，ETA 值往往用于控制时间步（time step）。另外，运动的类型也是模拟参数之一，其进一步的讨论参见 2.4 节。

2.4　环境模拟参数

如本章前面内容所述，模拟过程中的某些参数将直接影响碰撞检测系统的相应算法的选择。为了演示环境模拟参数可能产生的问题，下面几节内容将集中考查物体对象的数量问题，以及碰撞处理过程中物体对象的运动方式等。

2.4.1　物体对象的数量

由于任何一个物体都有可能与其他物体发生碰撞，则包含 n 个物体的碰撞检测过程，在最坏情况下需要$(n-1)+(n-2)+\cdots+1=n(n-1)/2=O(n^2)$ 次相互测试。对于这种平方级别的时间复杂度，物体间碰撞测试的代价将会变得十分昂贵；同时，减少这种相互测试所产生的消耗，将会线性地影响测试的运行时间。为了提高测试速度，上述物体间相互测试的数量必须适度地降低。这种降低的方式可以描述为：将包含大量物体对象的碰撞操作划分为两个阶段——粗略测试阶段（broad phase）和精确

测试阶段（narrow phase）。

粗略测试将确定可能发生碰撞的较小的一组物体对象，并快速排除那些一定不会发生碰撞的物体对象。精确测试则继续在上述子划分组中执行物体对象间的相互碰撞测试。粗略测试和精确测试也分别称作“n 体操作”和“对组合操作”。

通过分治策略，粗略测试操作减少了相应的计算工作量，如图 2.4 所示。对于全部 11 个物体对象（如图 2.4 中矩形所示），其全部测试过程需要 55 次单独的对组合测试。经过粗略测试操作后，产生了 5 个独立的子划分（图 2.4 中着色区域）。那么，在精确测试中，只需执行 10 次独立的对组合测试。粗略测试操作将在第 6 章～第 8 章加以讨论；精确测试操作将在第 4 章、第 5 章、第 9 章加以讨论。

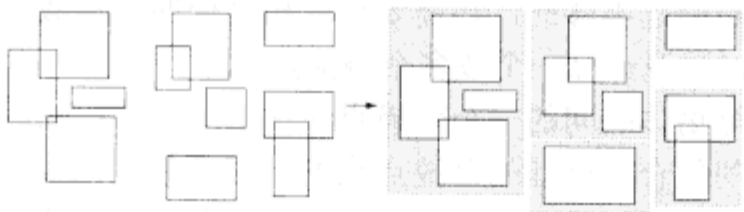


图 2.4　粗略测试阶段用于确定相交对象间的各个分离子集

除了物体对象的数量问题，其相应尺寸亦将影响测试的执行次数。与确定一组均匀尺寸的物体对象集相比，粗略测试系统将对子划分过程实施较多的（或较复杂的）计算。物体对象尺寸如何对粗略测试产生影响，将在第 7 章中加以讨论。

2.4.2　顺序移动和同步移动

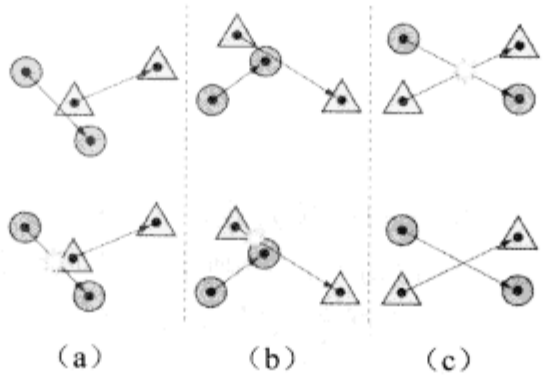
在现实生活中，当多个物体在给定的运动时间间隔内同步移动时，需要处理期间可能发生的碰撞。对于真实事件的准确模拟，要求确定某两个运动物体的先期碰撞时刻，当模拟系统准时到达这一时刻点时，将移动所有物体至首次发生碰撞的位置。从而，碰撞问题得以解决，同时，相关的操作过程将会继续计算下一次碰撞，重复上述步骤，直至全部的位移时间步用尽。

这种“重复地推进至下一个最早碰撞时刻”的模拟执行方式将会十分耗时。例如，当一个或多个物体逐渐停止向某一表面运动时，模拟过程仍然会即刻计算当前碰撞时刻后的下一次碰撞时刻。虽然模拟过程只是因此向后推移了一个很小的时间片，但是实际上却“永久地”独占了全部时间步。该问题的一种解决方案是，在时间步内，利用粗略测试确定可能与当前划分组碰撞的其他对象划分组而非组对象。而每一个子划分将以不同的代价加以执行，通常这将在一定程度上解决上述问题。

一种替代的方法是，仍然同步移动物体，但设定一个固定（较小）的位移时间步。同步移动物体可能导致物体之间的相互贯穿，一般可以通过将模拟备份至早期状态加以解决。在上述两种情况下，同步更新仍然是十分耗时的，因此只用于精确的刚体模拟中。然而，多数游戏和其他类型的应用程序并非刚体模拟，因此，试图模拟高精确度的运动方式将得不偿失。对此，一种可替代的方案是：以顺序的方式处理运动问题，即一次只移动一个物体，若检测到碰撞，则进行处理，随后再继续处理下一个物体。

显然，这种顺序运动并非物理意义上的精确运动模型。当对象间进行同步移动时（如图 2.5（a）所示），对于尚未移至当前帧中的对象而言，则有可能产生碰撞；而对于已移出碰撞路径之外的对象，则不会产生碰撞。另外，对于顺序移动的对象，也将会出现碰撞现象（如图 2.5（b）所示））。在某

些情况下，同步移动的对象间可能产生碰撞，而顺序移动的对象反而不会出现碰撞（如图 2.5（c）所示）。针对游戏程序设计而言，由于较高的帧速率仅呈现很小的位移量，进而使得相交区域难以实现量化操作，因而通常可忽略顺序移动模型所产生的问题。



（a）上图：两个对象实现同步移动且不存在碰撞；下图：若圆形对象在三角形对象之前移动，则对象间将产生碰撞。（b）同步移动将不产生碰撞，顺序移动则产生碰撞。（c）同步移动将产生碰撞，顺序移动则不产生碰撞

图 2.5　对象问题碰撞模型

顺序移动模型的优点之一便是避免了单一物体的穿透现象。例如，如果一个物体在运动期间发生了碰撞，则可以简单地撤销该运动位移。仅当需要撤销单一物体的运动位移时，才与采用固定时间步的同步移动模型具有可比性——后者将撤销所有同步运动物体的位移。

2.4.3　不连续移动与连续移动

某些因素将在很大程度上影响碰撞检测结果的计算量，以及碰撞结果自身的准确度。例如，“对组合”测试中的两个物体在执行测试时是否处于运动状态。静态碰撞测试负责处理：物体在运动过程中以及每一个离散点处是否发生碰撞。在每一个离散点处，物体在其当前位置视作静止并且速度为 0。相比较而言，动态碰撞测试则视为：在给定时间间隔内，物体的连续位移。动态碰撞测试通常能够告知首次碰撞时的精确碰撞时刻和碰撞点。与动态碰撞测试相比较，静态碰撞测试无须付出太多的代价即可实现，但是，测试之间的时间步必须足够小，从而使得物体的位移小于其占据的空间范围。否则，物体只是在相邻的两个时间步内彼此简单地经过对方，并且无法检测到相应的碰撞。这种现象称作“隧道（tunneling）效应”。

在给定时间间隔内，物体连续移动所覆盖的空间体积称作扫掠体（swept volume）。如果两个运动物体对应的扫掠体并没有相交，则两个物体也不存在相交。另外，即使两个扫掠体出现相交的情况，对应的两个物体也未必在其运动过程中相交。因此，扫掠体的相交是物体相交的必要但非充分条件。对于复杂的运动过程，扫掠体难以计算与使用。幸好，在计算过程中，较高的精确度并不是必需的。对于那种复杂的、呈现翻转运动的动态碰撞测试，其往往可以设定为分段线性运动，从而得到相应的简化：即运动范围内的直线平移和终点（或起始）处的旋转，而二者之间的某一位置则可看作是自由的螺旋运动（即固定旋转和平移运动的合成）。

当处理运动的物体时，考虑它们的相对运动通常是一种更可取的方式，即从一个物体的运动中“减去”另一个物体的运动，从而使得前者静止。因此，物体间的直线平移使得上述操作构成一个向量减法。

考查相对运动方式的一个优点是：当前行为变为一个运动的物体朝向一个静止的物体执行碰撞测试，从而使得对应的扫掠体测试成为一种准确的相交测试。在游戏中，扫掠体常被“速度盒（speed box）”所替代——一个延长的、包含物体全部运动范围的盒体（或其他类似形式的简单替代物，不仅仅是盒体）。

2.5 性能

这里举一个游戏控制的例子。一些优秀的视频游戏，以每秒 60 帧运行（限 NTSC 电视制式的国家；PAL 制式则为每秒 50 帧）。该帧速率使得每一游戏帧运行时间约为 16.7ms。碰撞检测约占用一帧的 10%～30%，即 2ms～5ms，当然，这还要依据游戏的具体类型。对于一款动作类游戏，某一时刻可能会存在多个需要执行碰撞检测的活跃物体，对于每一个物体，可能只有 50μs～250μs 的时间来处理碰撞检测问题。显然，减少平均碰撞计算查询时间将变得十分重要。当游戏中的帧速率显著下降时，确保选择执行的算法在最坏情况下耗时不要超出平均值太多，这一点也是很重要的。

有多种方法可以加速碰撞检测处理过程，本书将讨论其中的大部分内容。一些与优化相关的常规方法，将在下列内容中讨论。

2.5.1 优化概览

优化的首要原则是：无须执行额外的任务。因此，较好的加速优化方案是将工作量削减至最低。同样，碰撞检测系统最重要的优化方案之一便是 2.4.1 节中提及的粗略测试操作，即物体对象的空间定位计算。因为物体只是有可能碰撞其附近的对象，而对于远端的对象可以通过空间划分加以排除，即忽略那些距离较远的、不可能相交的物体。对于空间划分和“视锥体剔除并限制渲染物体的数量”，它们之间存在着诸多类似之处。

可以采用平面结构执行空间划分，将空间分割为统一尺寸的单元格。这一过程也可以按照层次结构方式加以实现，其中空间被递归地划分为半空间，直至满足某种终止条件。物体对象将被置入上述网格或结构层次中。网格和层次结构的划分方式对于精确测试中的对组合测试也是十分有效的，特别是对象比较复杂时。通常不采用物体之间的一一测试方式，空间划分将碰撞测试限定在彼此接近的两个物体的范围之内。物体及其空间划分将在第 6 章和第 7 章加以讨论。

在执行代价高昂的几何体测试之前，预先计算耗费较少的体空间测试是一个较好的方法，这将减少碰撞过程中的计算量。例如，当封闭的包围球应用于所有物体上时，计算过程将呈现为简单的球体间的碰撞测试：如果球体间没有产生交叉重叠，则无须进一步执行几何体之间的复杂测试。包围体将在第 4 章加以讨论。

假设物体处于运动中，则每一帧之间物体会产生一个较小的位移量，针对于此，引入了第 3 种优化方法：时间一致性（或称作帧一致性）。例如，只是自上一帧开始处的运动物体需要执行测试，而对于其他物体，其碰撞信息需要保持一致。时间一致性还应允许缓存并复用当前帧的数据和计算结果，从而提高测试速度。另外，如果物体可以“瞬移”至任一位置，那么，基于这种运动一致性的假设则肯定是无效的。一致性的进一步讨论可参见第 9 章。

最后，针对特定平台结构的优化也是非常重要的。多种平台都支持某些类型的代码和数据的并行

计算，充分利用这一特性将极大地提高运行速度。由于 CPU 操作与主存向其提供操作数据之间存在着显著的速度差异，主存中碰撞几何体和其他数据的存储方式，也将对碰撞系统产生较为明显的速度影响。上述问题将在第 13 章中加以讨论。

2.6　健壮性

碰撞检测是空间几何应用程序的一种，这里，健壮性问题是非常重要的。在本书中，健壮性是指程序处理数字计算和复杂几何结构的相应能力。当出现复杂的数据输入时，健壮的计算程序可以提供相应的期望值，否则，可能会出现系统崩溃或产生死循环。相关问题可以划分为两类：数字精度的丢失和错误的几何形状。

数字精度问题源于计算过程中变量的有限精度。例如，当中间计算结果值大于浮点变量或整型变量所能表达的范围，其值将无效。如果该问题未被检测到，计算的最终结果很可能是错误的。健壮的计算实现方式必须防止这种问题的发生，或者即使发生了上述错误，系统也能够进行相应的调整并返回正确的结果。

几何健壮性应确保拓扑关系的完整性以及几何完整性。这类问题常常来源于错误的几何图形或退化的几何图形，从而导致错误的数字计算结果。在某种程度上，大多数算法期望具有良好格式的输入值。当发生错误的几何输入时（如退化至一点的三角形，或顶点不共面的多边形），如果这种情况不加以捕捉并即时处理，其计算结果将是未知的。

数字健壮性和几何健壮性的差别有时难以确定，甚至相互作用。为了避免这种隐藏的、难以修复的错误，健壮性问题应在碰撞检测系统的整个设计与开发过程中加以考虑。第 11 章和第 12 章将从更深层次的角度讨论健壮性问题。

2.7　实现与使用的简洁性

对于一个处于前期规划的碰撞检测系统来说，确定预期开发时间将与计算期望特征集具有同等的重要性。例如，游戏开发常常是预算紧张，时间紧迫，任何一个细节的延期都将付出高昂的代价。当估算系统实现的简洁性时，不应只考虑算法的复杂度，还应考虑系统中特例的数量、种类；需要调整尺寸的变量的数量（如导致数字误差的变量），以及其他影响系统开发时间的限制。

下面简述一些与碰撞检测系统使用相关的问题。例如，系统的通用性如何？系统是否可以处理变尺寸的物体？是否支持区间范围查询？构造碰撞相关的数据结构的耗时如何？对于最后一个问题，虽然预处理的耗时与运行时的性能无关，但在设计与制作阶段，该问题仍然是比较重要的。在整个开发阶段，将会频繁地更改模型，其较长的处理过程将会在一定程度上削弱生产力并延缓后期的测试。解决这一类问题，可以考虑在开发阶段使用一类快速、非优化的数据结构框架，而对于“non-debug”建立阶段，则采用一种慢速、最优的框架。

2.7.1 碰撞检测系统的调试

就像程序代码一样，碰撞检测系统也容易受到各种错误的影响。当然，发现这些错误往往比较困难并且十分耗时。在开发阶段，可以采取相应的步骤来减轻调试过程带来的痛苦，下面是一些有用的建议：

- ☑ 对于最近的 n 次碰撞查询，保持一个循环缓冲区，对应于若干秒（或更多）的数据值。当运行过程中出现某种错误时，可以暂停程序并且输出上述数据，以供进一步分析。例如，步进调试某些带有保存参数的调用。另外，当触发“断言”时，所记录的数据也会提供一些有用的信息。
- ☑ 提供碰撞几何体的可视化方案。例如，可以显示测试面、碰撞属性值，以及面的层次结构和分组。另外，还可以显示由前述的循环缓冲区提供的碰撞查询历史。这种可视化方案使得定位错误的碰撞查询变得十分简便。
- ☑ 实现一个简单的参考算法（如蛮力算法，该算法将对所有物体或多边形进行一一测试），并对应于其高级算法加以运行。如果运行结果出现差异，则算法一定产生了问题（一般位于高级算法中）。
- ☑ 维护一个测试查询组，当代码改变时，可以对照测试查询组运行碰撞测试代码。具有几何性质的代码一般都存在多种特例，因此，全面的测试将有助于早期捕捉问题。无论何时发现 bug，都需要添加测试用例以检测其是否被重引入。

当然，也可以采用诸如 assert()这种通用的调试策略。关于调试策略更多的讨论，可以参见【McConnell93，第 26 章】。

2.8 小结

本章简要介绍了碰撞检测系统设计与实现过程中须考虑的诸多因素，同时，还讨论了合理的碰撞几何体表达方式，以及是否应使用渲染几何体或某些特定几何体进行碰撞测试。碰撞处理过程分为两步执行：精确测试和粗略测试。其中，粗略测试将潜在碰撞对象划分为若干个子集，而精确测试则在对象间执行进一步的组合碰撞测试。精确碰撞测试将在第 4 章和第 5 章加以详细讨论，其中，将针对多种几何体对象表达方式的查询类型进行深入的分析。同时，在第 9 章中还将对该问题加以阐述。粗略测试算法将在第 6～8 章中讨论。考虑到健壮性问题的重要性，本书将在第 11、12 章中对这一问题进行全面的讲解。另外，本书主要侧重于实时应用程序的碰撞检测过程，因而计算性能也将会作为一个重点考查对象。从某种角度上讲，本书的主要篇幅都集中于性能的考量，因而关于高效的算法和数据结构的讨论将会贯穿全书。最后，本书将在第 13 章探讨相应的优化操作并以此作为本书的完结篇。

第 3 章　数学和几何学入门

碰撞检测领域具有较强的几何空间特性。例如，在场景世界的模拟过程中，场景以及其中的物体一般采用诸如多边形、球体和盒体等几何实体加以表达。为了实现实体间高效的相交测试，一般需要掌握向量、矩阵和线性代数等相关内容。本书假设读者对上述内容具有一定的了解，但为了方便，本章所涵盖的知识点将作为一个快速回顾供读者使用，包括相关概念、定义以及全书中使用的数学等式，相关内容只是一个非正式的复习材料，而非综合、正式的论述。对于向量空间、线性代数和几何学，有兴趣的读者可以阅读[Hausner98]或[Anton00]，以获取更加正式、全面的内容。

本章还给出了计算几何（如 Voronoi 域和凸包）和凸集理论（如分割面、支撑映射、Minkowski 和、Minkowski 差）中的一些概念，这些概念对于本书中的许多算法都是十分重要的。

3.1　矩阵

矩阵 **A** 是一个 $m\times n$ 的矩形数字阵列，包含 m 行，n 列：

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}=\left[a_{ij}\right]$$

矩阵中的 a_{ij} 项是指位于阵列中的第 i 行、第 j 列的元素。一个 $m\times n$ 的矩阵也称作 $m\times n$ 阶矩阵。如果 $m=n$，则称为方阵（n 阶）。只包含单一行的矩阵称作行矩阵；只包含单一列的矩阵称作列矩阵：

$$\mathbf{A}=\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix},\ \mathbf{B}=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

矩阵常视为由一定数量的行矩阵和列矩阵构成的。相应地，行矩阵和列矩阵也常常称作行向量和列向量。对于一个方阵，当矩阵元素下标 $i=j$ 时（即 $a_{11},a_{22},\cdots,a_{nn}$），该元素称为矩阵的主对角线项。若 $a_{ij}=0$，且 $i\neq j$，则矩阵称作对角矩阵：

$$\mathbf{A}=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

若矩阵是方阵且为对角矩阵，对角线上的元素为 1，同时其他项皆为 0，则称该矩阵为单位矩阵，表示为 **I**。对于方阵 **L**，若主对角线上方的元素皆为 0，则称该矩阵为下三角矩阵；对于方阵 **U**，若主

对角线下方的元素皆为 0，则称该矩阵为上三角矩阵，例如：

$$\mathbf{I}=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix},\quad \mathbf{L}=\begin{bmatrix}2&0&0\\1&-2&0\\-5&0&-1\end{bmatrix},\quad \mathbf{U}=\begin{bmatrix}1&5&4\\0&2&1\\0&0&3\end{bmatrix}$$

矩阵 **A** 的转置矩阵写作 $\mathbf{A}^T$，可以通过交换行、列元素得到，反之亦然。即，矩阵 **A** 的转置矩阵 **B** 记作：$b_{ij}=a_{ji}$。

$$\mathbf{A}=\begin{bmatrix}5&2\\-3&-1\\0&-4\end{bmatrix},\quad \mathbf{B}=\mathbf{A}^T=\begin{bmatrix}5&-3&0\\2&-1&-4\end{bmatrix}$$

如果有 $\mathbf{A}^T=\mathbf{A}$，则称矩阵是对称的，即对于所有的 i 和 j，有 $a_{ij}=a_{ji}$；若 $\mathbf{A}^T=-\mathbf{A}$，则称矩阵是斜对称的（或反对称）。

3.1.1 矩阵运算

给定两个 $m\times n$ 矩阵 $\mathbf{A}=[a_{ij}]$和 $\mathbf{B}=[b_{ij}]$，则矩阵加法（$\mathbf{C}=[c_{ij}]=\mathbf{A}+\mathbf{B}$）定义为：每一个矩阵对应位置的元素的加法，$c_{ij}=a_{ij}+b_{ij}$，或：

$$\mathbf{C}=\mathbf{A}+\mathbf{B}=\begin{bmatrix}a_{11}&a_{12}&\cdots&a_{1n}\\a_{21}&a_{22}&\cdots&a_{2n}\\\vdots&\vdots&\ddots&\vdots\\a_{m1}&a_{m2}&\cdots&a_{mn}\end{bmatrix}+\begin{bmatrix}b_{11}&b_{12}&\cdots&b_{1n}\\b_{21}&b_{22}&\cdots&b_{2n}\\\vdots&\vdots&\ddots&\vdots\\b_{m1}&b_{m2}&\cdots&b_{mn}\end{bmatrix}$$

$$=\begin{bmatrix}a_{11}+b_{11}&a_{12}+b_{12}&\cdots&a_{1n}+b_{1n}\\a_{21}+b_{21}&a_{22}+b_{22}&\cdots&a_{2n}+b_{2n}\\\vdots&\vdots&\ddots&\vdots\\a_{m1}+b_{m1}&a_{m2}+b_{m2}&\cdots&a_{mn}+b_{mn}\end{bmatrix}=\left[a_{ij}+b_{ij}\right]$$

矩阵的减法可以类似地定义。矩阵的乘法存在两种格式：如果 c 是一个标量，则标量乘积为 $\mathbf{B}=c\mathbf{A}$，其中，$[b_{ij}]=[ca_{ij}]$。例如：

$$\mathbf{B}=4\mathbf{A}=4\begin{bmatrix}-2&0&1\\5&-3&-1\end{bmatrix}=\begin{bmatrix}-8&0&4\\20&-12&-4\end{bmatrix},\quad 其中，\mathbf{A}=\begin{bmatrix}-2&0&1\\5&-3&-1\end{bmatrix}$$

若 **A** 是一个 $m\times n$ 矩阵，**B** 是一个 $n\times p$ 矩阵，则矩阵乘法（$\mathbf{C}=\mathbf{AB}$）定义为：

$$c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}$$

例如：

$$\mathbf{C}=\mathbf{AB}=\begin{bmatrix}3&0&-5\\2&-1&-4\end{bmatrix}\begin{bmatrix}3&-1\\4&7\\2&-2\end{bmatrix}=\begin{bmatrix}-1&7\\-6&-1\end{bmatrix}$$

若以向量（定义于 3.3 节）的角度来看，c_{ij} 为矩阵 **A** 第 i 行元素与矩阵 **B** 第 j 列的点积值。矩阵乘法一般不满足交换律，即 $\mathbf{AB}\neq\mathbf{BA}$。矩阵算术运算并没有定义除法操作，但某些矩阵存在逆矩阵，记

为 inv(**A**)或 $\mathbf{A}^{-1}$，并有 $\mathbf{AA}^{-1}=\mathbf{A}^{-1}\mathbf{A}=\mathbf{I}$。不存在逆的矩阵则称作奇异矩阵（或不可逆矩阵）。

3.1.2　矩阵的几何代数符号

给定标量 r 和 s，以及矩阵 **A**、**B**、**C**，下列运算符给出了矩阵加法、矩阵减法以及矩阵的标量乘法：

$$\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$$
$$\mathbf{A}+(\mathbf{B}+\mathbf{C})=(\mathbf{A}+\mathbf{B})+\mathbf{C}$$
$$\mathbf{A}-\mathbf{B}=\mathbf{A}+(-\mathbf{B})$$
$$-(-\mathbf{A})=\mathbf{A}$$
$$s(\mathbf{A}\pm\mathbf{B})=s\mathbf{A}\pm s\mathbf{B}$$
$$(r\pm s)\mathbf{A}=r\mathbf{A}\pm s\mathbf{A}$$
$$r(s\mathbf{A})=s(r\mathbf{A})=(rs)\mathbf{A}$$

对于矩阵乘法，有下列运算操作：

$$\mathbf{AI}=\mathbf{IA}=\mathbf{A}$$
$$\mathbf{A}(\mathbf{BC})=(\mathbf{AB})\mathbf{C}$$
$$\mathbf{A}(\mathbf{B}\pm\mathbf{C})=\mathbf{AB}\pm\mathbf{AC}$$
$$(\mathbf{A}\pm\mathbf{B})\mathbf{C}=\mathbf{AC}\pm\mathbf{BC}$$
$$(s\mathbf{A})\mathbf{B}=s(\mathbf{AB})=\mathbf{A}(s\mathbf{B})$$

最后，对于矩阵的转置，有下列运算操作：

$$(\mathbf{A}\pm\mathbf{B})^T=\mathbf{A}^T\pm\mathbf{B}^T$$
$$(s\mathbf{A})^T=s\mathbf{A}^T$$
$$(\mathbf{AB})^T=\mathbf{B}^T\mathbf{A}^T$$

3.1.3　行列式

矩阵 **A** 的行列式是与 **A** 相关的一组数字，记作 det(**A**)或|**A**|。行列式常用于线性方程式的求解计算，这将在 3.1.4 节加以讨论。本书将重点考查 2×2 和 3×3 矩阵。对于 3×3 矩阵，其行列式按如下方式加以计算：

$$|\mathbf{A}|=|u_1|=u_1,$$
$$|\mathbf{A}|=\begin{vmatrix}u_1 & u_2\\ v_1 & v_2\end{vmatrix}=u_1v_2-u_2v_1,$$
$$|\mathbf{A}|=\begin{vmatrix}u_1 & u_2 & u_3\\ v_1 & v_2 & v_3\\ w_1 & w_2 & w_3\end{vmatrix}=u_1(v_2w_3-v_3w_2)+u_2(v_3w_1-v_1w_3)$$
$$+u_3(v_1w_2-v_2w_1)=\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w})$$

（符号 • 和×分别代表点积和叉积，相关内容将在 3.3.3 节和 3.3.5 节中加以讨论）在几何学中，行列式一般与（方向性的）超体积有关。超体积是一种较为抽象的概念，对于一个 n 维空间的 $n\times n$ 行列式，分别代表 1D、2D 和 3D 空间的长度、面积和体积。例如，对于一个 1×1 的矩阵，$\mathbf{A}=[u_1]$，|**A**|则代

表原点至 u_1 的线段的有符号距离长度；对于一个 2×2 矩阵：

$$\mathbf{A}=\begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}$$

|**A**|则代表原点(0,0)和点(u_1,u_2)、点(v_1,v_2)确定的平行四边形的有符号面积。如果平行四边形的首顶点至第二个顶点是逆时针方向的，则行列式的值为正，否则为负值。对于一个 3×3 矩阵 **A**=[**u v w**]（其中，**u**、**v**、**w** 为列向量），|**A**|则代表 3 个向量确定的平行六面体的有符号体积。通常，对于一个 n 维矩阵，其行列式对应着：由列向量确定的 n 维超平行六面体的有符号体积。

本书将大量采用行列式这一概念，其中，需要注意以下一些重要内容。

- ☑ 如果矩阵被转置，则行列式保持不变：

$$|\mathbf{A}|=|\mathbf{A}^T|$$

- ☑ 交换矩阵 **A** 的两行或两列得到矩阵 **B**，则有|**B**| = −|**A**|（互换两行或两列改变了超体的构造方式，因而改变了其行列式的符号）。
- ☑ 若将行列式的某一行（或列）的倍数加到另一行（或列）对应的元素上，则行列式的值不变，有|**B**| = |**A**|（只是改变了超体某一个面的倾斜程度，而保持体积不变）。
- ☑ 矩阵 **A** 和矩阵 **B** 的乘积的行列式，等于各自行列式的乘积，即|**AB**| = |**A**| |**B**|。
- ☑ 若矩阵 **A** 的一行（或一列）乘以一个常量系数 k，则有|**B**| = k|**A**|。
- ☑ 如果矩阵 **A** 中的一行为另一行的倍数，则有|**A**|=0。对于列，则同理。
- ☑ 对于包含全 0 行或全 0 列的行列式，则有|**A**| =0。

一种计算行列式的有效方法是：利用矩阵的行、列操作，将其降至三角矩阵（其中主对角线之上或之下的全部元素为 0）。行列式的最终结果为主对角线上各元素的乘积。例如，有如下行列式：

$$\mathbf{A}=\begin{bmatrix} 4 & -2 & 6 \\ 2 & 5 & 0 \\ -2 & 1 & -4 \end{bmatrix}$$

且可按下列方式加以计算：

$$\det(\mathbf{A})=\begin{vmatrix} 4 & -2 & 6 \\ 2 & 5 & 0 \\ -2 & 1 & -4 \end{vmatrix}$$

将第 2 行数据加至第 3 行

$$=\begin{vmatrix} 4 & -2 & 6 \\ 2 & 5 & 0 \\ 0 & 6 & -4 \end{vmatrix}$$

将第 1 行数据乘以−1/2 并加至第 2 行

$$=\begin{vmatrix} 4 & -2 & 6 \\ 0 & 6 & -3 \\ 0 & 6 & -4 \end{vmatrix}$$

将第 1 行数据乘以−1 并加至第 3 行

$$=\begin{vmatrix} 4 & -2 & 6 \\ 0 & 6 & -3 \\ 0 & 0 & -1 \end{vmatrix}$$

计算三角矩阵，最终行列式的结果为主对角线上各元素的乘积

$$=-24$$

行列式的一种典型应用方式是利用克莱姆法则计算线性方程组，其具体内容将在3.1.4 节加以讨论。

行列式也可以通过“展开余子式”加以计算。首先，通过消除第 i 行和第 j 列得到矩阵的行列式 m_{ij}，即余子式，则代数余子式 c_{ij} 为：

$$c_{ij}=(-1)^{i+j}m_{ij}$$

则 **A** 的行列式可表示为：

$$|\mathbf{A}|=\sum_{j=1}^{n}a_{rj}c_{rj}=\sum_{i=1}^{n}a_{ik}c_{ik}$$

其中，r 和 k 代表任一行或列的索引。例如，给定前述矩阵 **A**，则|**A**|计算为：

$$|\mathbf{A}|=\begin{vmatrix}4 & -2 & 6\\ 2 & 5 & 0\\ -2 & 1 & -4\end{vmatrix}=4\begin{vmatrix}5 & 0\\ 1 & -4\end{vmatrix}-(-2)\begin{vmatrix}2 & 0\\ -2 & -4\end{vmatrix}+6\begin{vmatrix}2 & 5\\ -2 & 1\end{vmatrix}=-80-16+72=-24$$

3.1.4　利用克莱姆法则计算线性方程组

对于未知变量 x 和 y，考查下面两个线性方程：

$$ax+by=e,\quad cx+dy=f$$

分别乘以 d 和 b，得到：

$$adx+bdy=de,\quad bcx+bdy=bf$$

从第一个方程中减去第二个方程，得到 $adx-bcx=de-bf$，则 $x=(de-bf)/(ad-bc)$。同理，$y=(af-ce)/(ad-bc)$。

该方程组的解对应着计算两条直线的交点，因此该解存在 3 种情况：两直线相交于一点；两直线平行且不存在交点；两直线平行且重合。只有第一种情况使得方程组存在唯一的解，同时分母 $ad-bc$ 非 0。上述方程组可以写成矩阵等式形式 $\mathbf{AX}=\mathbf{B}$，其中：

$$\mathbf{A}=\begin{bmatrix}a & b\\ c & d\end{bmatrix},\quad \mathbf{X}=\begin{bmatrix}x\\ y\end{bmatrix},\quad \mathbf{B}=\begin{bmatrix}e\\ f\end{bmatrix}$$

A 称作系数矩阵，**X** 称作求解向量，**B** 称作常数向量。当且仅当系数矩阵的行列式不为 0，即|**A**|≠0 时，线性方程组有唯一的解。在本例中，$\mathbf{X}=\mathbf{A}^{-1}\mathbf{B}$。

考查前述 x 和 y 的求解值，很明显，对应解可以表达为行列式之比：

$$x=\frac{\begin{vmatrix}e & b\\ f & d\end{vmatrix}}{\begin{vmatrix}a & b\\ c & d\end{vmatrix}},\quad y=\frac{\begin{vmatrix}a & e\\ c & f\end{vmatrix}}{\begin{vmatrix}a & b\\ c & d\end{vmatrix}}$$

这里，分母为系数矩阵的行列式；对于 x，分子也与系数矩阵的行列式相关，其中，第一列被常数向量替换；对于 y，第二列被常数向量替换。

上述求解过程可以以同一方式扩展至更复杂的方程组，变量求解为：系数矩阵行列式中的变量列分别被常数向量替代，再除以原系数矩阵的行列式。例如，对于下列方程组：

$$a_1x+b_1y+c_1z=d_1,\quad a_2x+b_2y+c_2z=d_2,\quad a_3x+b_3y+c_3z=d_3$$

利用克莱姆法则求解为：

$$x=\frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{d},\quad y=\frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{d},$$

$$z=\frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{d},\quad d=\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

利用克莱姆法则求解线性方程组时，建议等式的数量不要超过 3 个或 4 个，否则涉及的计算量将会显著增加。对于较大的方程组，较好的方法是使用高斯消元法。然而，对于小型的方程组，克莱姆法则效果良好并易于使用。本书中所讲解的线性方程组，都是这种小型的方程组。

3.1.5 2×2 矩阵和 3×3 矩阵的逆矩阵

行列式也与逆矩阵的计算公式相关。详细地讨论逆矩阵的计算超出了本书的范围，而对于本书，下列逆矩阵的计算已经足够。注意，对于 2×2 矩阵和 3×3 矩阵，其逆矩阵写作：

$$\mathbf{A}^{-1}=\frac{1}{\det(\mathbf{A})}\begin{bmatrix} u_{22} & -u_{12} \\ -u_{21} & u_{11} \end{bmatrix}$$

$$\mathbf{A}^{-1}=\frac{1}{\det(\mathbf{A})}\begin{bmatrix} u_{22}u_{33}-u_{23}u_{32} & u_{13}u_{32}-u_{12}u_{33} & u_{12}u_{23}-u_{13}u_{22} \\ u_{23}u_{31}-u_{21}u_{33} & u_{11}u_{33}-u_{13}u_{31} & u_{13}u_{21}-u_{11}u_{23} \\ u_{21}u_{32}-u_{22}u_{31} & u_{12}u_{31}-u_{11}u_{32} & u_{11}u_{22}-u_{12}u_{21} \end{bmatrix}$$

从上述计算公式中可以看出，如果矩阵 **A** 的行列式等于 0，则 inv(**A**)不存在，因为这将产生除 0 错误（不仅是 2×2 矩阵和 3×3 矩阵，对于任意尺寸的方阵，这种错误都应避免）。对于 3×3 矩阵的逆矩阵也可以采用更具几何特征的表达方式。令矩阵 **A** 包含 3 个列向量 **u**、**v** 和 **w**：

$$\mathbf{A}=\begin{bmatrix}\mathbf{u} & \mathbf{v} & \mathbf{w}\end{bmatrix}$$

则矩阵 **A** 的逆矩阵为：

$$\mathbf{A}^{-1}=\begin{bmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{bmatrix}^T$$

其中，**a**、**b** 和 **c** 分别为列向量：

$$\mathbf{a}=(\mathbf{v}\times\mathbf{w})/(\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w}))$$
$$\mathbf{b}=(\mathbf{w}\times\mathbf{u})/(\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w}))$$
$$\mathbf{c}=(\mathbf{u}\times\mathbf{v})/(\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w}))$$

一般地，若矩阵 **A** 的逆矩阵 inv(**A**)存在，则 inv(**A**)可以表达为：

$$\mathrm{inv}(\mathbf{A})=\frac{1}{\det(\mathbf{A})}\mathbf{M}$$

其中，**M** 称作矩阵 **A** 的伴随矩阵，记作 adj(**A**)。

3.1.6　行列式断言

在几何测试的某些简洁表达中，行列式的计算是非常有用的。一些几何测试可以转化为行列式的计算格式。因此，如果行列式的求解过程健壮而高效，那么对几何测试的计算也是很有帮助的。特别地，行列式的正负性在许多几何测试中扮演着重要的角色，并常用作拓扑断言。例如，方向测试、相对于面的位置测试，以及点的剔除测试。需注意的是，行列式断言的直接求解（如未采用公共子表达式消元）一般无法得到高效、健壮的算式。还需要注意，采用浮点运算的行列式断言对舍入错误是十分敏感的；同时，对于在简化计算过程中依赖正确正负号的算法来说，有时很可能无法得到期望的运算结果。对于更多的健壮性错误及其处理方法，可参见第 11 章和第 12 章。基于上述事项，下面将阐述拓扑断言的一些具体使用方法，从而体现行列式的应用。

3.1.6.1　ORIENT2D(*A*,*B*,*C*)

令 $A=(a_x,a_y)$，$B=(b_x,b_y)$，$C=(c_x,c_y)$为 3 个 2D 点，同时，ORIENT2D(*A*,*B*,*C*)定义为：

$$\mathrm{ORIENT2D}(A,B,C)=\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}=\begin{vmatrix} a_x-c_x & a_y-c_y \\ b_x-c_x & b_y-c_y \end{vmatrix}$$

如果 ORIENT2D(*A*,*B*,*C*) > 0，则 *C* 位于有向直线 *AB* 的左侧，相应地，三角形 *ABC* 的顶点为逆时针旋转；如果 ORIENT2D(*A*,*B*,*C*) < 0，则 *C* 位于有向直线 *AB* 的右侧，且三角形 *ABC* 顶点为顺时针旋转；如果 ORIENT2D(*A*,*B*,*C*) = 0，则三点共线。ORIENT2D(*A*,*B*,*C*)的实际返回值将是三角形 *ABC* 有符号面积的 2 倍（如果三角形 *ABC* 逆时针旋转，则为正值，否则为负值）。另外，该行列式可以看作 2D 直线 $L(x,y)=0$ 的隐式方程，该直线包括线上两点 $A=(a_x,a_y)$和 $B=(b_x,b_y)$，并有下列定义：

$$L(x,y)=\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ x & y & 1 \end{vmatrix}$$

3.1.6.2　ORIENT3D(*A*,*B*,*C*,*D*)

给定 4 个 3D 顶点 $A=(a_x,a_y,a_z)$，$B=(b_x,b_y,b_z)$，$C=(c_x,c_y,c_z)$和 $D=(d_x,d_y,d_z)$，且 ORIENT3D(*A*,*B*,*C*,*D*)定义为：

$$\begin{aligned}\mathrm{ORIENT3D}(A,B,C,D)&=\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}=\begin{vmatrix} a_x-d_x & a_y-d_y & a_z-d_z \\ b_x-d_x & b_y-d_y & b_z-d_z \\ c_x-d_x & c_y-d_y & c_z-d_z \end{vmatrix}\\ &=(A-D)\bullet((B-D)\times(C-D))\end{aligned}$$

如果 ORIENT3D(*A*,*B*,*C*,*D*)<0，则顶点 *D* 位于三角形 *ABC* 之上，并且当从顶点 *D* 处观察时，三角形 *ABC* 呈逆时针限制状态；如果 ORIENT3D(*A*,*B*,*C*,*D*)>0，则顶点 *D* 位于三角形 *ABC* 之下；如果 ORIENT3D(*A*,*B*,*C*,*D*)=0，则 4 点共面。ORIENT3D(*A*,*B*,*C*,*D*)的返回值将是有符号四面体体积的 4 倍。

另外，该行列式可以看作 3D 面 P(x,y,z)=0 的隐式方程，且该面包含 3 顶点 $A=(a_x,a_y,a_z)$，$B=(b_x,b_y,b_z)$和 $C=(c_x,c_y,c_z)$，并定义 P(x,y,z)为：

$$\mathrm{P}(x,y,z)=\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ x & y & z & 1 \end{vmatrix}$$

3.1.6.3 INCIRCLE2D(*A*,*B*,*C*,*D*)

给定 4 个 2D 顶点 $A=(a_x,a_y)$，$B=(b_x,b_y)$，$C=(c_x,c_y)$和 $D=(d_x,d_y)$，且定义 INCIRCLE2D(A,B,C,D)为：

$$\mathrm{INCIRCLE2D}(A,B,C,D)=\begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix}$$

$$=\begin{vmatrix} a_x-d_x & a_y-d_y & (a_x-d_x)^2+(a_y-d_y)^2 \\ b_x-d_x & b_y-d_y & (b_x-d_x)^2+(b_y-d_y)^2 \\ c_x-d_x & c_y-d_y & (c_x-d_x)^2+(c_y-d_y)^2 \end{vmatrix}$$

令三角形 ABC 以逆时针方向旋转，即 ORIENT2D(A,B,C)>0。如果 INCIRCLE2D(A,B,C,D)>0，则顶点 D 位于圆的内部，且该圆通过顶点 A，B，C；如果 INCIRCLE2D(A,B,C,D)<0，则顶点 D 位于圆的外部；如果 INCIRCLE2D(A,B,C,D)=0，则 4 点共圆。如果 ORIENT2D(A,B,C)<0，则上述结果相反。

3.1.6.4 INSPHERE(*A*,*B*,*C*,*D*,*E*)

给定 5 个 3D 顶点 $A=(a_x,a_y,a_z)$，$B=(b_x,b_y,b_z)$，$C=(c_x,c_y,c_z)$，$D=(d_x,d_y,d_z)$和 $E=(e_x,e_y,e_z)$，并且 INSPHERE(A,B,C,D,E)定义为：

$$\mathrm{INSPHERE}(A,B,C,D,E)=\begin{vmatrix} a_x & a_y & a_z & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2+d_y^2+d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2+e_y^2+e_z^2 & 1 \end{vmatrix}$$

$$=\begin{vmatrix} a_x-e_x & a_y-e_y & a_z-e_z & (a_x-e_x)^2+(a_y-e_y)^2+(a_z-e_z)^2 \\ b_x-e_x & b_y-e_y & b_z-e_z & (b_x-e_x)^2+(b_y-e_y)^2+(b_z-e_z)^2 \\ c_x-e_x & c_y-e_y & c_z-e_z & (c_x-e_x)^2+(c_y-e_y)^2+(c_z-e_z)^2 \\ d_x-e_x & d_y-e_y & d_z-e_z & (d_x-e_x)^2+(d_y-e_y)^2+(d_z-e_z)^2 \end{vmatrix}$$

令顶点 A，B，C，D 的方向性遵循 ORIENT3D(A,B,C,D)>0。如果 INSPHERE(A,B,C,D,E)>0，则顶点 E 位于球体之内，且顶点 A，B，C，D 位于球面上；如果 INSPHERE(A,B,C,D,E)<0，则顶点 E 位于球体之外；如果 INSPHERE(A,B,C,D,E)=0，则 5 点共体。

3.2　坐标系统和顶点

顶点代表了空间中的某一位置，其定位方式采用坐标系统加以描述：给定一个称作原点的参考点和几个坐标轴。在 n 维坐标系统中，顶点由 n 元实数加以确定：$(x_1,x_2,\cdots,x_n)$。该 n 元称作顶点的坐标。由 n 元描述的顶点可描述为：沿第一个坐标轴并由原点开始移动 x_1 个单位；沿第二个坐标轴并由原点开始移动 x_2 个单位，对于其他实数，依此类推。而原点的各个元素皆为 0：$(0,0,\cdots,0)$。任何一个坐标系统都可以相对于父坐标系统加以定义，而子系统的原点可对应于父系统中的任何一点。

首先讨论的是笛卡儿坐标系统，其中，坐标轴彼此垂直。对于 2D 空间，两个坐标轴一般记为 x 轴和 y 轴；在 3D 空间中，第 3 个轴通常称作 z 轴。

坐标空间是指坐标系统可以确定的点集，同时，坐标系统跨越该空间。一个给定的、跨越空间的坐标轴集合，称作该空间的参考坐标系或参考基准。对于一个给定的坐标空间，将有无限多个参考坐标系。

在本书中，点将采用大写字母并以斜体加以标记（如 P、Q 和 R）。顶点与向量的关系十分密切，这将在 3.3 节中加以讨论。

3.3　向量

抽象地讲，向量定义为向量空间元，即向量空间以元素集合（向量集）的方式加以定义，并支持诸如向量加法、标量乘等向量操作，其元素和操作都遵循相关的定理。因此，理论上，$m\times n$ 实数矩阵即是向量空间中的元素（向量）集。然而，在本书的实际操作中，向量一般属于向量空间 $\mathbb{R}^n$，其元素为 n 元实数。向量 $\mathbf{v}$ 以下列方式给出：

$$\mathbf{v}=(v_1,v_2,\cdots,v_n)$$

其中，v_1，$v_2\cdots v_n$ 称作向量 $\mathbf{v}$ 的分量。一般情况下，本书中的向量限制为 $\mathbb{R}^2$ 和 $\mathbb{R}^3$，因而通常为二元或三元实数向量。

一种非正式的描述是：向量是一种包含方向和大小的实体，因此，其图形表示常常是一个箭头；或者其几何描述为一个有向线段，指向一个给定的方向并包含一个给定的长度。本书中，向量采用黑体小写字母加以表示（如 $\mathbf{u}$、$\mathbf{v}$、$\mathbf{w}$）。

向量 $\mathbf{v}$ 可以解释为从原点到顶点 P（直观地讲，是位置 P）的位移。按照这种解释，$\mathbf{v}$ 在某种意义上被约束至原点 O。采用顶点位置加以描述的向量称作位置向量，或束缚向量。向量 $\mathbf{v}$ 还可以解释为从起点 P 至终点 O 的位移，其中 $Q=P+\mathbf{v}$。从这种意义上讲，$\mathbf{v}$ 可以自由地应用于任一顶点之上。以任意位移方式表示的向量称作自由向量。如果原点发生改变，束缚向量将会随之改变，而自由向量则保持不变。如果两个向量具有相同的方向和大小，则称两个向量相等，即二者的分量相等。然而，如果两个束缚向量被约束至不同的原点，即使二者的分量相等，则二者也不相等。虽然自由向量和束缚向量上的多数操作都是相同的，但二者之间仍存在一些差别。例如，作为面法线向量的自由向量则与束缚向量存在不同的转换方式。

位置向量的存在意味着在顶点与向量之间存在一对一的关系。通常，总是存在一个固定的原点，因此，顶点和向量可以互相替换使用。

如图 3.1（a）所示，自由向量 **v** 可以表示为始于任意顶点的位移：从顶点 A 至顶点 B，或者从顶点 C 至顶点 D。如图 3.1（b）所示，两个束缚向量 **p** 和 **q** 确定了顶点 P 和顶点 Q 的相应位置——只是各自的位置。

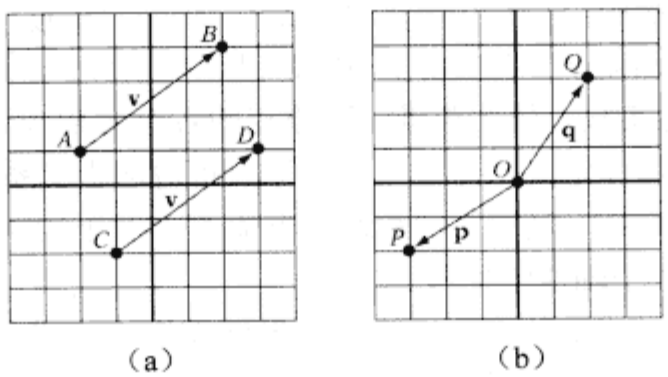


图 3.1　（a）（自由）向量 **v** 可描述为从任意顶点的一段位移。例如，从 A 至 B，从 C 至 D。
（b）位置向量可表示为基于原点的某一向量。这里，位置向量 **p** 和 **q** 分别定义顶点 P 和 Q 的位置

从顶点 A 至顶点 B 的向量 **v** 写作 $\overrightarrow{AB}$（等价于 $\mathbf{v}=\overrightarrow{OB}-\overrightarrow{OA}$）。有时，可以省去上方的箭头，简单表示为 $\mathbf{v}=AB$。一种特殊的情况是，P 指向自身，此时，该向量称作 0 向量并记作 0。

在使用向量时，实数作为标量采用小写斜体字母加以表示（如 a、b、c）。

3.3.1　向量运算

向量 **u** 和向量 **v** 之和 $\mathbf{w}=\mathbf{u}+\mathbf{v}$ 可以通过两个向量中的各个元素对应之和计算得到：

$$\mathbf{w}=\mathbf{u}+\mathbf{v}=(u_1,u_2,\cdots,u_n)+(v_1,v_2,\cdots,v_n)=(u_1+v_1,u_2+v_2,\cdots,u_n+v_n)$$

如图 3.2 所示，从几何角度上讲，向量和可以表示为箭头从向量 **u** 的起始点指向向量 **v** 的终点的有向线段。向量和的几何视角特征常称作向量相加的平行四边形法则，其中，向量和对应着两个向量形成的平行四边形的对角线。

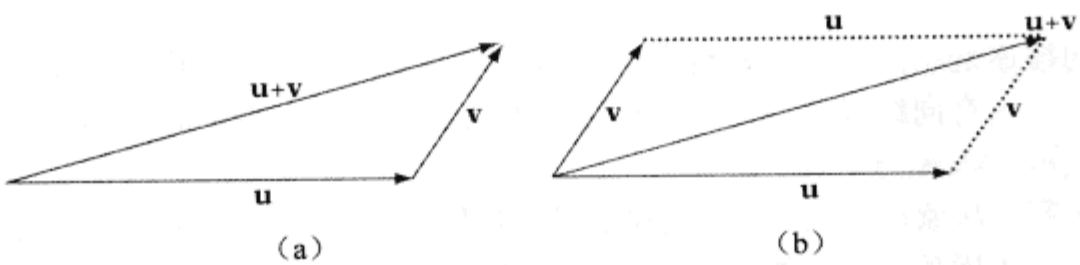


图 3.2　（a）从几何角度上讲，向量和可表示为：将向量 **u**、**v** 相连并连接向量的尾部和首部。
（b）另外，根据平行四边形法则，向量和也可视为平行四边形的对角线

向量减法 $\mathbf{w}=\mathbf{u}-\mathbf{v}$ 定义为 **u** 和 $-\mathbf{v}$ 的向量加法，即 $\mathbf{w}=\mathbf{u}+(-\mathbf{v})$。逆向量 $-\mathbf{v}$ 与向量 **v** 相比，具有相同的大小、相反的方向，并可以通过将向量 **v** 中的每一个元素取逆值而获得：

$$-\mathbf{v}=-(v_1,v_2,\cdots,v_n)=(-v_1,-v_2,\cdots,-v_n)$$

相应地，两个向量的减法可以定义为：

$$\mathbf{w}=\mathbf{u}-\mathbf{v}=(u_1,u_2,\cdots,u_n)-(v_1,v_2,\cdots,v_n)=(u_1-v_1,u_2-v_2,\cdots,u_n-v_n)$$

如图 3.3 所示，常数的向量乘法可以实现向量的缩放。对于结果向量 $\mathbf{w}=k\mathbf{v}$，其中，使用了常数 k 的标量乘法给定为：

$$\mathbf{w}=k\mathbf{v}=k(v_1,v_2,\cdots,v_n)=(kv_1,kv_2,\cdots,kv_n)$$

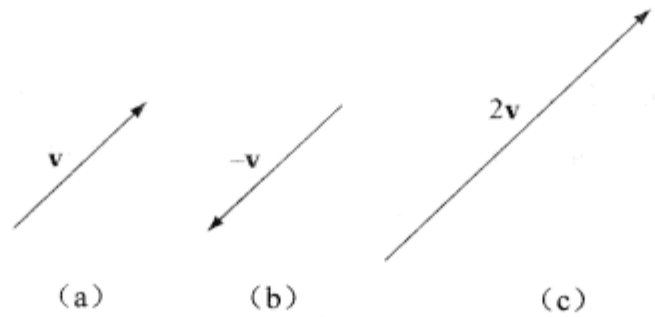


图 3.3　（a）向量 **v**。（b）向量 **v** 的逆向量。（c）缩放系数为 2 时的向量 **v**

当标量 k 取负值时，**w** 则与向量 **v** 的方向相反。向量 **v** 的长度标记为$\|\mathbf{v}\|$，并依据向量中的各元素定义为：

$$\|\mathbf{v}\|=\sqrt{(v_1^2+v_2^2+\cdots+v_n^2)}$$

向量 **v** 的长度也称作向量的模或向量的大小。大小为 1 的向量称作单位向量。非 0 向量可以计算其单位向量（或称作规范化）：乘以标量 $1/\|\mathbf{v}\|$。

对于任意的 n 维向量空间（记为 $\mathbb{R}^n$），都存在 n 个线性无关的基向量 $\mathbf{e}_1,\mathbf{e}_2,\cdots,\mathbf{e}_n$。任何一个向量 **v** 都可以写作上述基向量的线性组合：

$$\mathbf{v}=a_1\mathbf{e}_1+a_2\mathbf{e}_2+\cdots+a_n\mathbf{e}_n，其中，a_1,a_2,\cdots,a_n 为标量$$

给定一个向量集合，如果其中的一个向量能用别的向量的线性组合的形式表示出来，则这组向量是线性相关的，否则向量集合就是线性无关的。例如，给定两个线性无关的向量 $\mathbf{e}_1$ 和 $\mathbf{e}_2$，在这两个向量构成的平面中，向量 **v** 可以线性地表示为 $\mathbf{v}=a_1\mathbf{e}_1+a_2\mathbf{e}_2$，其中，$a_1$ 和 a_2 为常量。

这里，大多数基向量是标准正交的，即向量间两两垂直并且都是单位向量。$\mathbb{R}^3$ 中标准的基向量彼此正交，且包含向量(1,0,0)、(0,1,0)和(0,0,1)，通常标记为 **i**、**j** 和 **k**。

3.3.2　向量的代数恒等式

给定向量 **u**、**v**、**w**，下列恒等式代表了向量加法和向量减法：

$$\mathbf{u}+\mathbf{v}=\mathbf{v}+\mathbf{u}$$
$$(\mathbf{u}+\mathbf{v})+\mathbf{w}=\mathbf{u}+(\mathbf{v}+\mathbf{w})$$
$$\mathbf{u}-\mathbf{v}=\mathbf{u}+(-\mathbf{v})$$
$$-(-\mathbf{v})=\mathbf{v}$$
$$\mathbf{v}+(-\mathbf{v})=0$$
$$\mathbf{v}+0=0+\mathbf{v}=\mathbf{v}$$

另外，给定标量 r 和 s，下列运算符号代表了向量的标量乘法：

$$r(s\mathbf{v})=(rs)\mathbf{v}$$
$$(r+s)\mathbf{v}=r\mathbf{v}+s\mathbf{v}$$
$$s(\mathbf{u}+\mathbf{v})=s\mathbf{u}+s\mathbf{v}$$
$$1\mathbf{v}=\mathbf{v}$$

3.3.3 点积

向量 **u** 和向量 **v** 的点积（标量积）定义为：对应向量分量乘积之和，并记为 **u** • **v**。因此有：

$$\mathbf{u}\bullet\mathbf{v}=(u_1,u_2,\cdots,u_n)\bullet(v_1,v_2,\cdots,v_n)=u_1v_1+u_2v_2+\cdots+u_nv_n$$

需注意的是，点积是一个标量值，而非向量。向量自身的点积结果为向量长度的平方。

$$\mathbf{v}\bullet\mathbf{v}=v_1^2+v_2^2+\cdots+v_n^2=\|\mathbf{v}\|^2$$

对于向量 **u** 和向量 **v** 之间的（最小）夹角，满足下列公式：

$$\mathbf{u}\bullet\mathbf{v}=\|\mathbf{u}\|\,\|\mathbf{v}\|\cos\theta$$

则夹角 θ 为：

$$\theta=\cos^{-1}\frac{\mathbf{u}\bullet\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$$

结果表明，对于两个非 0 向量，当 θ 为锐角时，点积值为正值；θ 为钝角时，点积值为负值；当两个向量垂直时，点积值为 0（如图 3.4 所示）。这种可以判断两向量间夹角是否为锐角、钝角、直角的能力，是点积计算的一个非常有用的属性，并常用于各种几何测试中。

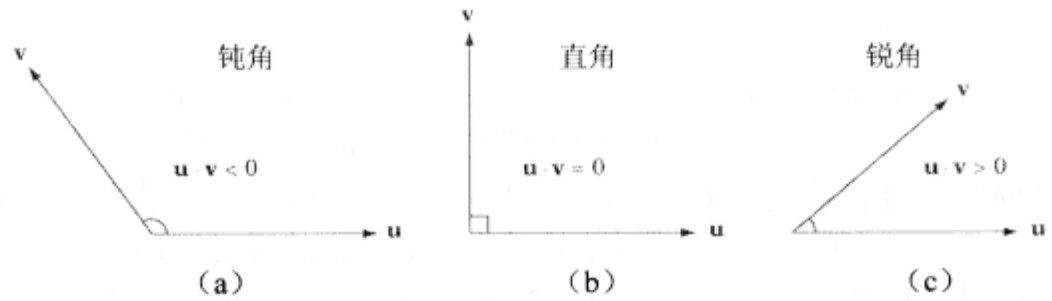


图 3.4　两个向量的点积符号表明了向量间的夹角。（a）钝角。（b）直角。（c）锐角

从几何角度，点积可以看作是向量 **v** 在向量 **u** 上的投影，并返回 **v** 沿 **u**、$\|\mathbf{u}\|$个单位的有符号距离 d：

$$d=\frac{\mathbf{u}\bullet\mathbf{v}}{\|\mathbf{u}\|}$$

该投影如图 3.5（a）所示。给定向量 **u** 和 **v**，向量 **v** 可以分解为平行于 **u** 的向量 **p**，以及垂直于 **u** 的向量 **q**，因此有 **v**=**p**+**q**：

$$\mathbf{p}=\frac{\mathbf{u}\bullet\mathbf{v}}{\|\mathbf{u}\|}\frac{\mathbf{u}}{\|\mathbf{u}\|}=\frac{\mathbf{u}\bullet\mathbf{v}}{\|\mathbf{u}\|^2}\mathbf{u}=\frac{\mathbf{u}\bullet\mathbf{v}}{\mathbf{u}\bullet\mathbf{u}}\mathbf{u}$$

$$\mathbf{q}=\mathbf{v}-\mathbf{p}=\mathbf{v}-\frac{\mathbf{u}\bullet\mathbf{v}}{\mathbf{u}\bullet\mathbf{u}}\mathbf{u}$$

图 3.5（b）显示了向量 **v** 分解为 **p** 和 **q**。

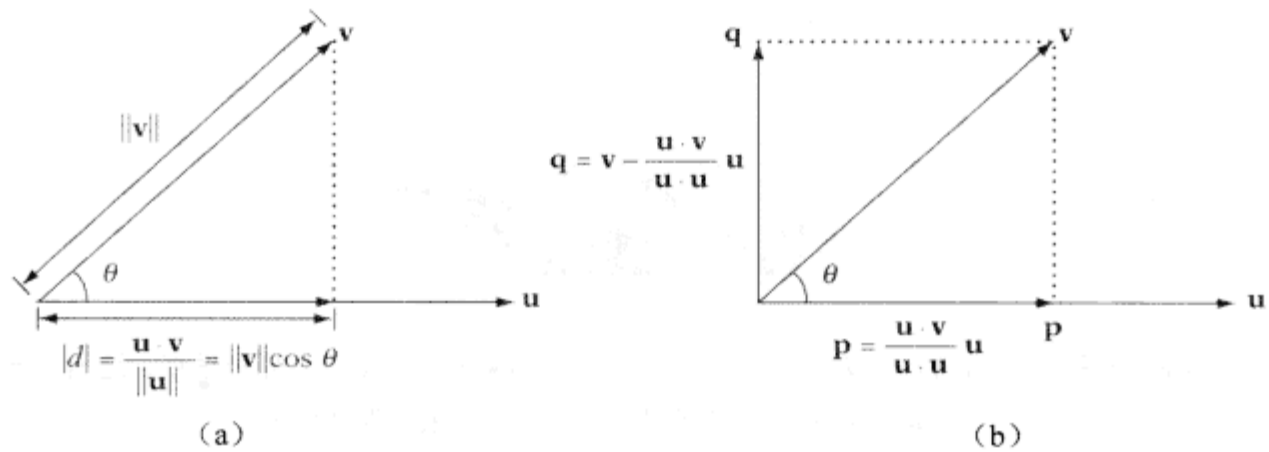


图 3.5　（a）向量 **v** 在向量 **u** 上的距离。（b）向量 **v** 分解为平行于 **u** 的向量 **p** 以及垂直于 **u** 的向量 **q**

需要注意的是，点积由于满足交换律，因此也可以视为：向量 **u** 在向量 **v** 上的投影，并返回 **u** 沿 **v**、$\|\mathbf{u}\|$个单位的距离。

3.3.4　点积的代数恒等式

给定标量 r 和 s，以及向量 **u** 和 **v**，下列恒等式表示点积：

$$\mathbf{u}\cdot\mathbf{v}=u_1v_1+u_2v_2+\cdots+u_nv_n$$

$$\mathbf{u}\cdot\mathbf{v}=\|\mathbf{u}\|\,\|\mathbf{v}\|\cos\theta$$

$$\mathbf{u}\cdot\mathbf{u}=\|\mathbf{u}\|^2$$

$$\mathbf{u}\cdot\mathbf{v}=\mathbf{v}\cdot\mathbf{u}$$

$$\mathbf{u}\cdot(\mathbf{v}\pm\mathbf{w})=\mathbf{u}\cdot\mathbf{v}\pm\mathbf{u}\cdot\mathbf{w}$$

$$r\mathbf{u}\cdot s\mathbf{v}=rs(\mathbf{u}\cdot\mathbf{v})$$

3.3.5　叉积

两个 3D 向量 $\mathbf{u}=(u_1,u_2,u_3)$和 $\mathbf{v}=(v_1,v_2,v_3)$的叉积（或向量积）标记为 $\mathbf{u}\times\mathbf{v}$，按照向量分量的形式定义为：

$$\mathbf{u}\times\mathbf{v}=(u_2v_3-u_3v_2,\ -(u_1v_3-u_3v_1),\ u_1v_2-u_2v_1)$$

计算结果为垂直于 **u** 和 **v** 的一个向量。其大小为 **u**、**v** 的长度与二者夹角的正弦值的乘积。即：

$$\mathbf{u}\times\mathbf{v}=\mathbf{n}\|\mathbf{u}\|\,\|\mathbf{v}\|\sin\theta$$

其中，**n** 为垂直于 **u**、**v** 形成的面的单位向量。对于叉积 $\mathbf{w}=\mathbf{u}\times\mathbf{v}$，**w** 存在两种可能的方向。这里，**w** 的方向具有可选性，从而可以与 **u** 和 **v** 构造出右手坐标系统。利用右手法则有助于形象地记忆右手坐标系统。即如果右手“握住”**w** 向量，手指将从 **u** 向 **v** 环绕，**w** 的方向则与伸出的大拇指方向一致。

$\mathbf{u}\times\mathbf{v}$ 的大小等于 **u**、**v** 构成的平行四边形的面积：对于底边$\|\mathbf{u}\|$和高$\|\mathbf{v}\|$，有$\|\mathbf{u}\|\cdot\|\mathbf{v}\|\sin\theta$，如图 3.6 所示。当两个向量垂直时，其叉积的大小为最大值。

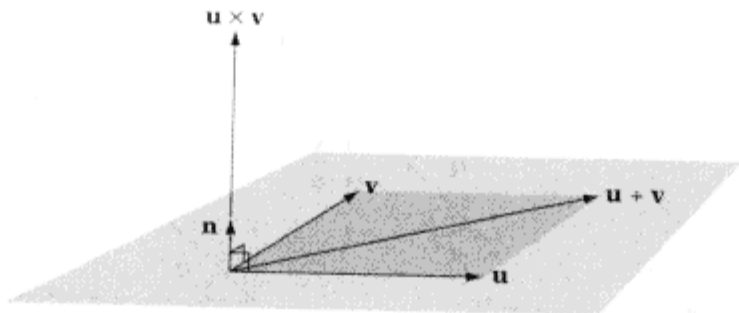


图 3.6　定义两个平面向量 **u** 和 **v**，基于右手法则，其叉积结果 **w**（**w**=**u**×**v**）分别垂直于这两个向量。**w** 的值等于 **u** 和 **v** 构成的平行四边形的面积（采用深灰色表示）

当熟悉了行列式计算后，叉积的表达式可以自然地记为：

$$\mathbf{u}\times\mathbf{v}=\begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ u_1 & u_2 & u_3\\ v_1 & v_2 & v_3\end{vmatrix}=\begin{vmatrix}u_2 & u_3\\ v_2 & v_3\end{vmatrix}\mathbf{i}-\begin{vmatrix}u_1 & u_3\\ v_1 & v_3\end{vmatrix}\mathbf{j}+\begin{vmatrix}u_1 & u_2\\ v_1 & v_2\end{vmatrix}\mathbf{k}$$

其中，**i**=(1,0,0)、**j**=(0,1,0)、**k**=(0,0,1)是平行于坐标轴的单位向量。叉积也可以采用矩阵的形式加以表达，即一个（斜对称）矩阵和一个向量的积：

$$\mathbf{u}\times\mathbf{v}=\begin{bmatrix}0 & -u_3 & u_2\\ u_3 & 0 & -u_1\\ -u_2 & u_1 & 0\end{bmatrix}\begin{bmatrix}v_1\\ v_2\\ v_3\end{bmatrix}$$

需注意的是，叉积实际上只需要 5 次乘法运算，而非如前文所提到的 6 次。如下所示：

$$\mathbf{u}\times\mathbf{v}=(v_2(t_1-u_3)-t_4,u_3v_1-t_3,t_4-u_2(v_1-t_2))$$

其中：

$$t_1=u_1-u_2,\quad t_2=v_2+v_3,\quad t_3=u_1v_3,\quad t_4=t_1t_2-t_3$$

由于该公式增加了操作的运算量：从 9 次（6 次乘法，3 次加法）到 13 次（5 次乘法，8 次加法），因此其计算时的实际操作性能收效将依赖于具体的硬件。

给定一个三角形 *ABC*，两条边叉积的大小将是三角形 *ABC* 面积的 2 倍。对于任意一个非自身相交的四边形 *ABCD*，其两个对角线叉积的大小||**e**||则是四边形 *ABCD* 面积的 2 倍，其中，**e**=(*C*−*A*)×(*D*−*B*)。该特性如图 3.7 所示。

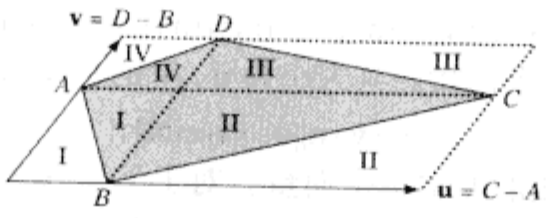


图 3.7　定义四边形 *ABCD*，对角线 *AC* 和 *BD* 的叉积值是四边形 *ABCD* 面积的 2 倍。其中，多边形内的 4 个灰色区域分别等于白色区域，其全部面积和不会大于 *AC* 和 *BD* 构成的平行多边形的面积

在二维空间中，由于第 3 个叉积向量垂直于其他两个向量且不共面，所以不存在直观的叉积等价物。类似的定义是 2D 拟叉积（pseudo cross product），如下：

$$\mathbf{u}^{\perp}\cdot\mathbf{v}$$

其中，$\mathbf{u}^{\perp}=(-u_2,u_1)$为垂直于 **u** 的逆时针向量。$\mathbf{u}^{\perp}$读作“u-perp”。因此，拟叉积有时也称作 perp

点积。2D 拟叉积是一个标量，其值（类似于叉积的大小）对应于向量 **u**、**v** 确定的平行四边形的有符号面积，并且若 **v** 相对于 **u** 逆时针旋转，则为正值；若 **v** 相对于 **u** 顺时针旋转，则为负值；否则为 0。

给定（3D）三角形 ABC，应注意叉积 $\mathbf{d}=(B-A)\times(C-A)$的每一个分量都包含一个 2D 拟叉积，因此，每一个分量都是三角形 ABC 在 yz 轴、zx 轴和 xy 轴上投影面积的 2 倍。对于确定 2D 三角形内部点测试，单独的叉积运算即可满足，同时，若硬件支持上述叉积运算，这一测试过程将更加有效。

3.3.6　叉积的代数恒等式

给定标量 r、s 以及向量 **u**、**v**、**w**，可将叉积恒等式定义为：

$$\mathbf{u}\times\mathbf{v}=-(\mathbf{v}\times\mathbf{u})$$

$$\mathbf{u}\times\mathbf{u}=0$$

$$\mathbf{u}\times 0=0\times\mathbf{u}=0$$

$$\mathbf{u}\cdot(\mathbf{u}\times\mathbf{v})=\mathbf{v}\cdot(\mathbf{u}\times\mathbf{v})=0\text{（}\mathbf{u}\times\mathbf{v}\text{ 垂直于 }\mathbf{u}\text{ 和 }\mathbf{v}\text{）}$$

$$\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w})=(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}$$

$$\mathbf{u}\times(\mathbf{v}\pm\mathbf{w})=\mathbf{u}\times\mathbf{v}\pm\mathbf{u}\times\mathbf{w}$$

$$(\mathbf{u}\pm\mathbf{v})\times\mathbf{w}=\mathbf{u}\times\mathbf{w}\pm\mathbf{v}\times\mathbf{w}$$

$$(\mathbf{u}\times\mathbf{v})\times\mathbf{w}=\mathbf{w}\times(\mathbf{v}\times\mathbf{u})=(\mathbf{u}\cdot\mathbf{w})\mathbf{v}-(\mathbf{v}\cdot\mathbf{w})\mathbf{u}\text{（}\mathbf{u}\text{、}\mathbf{v}\text{ 平面中的一个向量）}$$

$$\mathbf{u}\times(\mathbf{v}\times\mathbf{w})=(\mathbf{w}\times\mathbf{v})\times\mathbf{u}=(\mathbf{u}\cdot\mathbf{w})\mathbf{v}-(\mathbf{u}\cdot\mathbf{v})\mathbf{w}\text{（}\mathbf{v}\text{、}\mathbf{w}\text{ 平面中的一个向量）}$$

$$\|\mathbf{u}\times\mathbf{v}\|=\|\mathbf{u}\|\,\|\mathbf{v}\|\sin\theta$$

$$(\mathbf{u}\times\mathbf{v})\cdot(\mathbf{w}\times\mathbf{x})=(\mathbf{u}\cdot\mathbf{w})(\mathbf{v}\cdot\mathbf{x})-(\mathbf{v}\cdot\mathbf{w})(\mathbf{u}\cdot\mathbf{x})\text{（拉格朗日定理）}$$

$$r\mathbf{u}\times s\mathbf{v}=rs(\mathbf{u}\times\mathbf{v})$$

$$\mathbf{u}\times(\mathbf{v}\times\mathbf{w})+\mathbf{v}\times(\mathbf{w}\times\mathbf{u})+\mathbf{w}\times(\mathbf{u}\times\mathbf{v})=0\text{（Jacobi 定理）}$$

拉格朗日定理对于减少几何测试的计算量是十分有效的，关于这一过程的具体实例可参见第 5 章。

3.3.7　标量三重积

表达式$(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}$ 经常出现在计算中，其自身名称为“标量三重积”（也称作三重标量积或 box 积）。从几何角度来看，标量三重积的值对应 3 个非相关向量 **u**、**v**、**w** 形成的平行六面体的（有符号）体积。相应地，其值是 **u**、**v**、**w** 构成的四面体的体积的 6 倍。标量三重积和平行六面体之间的关系如图 3.8 所示。

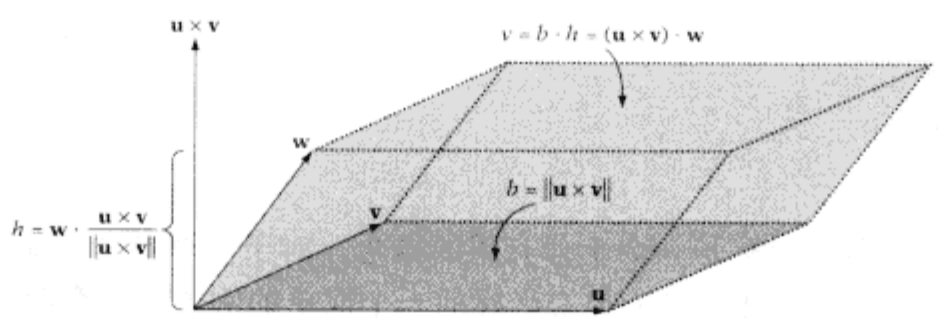


图 3.8　标量三重积$(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}$ 等价于向量 **u**、**v**、**w** 形成的平行六面体的（有符号）体积空间

在三重积运算中，叉积和点积可以互换，且不影响计算结果：

$$(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}=\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w})$$

同时，3 个参数之间可以轮流互换，且标量三重积的值保持不变：

$$(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}=(\mathbf{v}\times\mathbf{w})\cdot\mathbf{u}=(\mathbf{w}\times\mathbf{u})\cdot\mathbf{v}$$

基于上述恒等式，特定的符号[**u v w**]常用于表示三重积，且定义为：

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=(\mathbf{u}\times\mathbf{v})\cdot\mathbf{w}=\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w})$$

该符号抽象化了点积和叉积的运算顺序，且便于记忆正确的运算符号。如果向量从左至右读作 **uvw**（始于 **u** 并允许环绕），则积恒等式为正值，否则为负值：

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=[\mathbf{v}\ \mathbf{w}\ \mathbf{u}]=[\mathbf{w}\ \mathbf{u}\ \mathbf{v}]=-[\mathbf{u}\ \mathbf{w}\ \mathbf{v}]=-[\mathbf{v}\ \mathbf{u}\ \mathbf{w}]=-[\mathbf{w}\ \mathbf{v}\ \mathbf{u}]$$

标量三重积也可以表示为 3×3 矩阵的行列式，其中行（或列）为向量的分量：

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=\begin{vmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix}=u_1\begin{vmatrix} v_2 & v_3 \\ w_2 & w_3 \end{vmatrix}-u_2\begin{vmatrix} v_1 & v_3 \\ w_1 & w_3 \end{vmatrix}+u_3\begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix}=\mathbf{u}\cdot(\mathbf{v}\times\mathbf{w})$$

向量 **u**、**v**、**w** 共面，当且仅当$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=0$。

3.3.8 标量三重积的代数恒等式

给定向量 **u**、**v**、**w**、**x**、**y**、**z**，下列恒等式定义了标量三重积的计算方法：

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=[\mathbf{v}\ \mathbf{w}\ \mathbf{u}]=[\mathbf{w}\ \mathbf{u}\ \mathbf{v}]=-[\mathbf{u}\ \mathbf{w}\ \mathbf{v}]=-[\mathbf{v}\ \mathbf{u}\ \mathbf{w}]=-[\mathbf{w}\ \mathbf{v}\ \mathbf{u}]$$

$$[\mathbf{u}\ \mathbf{u}\ \mathbf{v}]=[\mathbf{v}\ \mathbf{u}\ \mathbf{v}]=0$$

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]^2=[(\mathbf{u}\times\mathbf{v})\ (\mathbf{v}\times\mathbf{w})\ (\mathbf{w}\times\mathbf{u})]$$

$$\mathbf{u}[\mathbf{v}\ \mathbf{w}\ \mathbf{x}]-\mathbf{v}[\mathbf{w}\ \mathbf{x}\ \mathbf{u}]+\mathbf{w}[\mathbf{x}\ \mathbf{u}\ \mathbf{v}]-\mathbf{x}[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]=0$$

$$(\mathbf{u}\times\mathbf{v})\times(\mathbf{w}\times\mathbf{x})=\mathbf{v}[\mathbf{u}\ \mathbf{w}\ \mathbf{x}]-\mathbf{u}[\mathbf{v}\ \mathbf{w}\ \mathbf{x}]$$

$$[(\mathbf{u}\times\mathbf{v})\ (\mathbf{w}\times\mathbf{x})\ (\mathbf{y}\times\mathbf{z})]=[\mathbf{v}\ \mathbf{y}\ \mathbf{z}][\mathbf{u}\ \mathbf{w}\ \mathbf{x}]-[\mathbf{u}\ \mathbf{y}\ \mathbf{z}][\mathbf{v}\ \mathbf{w}\ \mathbf{x}]$$

$$[(\mathbf{u}+\mathbf{v})\ (\mathbf{v}+\mathbf{w})\ (\mathbf{w}+\mathbf{u})]=2\ [\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$$

$$[\mathbf{u}\ \mathbf{v}\ \mathbf{w}][\mathbf{x}\ \mathbf{y}\ \mathbf{z}]=\begin{vmatrix} \mathbf{u}\cdot\mathbf{x} & \mathbf{u}\cdot\mathbf{y} & \mathbf{u}\cdot\mathbf{z} \\ \mathbf{v}\cdot\mathbf{x} & \mathbf{v}\cdot\mathbf{y} & \mathbf{v}\cdot\mathbf{z} \\ \mathbf{w}\cdot\mathbf{x} & \mathbf{w}\cdot\mathbf{y} & \mathbf{w}\cdot\mathbf{z} \end{vmatrix}$$

$$\begin{aligned}[(\mathbf{u}-\mathbf{x})\ (\mathbf{v}-\mathbf{x})\ (\mathbf{w}-\mathbf{x})]&=[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]-[\mathbf{u}\ \mathbf{v}\ \mathbf{x}]-[\mathbf{u}\ \mathbf{x}\ \mathbf{w}]-[\mathbf{x}\ \mathbf{v}\ \mathbf{w}]\\&=[(\mathbf{u}-\mathbf{x})\mathbf{v}\ \mathbf{w}]-[(\mathbf{v}-\mathbf{w})\mathbf{x}\ \mathbf{u}]\end{aligned}$$

3.4 质心坐标

在多种相交测试中，质心坐标是一个十分有用的概念。对于一个参考点集合的加权组合构成的空间，质心坐标可以对其实现参数化。作为质心坐标的一个简单的例子，考查两个顶点 A 和 B，其间有点 P 且 $P=A+t(B-A)=(1-t)A+tB$，或简写成 $P=uA+vB$，其中，$u+v=1$。当且仅当 $0\leqslant u\leqslant 1$、$0\leqslant v\leqslant 1$ 时，

顶点 P 位于线段 AB 之上。对于最后一种形式，(u,v)称作顶点 P 在线段 AB 的质心坐标。顶点 A 的质心坐标为(1,0)，顶点 B 的质心坐标为(0,1)。

u、v 作为权值置于线段 AB 的两个端点 A 和 B 之间，则按比例 $v:u$ 分割线段的顶点 Q 称作质心或重心（线段所受重力的中心点或其平衡位置）。质心坐标的典型应用之一是参数化三角形（或三角形的平面）。考查 3 个非共线顶点 A、B、C 构造的三角形 ABC。面上一点 P 可以单独表示为 $P=uA+vB+wC$，其中 u、v、w 为常数，且 $u+v+w=1$。

三元组(u,v,w)对应着顶点的质心坐标。对于三角形 ABC，顶点 A、B、C 的质心坐标分别为(1,0,0)、(0,1,0)、(0,0,1)。一般地，当且仅当 $0\leqslant u$、v、$w\leqslant 1$，或当且仅当 $0\leqslant v\leqslant 1$、$0\leqslant w\leqslant 1$、$v+w\leqslant 1$ 时，具有质心坐标(u,v,w)的顶点将位于三角形的内部（或边上）。质心坐标实际上按照 $P=uA+vB+wC$ 的方式参数化了平面，且是 $P=A+v(B-A)+w(C-A)$的另一种表述形式：

$$P=A+v(B-A)+w(C-A)=(1-v-w)A+vB+wC$$

其中，两个不相关的方向向量 AB 和 AC 以 A 作为原点，构成了一个坐标系统，并使得面中的顶点依据 v 和 w 加以参数化。很明显，(u,v,w)形式的质心坐标是一种冗余的表达形式，其中，第 3 个分量可以依据前两个分量加以表达，其存在的原因只是对称性的需要。

为了求解质心坐标，表达式 $P=A+v(B-A)+w(C-A)$或 $v(B-A)+w(C-A)=P-A$ 可以写作 $v\mathbf{v}_0+w\mathbf{v}_1=\mathbf{v}_2$，其中 $\mathbf{v}_0=B-A$，$\mathbf{v}_1=C-A$，$\mathbf{v}_2=P-A$。则通过两边乘以 $\mathbf{v}_0$、$\mathbf{v}_1$ 构造一个 2×2 线性方程组：

$$(v\mathbf{v}_0+w\mathbf{v}_1)\cdot\mathbf{v}_0=\mathbf{v}_2\cdot\mathbf{v}_0$$
$$(v\mathbf{v}_0+w\mathbf{v}_1)\cdot\mathbf{v}_1=\mathbf{v}_2\cdot\mathbf{v}_1$$

由于点积是一个线性操作符，上述表达式等价于：

$$v(\mathbf{v}_0\cdot\mathbf{v}_0)+w(\mathbf{v}_1\cdot\mathbf{v}_0)=\mathbf{v}_2\cdot\mathbf{v}_0$$
$$v(\mathbf{v}_0\cdot\mathbf{v}_1)+w(\mathbf{v}_1\cdot\mathbf{v}_1)=\mathbf{v}_2\cdot\mathbf{v}_1$$

该方程组可以简单地应用克莱姆法则，下列代码将使用该方法计算质心坐标：

```
// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
void Barycentric(Point a, Point b, Point c, Point p, float &u, float &v, float &w)
{
    Vector v0 = b - a, v1 = c - a, v2 = p - a;
    float d00 = Dot(v0, v0);
    float d01 = Dot(v0, v1);
    float d11 = Dot(v1, v1);
    float d20 = Dot(v2, v0);
    float d21 = Dot(v2, v1);
    float denom = d00 * d11 - d01 * d01;
    v = (d11 * d20 - d01 * d21) / denom;
    w = (d00 * d21 - d01 * d20) / denom;
    u = 1.0f - v - w;
}
```

如果多个顶点只对同一个三角形执行测试，则 **d00**、**d01**、**d11**、**denom** 只需计算一次，因为对于一个给定的三角形，上述 4 项的值是不变的。

对于任一维度的单形体（simplex，参见 3.8 节），都可以计算一点的质心坐标。例如，给定由顶

点 A、B、C、D 构造的四面体，质心坐标(u,v,w,x)确定了 3D 空间中的一点 P，$P=uA+vB+wC+xD$ 且 $u+v+w+x=1$。当且仅当 $0\leqslant u$、v、w、$x\leqslant 1$，点 P 位于四面体内部。

给定点 $A=(a_x,a_y,a_z)$，$B=(b_x,b_y,b_z)$，$C=(c_x,c_y,c_z)$，$D=(d_x,d_y,d_z)$，且 $P=(p_x,p_y,p_z)$，质心坐标可以通过建立下列线性方程组进行求解：

$$a_xu+b_xv+c_xw+d_xx=p_x$$
$$a_yu+b_yv+c_yw+d_yx=p_y$$
$$a_zu+b_zv+c_zw+d_zx=p_z$$
$$u+v+w+x=1$$

将 $P=uA+vB+wC+xD$ 等式两边分别减去 A，得到：

$$P-A=v(B-A)+w(C-A)+x(D-A)$$

通过求解下列等式，可以得到 4 个质心坐标中的 3 个解：

$$(b_x-a_x)v+(c_x-a_x)w+(d_x-a_x)x=p_x-a_x$$
$$(b_y-a_y)v+(c_y-a_y)w+(d_y-a_y)x=p_y-a_y$$
$$(b_z-a_z)v+(c_z-a_z)w+(d_z-a_z)x=p_z-a_z$$

且第 4 个分量可以利用 $u=1-v-w-x$ 得到。上述方程组可以利用克莱姆法则或高斯消元法方便地进行求解。例如，在前一个方程中，坐标可以通过克莱姆法则以及行列式之比求得：

$$u=d_{PBCD}/d_{ABCD}$$
$$v=d_{APCD}/d_{ABCD}$$
$$w=d_{ABPD}/d_{ABCD}$$
$$x=d_{ABCP}/d_{ABCD}=1-u-v-w$$

$$d_{PBCD}=\begin{vmatrix} p_x & b_x & c_x & d_x \\ p_y & b_y & c_y & d_y \\ p_z & b_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix},\quad d_{APCD}=\begin{vmatrix} a_x & p_x & c_x & d_x \\ a_y & p_y & c_y & d_y \\ a_z & p_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix},\quad d_{ABPD}=\begin{vmatrix} a_x & b_x & p_x & d_x \\ a_y & b_y & p_y & d_y \\ a_z & b_z & p_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix},$$

$$d_{ABCP}=\begin{vmatrix} a_x & b_x & c_x & p_x \\ a_y & b_y & c_y & p_y \\ a_z & b_z & c_z & p_z \\ 1 & 1 & 1 & 1 \end{vmatrix},\quad d_{ABCD}=\begin{vmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix}$$

行列式对应着四面体 $PBCD$、$APCD$、$ABPD$、$ABCP$、$ABCD$ 的有符号体积（严格地讲，是每一个有符号体积的 1/6）。行列式的比值将简化为：顶点到其对应平面的相对高度。

现在再来考查三角形，正如四面体的质心坐标可以表示为体空间之比，三角形的质心坐标也可以看作是面积之比。特别地，点 P 的质心坐标可以看作是三角形 PBC、PCA、PAB 与整个三角形 ABC 之比。因此，质心坐标也常称作面积坐标。当采用有符号的三角形面积时，上述表达式对于判断位于三角形之外的点也是有效的。鉴于此，质心坐标(u,v,w)记作：

```
u = SignedArea(PBC)/SignedArea(ABC)
v = SignedArea(PCA)/SignedArea(ABC)
w = SignedArea(PAB)/SignedArea(ABC) = 1 − u −v
```

因为已消除常量因子，所以任意三角形比例函数都可以用于计算上述比值。特别地，这里可以采用三角形两边叉积的量值。正确的计算符号由叉积和三角形 *ABC* 面法线的点积值加以确定。例如，三角形 *ABC* 的有符号面积可以计算为：

SignedArea(PBC) = Dot(Cross(B-P, C-P), Normalize(Cross(B-A, C-A))).

由于三角形的面积可以记作底边×高/2，又前述各三角形比值项中都包含相同的底边，因此表达式可以简化为高度之比。因此，另一种看待质心坐标的方式是：分量 *u*、*v*、*w*——点 *P* 对应于边 *BC*、*AC*、*AB* 的规范化高度——与顶点至对应边上的高度成某种比例。由于三角形是 3 个 2D 空间的交集，每一个空间都是由基于三角形某一边的且高度为 0、1 的两条平行线（或平行面）间的无限空间加以定义，这也是当一点位于三角形内部时，0≤*u*、*v*、*w*≤1 的原因，如图 3.9 所示。

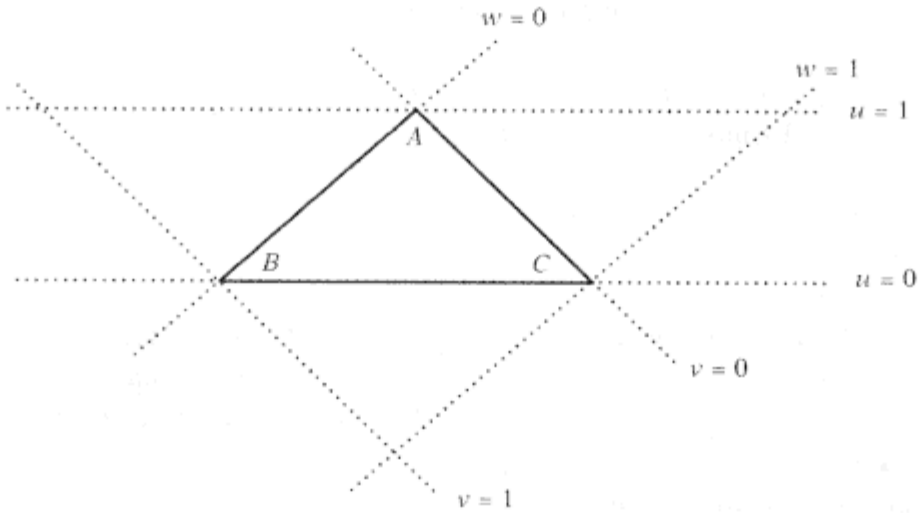


图 3.9　三角形 *ABC* 采用“高度直线”加以标注，其中，分别定义 *u*=0，*u*=1，*v*=0，*v*=1，*w*=0，*w*=1

与三角形各边重合的直线可以看作将三角形平面分割为 7 个质心域，每一个域的符号将依据质心坐标分量的正负值计算：3 个边域、3 个顶点域以及三角形内部域（如图 3.10 所示）。上述关于域的划分，对于网格遍历算法以及包含算法都是十分有意义的。

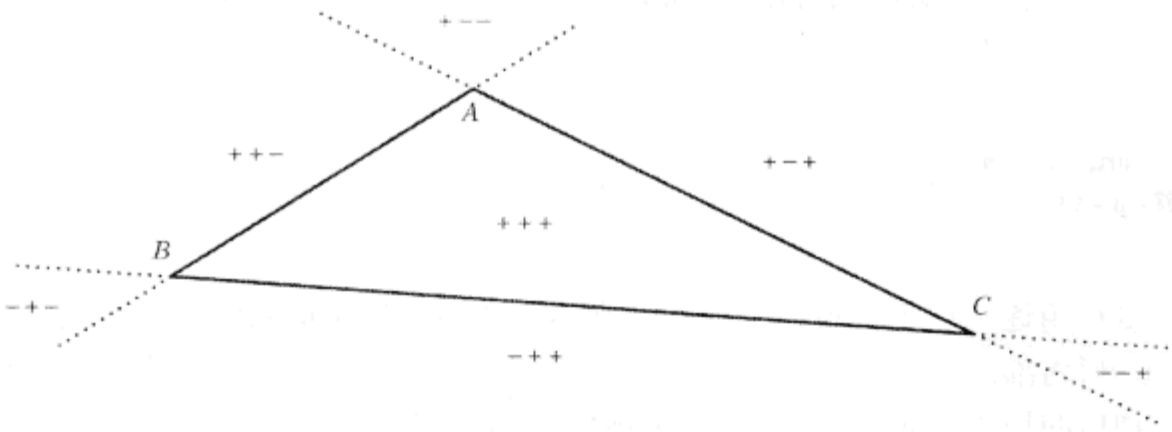


图 3.10　基于坐标分量的符号，质心坐标将三角形 *ABC* 的平面分割为 7 个区域

关于质心坐标的重要特征之一是投影不变性，这一特征与前述的坐标计算方法相比更具高效性：即无须使用顶点的 3D 坐标来计算面积，可以将顶点投影至 *xy*、*xz* 或 *yz* 平面从而简化计算过程。

为了防止退化现象，投影总是作用于最大投影面，三角形法线分量的最大绝对值给出了投影时的

“舍弃”分量。

```
inline float TriArea2D(float x1, float y1, float x2, float y2, float x3, float y3)
{
    return (x1-x2)*(y2-y3) - (x2-x3)*(y1-y2);
}

// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
void Barycentric(Point a, Point b, Point c, Point p, float &u, float &v, float &w)
{
    // Unnormalized triangle normal
    Vector m = Cross(b - a, c - a);
    // Nominators and one-over-denominator for u and v ratios
    float nu, nv, ood;
    // Absolute components for determining projection plane
    float x = Abs(m.x), y = Abs(m.y), z = Abs(m.z);

    // Compute areas in plane of largest projection
    if (x >= y && x >= z) {
        // x is largest, project to the yz plane
        *nu = TriArea2D(p.y, p.z, b.y, b.z, c.y, c.z); // Area of PBC in yz plane
        *nv = TriArea2D(p.y, p.z, c.y, c.z, a.y, a.z); // Area of PCA in yz plane
        ood = 1.0f / m.x;                              // 1/(2*area of ABC in yz plane)
    }else if (y >= x && y >= z) {
        // y is largest, project to the xz plane
        nu = TriArea2D(p.x, p.z, b.x, b.z, c.x, c.z);
        nv = TriArea2D(p.x, p.z, c.x, c.z, a.x, a.z);
        ood = 1.0f / -m.y;
    }else {
        // z is largest, project to the xy plane
        nu = TriArea2D(p.x, p.y, b.x, b.y, c.x, c.y);
        nv = TriArea2D(p.x, p.y, c.x, c.y, a.x, a.y);
        ood = 1.0f / m.z;
    }
    u = nu * ood;
    v = nv * ood;
    w = 1.0f - u - v;
}
```

质心坐标有多种用途，由于其投影不变性，常用于在不同的坐标系统间实行点映射；也可用于测试点是否位于三角形内部；另外，给定一个顶点光照三角形，可以获取三角形内部一点相应的 RGB 值，从而调整该位置物体的环境光颜色；对于三角形剪裁，可以插值计算任何属性值，包括颜色值（Gouraud 着色）、法线（Phong 着色）和纹理坐标（纹理映射）。如下代码为利用质心坐标测试一点 P 是否位于三角形 ABC 内部：

```
// Test if point p is contained in triangle (a, b, c)
int TestPointTriangle(Point p, Point a, Point b, Point c)
{
```

```
    float u, v, w;
    Barycentric(a, b, c, p, u, v, w);
    return v >= 0.0f && w >= 0.0f && (v + w) <= 1.0f;
}
```

对于不规则的 n 边凸多边形，质心坐标的通用计算公式可以参见[Meyer02]。当 n=3 时，此公式将还原为质心坐标的传统计算公式，参见[Floater04]。

对于顶点 P_0，P_1，…，P_n，若点 P 的质心坐标满足 a_0，a_1，…，$a_n \geqslant 0$，则称 P 是 P_0，P_1，…，P_n 的凸组合。

3.5　直线、光线和线段

直线可以定义为一个点集，即任意不重合的两点 A、B 间的线性组合：

$$L(t)=(1-t)A+t\,B$$

这里，t 的范围是全体实数，$-\infty<t<\infty$。连接 A、B 的线段只是该直线上的一部分，可以通过 $0\leqslant t\leqslant 1$ 加以限定。如果两个端点 A、B 以某种顺序给出，则该线段是有方向的。光线可以类似地定义为“半无穷（half-infinite）”直线，且有 $t\geqslant 0$。直线、光线、线段之间的区别如图 3.11 所示。

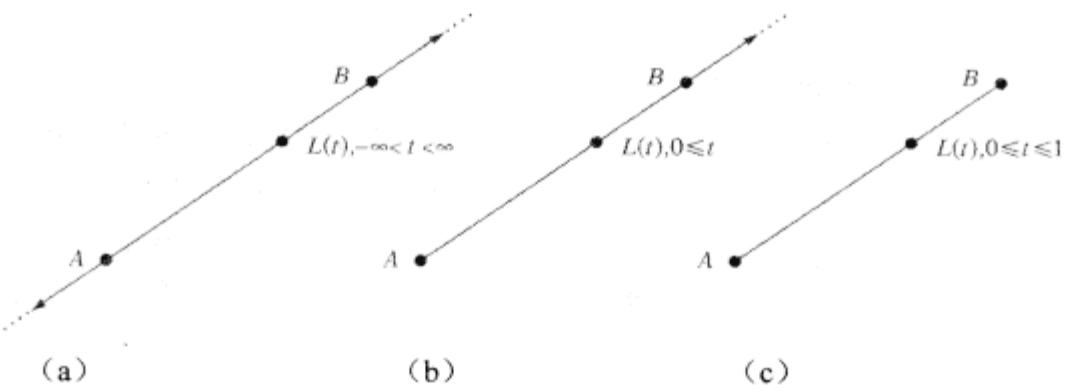


图 3.11　（a）直线。（b）光线。（c）线段

整理参数直线方程中的各项，得到：

$$L(t)=A+t\mathbf{v} \qquad (\mathbf{v}=B-A)$$

特别地，光线常以这种方式加以定义。上述两种形式都可以用作参数化的直线方程。在 3D 空间中，直线可以隐式地定义为满足点集 X 的方程：

$$\|(X-A)\times\mathbf{v}\|=0$$

其中，A 是 L 上的一点；$\mathbf{v}$ 则是平行于 L 的向量。该方程表明：当且仅当向量 $X-A$ 平行于向量 $\mathbf{v}$ 时，将得到 0 向量，则点 X 位于 L 之上。实际上，当 $\mathbf{v}$ 为单位向量时，点 P 距 L 的距离为$\|(P-A)\times\mathbf{v}\|$。该方程常用于计算点的共线性：当全部点位于同一条直线上时，则称其共线。考查 3 个顶点 A、B、C，当且仅当三角形的面积为 0 时，则三点共线。令 $\mathbf{m}=(B-A)\times(C-A)$，可以通过考查$\|\mathbf{m}\|=0$ 来测试共线性；或者为了避免计算平方根，也可以考查 $\mathbf{m}\cdot\mathbf{m}=0$ 来测试共线性。还可以采用下列计算方法：(m_x,m_y,m_z) 为 $\mathbf{m}$ 的 3 个分量，当且仅当$|m_x|+|m_y|+|m_z|=0$ 时，三点共线。

3.6 平面和半空间

3D 空间中的平面可以看作是沿各方向无限延伸的表面，并可以通过多种方式加以描述，例如：

- ☑ 3 点不共线（形成一个平面上的三角形）。
- ☑ 平面上的一点及其法线。
- ☑ 平面的法线和距离原点的距离。

在第一种情况下，平面 P 可以利用顶点 A、B、C 给出其参数表达：

$$P(u,v)=A+u(B-A)+v(C-A)$$

对于其他两种定义方式，平面的法线须是一个非 0 向量，并且是垂直于任一平面内的向量。对于一个给定的平面，存在两种法线的选择方式，二者分别指向相反的方向。当观察一个由三角形 ABC 确定的平面时，且三角形的 3 个顶点呈逆时针旋转，一般定义该平面法线指向观察者。此时，平面法线按下列叉积形式计算：$\mathbf{n}=(B-A)\times(C-A)$。若顶点位于平面法线指向的空间，则称该顶点位于平面的“正面”；否则，称顶点位于平面的“背面”。

给定平面的法线 $\mathbf{n}$ 及面上一点 P，平面内的点集 X 可以归结为：向量 $X-P$ 垂直于平面的法线 $\mathbf{n}$，即两个向量的点积值为 0。这种垂直特征定义了平面的隐式方程，即“点-法”式平面方程：

$$\mathbf{n}\cdot(X-P)=0$$

作为线性操作符，点积操作满足分配律，则上述方程可记作 $\mathbf{n}\cdot X=d$，其中 $d=\mathbf{n}\cdot P$，即平面的“常数-法线”式方程。若 $\mathbf{n}$ 为单位向量，则$|d|$等于原点距该平面的距离；若 $\mathbf{n}$ 不是单位向量，$|d|$仍为一个距离值，但为“向量 $\mathbf{n}$ 的长度”个单位量。如果不采用绝对值，d 值将是一个有符号距离。

“常数-法线”式平面方程常记作分量形式 $ax+by+cz-d=0$，其中 $\mathbf{n}=(a,b,c)$，$X=(x,y,z)$。这里，$ax+by+cz-d=0$ 更倾向于采取常规的形式 $ax+by+cz+d=0$，即消除了一个“多余的”负号（如在面相交计算中）。

当预构造平面时，令面法线为单位向量将是十分有用的。法线的规范化可以将 $\mathbf{n}$ 除以$\|\mathbf{n}\|$得到 $\sqrt{a^2+b^2+c^2}$ 来进行。单位面法线向量简化了很多平面计算，因此其平面方程称作归一化的平面方程。当给定一点并采用归一化的平面方程时，结果值为该点距离平面的有符号距离（若为负值，则点位于平面的“背面”，否则该点位于平面的“正面”）。

3 个非共线顶点的平面计算方式如下所示：

```
struct Plane {
*Vector n;        // Plane normal. Points x on the plane satisfy Dot(n,x) = d
*float d;         // d = dot(n,p) for a given point p on the plane
};

// Given three noncollinear points (ordered ccw), compute plane equation
Plane ComputePlane(Point a, Point b, Point c)
{
    Plane p;
    p.n = Normalize(Cross(b - a, c - a));
    p.d = Dot(p.n, a);
```

```
    return p;
}
```

参数化的平面方程可定义为：

$$P(s,t)=A+s\mathbf{u}+t\mathbf{v}$$

其中，**u** 和 **v** 为平面内两个不相关的向量，A 为平面内一点。

若两个平面彼此不平行，则相交于一条直线处。两个相交平面之间的夹角称作二面角。

任意维度内的平面称作超平面：其平面与其所处的空间相比，将减少一个维度。在 2D 空间中，超平面对应一条直线；在 3D 空间中，超平面对应一个平面。超平面将空间划分为两组位于平面两侧的无穷点集，这两组集合称作半空间（如图 3.12 所示）。若半空间包含分割面上的点，则称该半空间为“闭半空间”（否则，称作“开半空间”）。正半空间位于面法线所指向的一侧，而负半空间则位于该平面的另一侧。一个 2D 半空间也称作半平面。

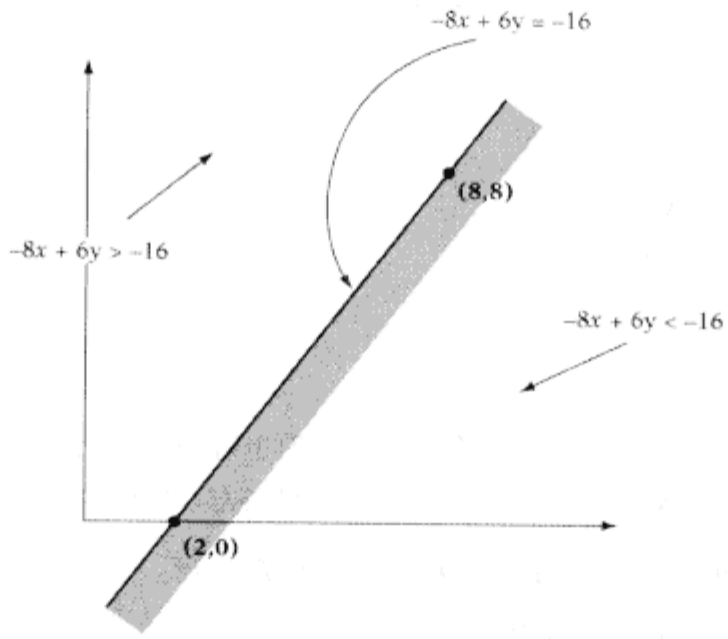


图 3.12　2D 超平面$-8x+6y=-16$（直线）将平面分割为两个半空间

3.7　多边形

多边形是具有 n 条边的封闭图元，且定义为有序的平面顶点集（顶点数量至少为 3 个），因此，每一个顶点依次连接到下一个顶点（最后一个顶点连接着第一个顶点），从而形成一条线段。对于一个包含 n 个顶点的集合，最终的多边形称作 n-边多边形或 n-gon。构成多边形边界的线段称作多边形的边，各点称作多边形的顶点。如果两个顶点由一条边连接，则称这两个顶点是相邻的。图 3.13 显示了多边形的各个元素。

若多边形的连续边之间除了连接顶点外没有其他公共点，则称该多边形为简单多边形。一个简单多边形将平面分为两个独立的部分：多边形内部（多边形覆盖的全部边界区域）和多边形外部（多边形外部的非边界区域）。通常，多边形的含义包括：多边形的边界和多边形的内部区域。多边形对角线定义为：连接两个多边形顶点的一条线段，且该线段全部位于多边形内部。如果多边形的内角（该

角度由连接一个顶点的两条边构成，且位于多边形内部）小于或等于 180°，则称相应顶点为凸顶点（如图 3.14（a）所示）。若该角度大于 180°，则相应顶点称作凹顶点（如图 3.14（b）所示）。

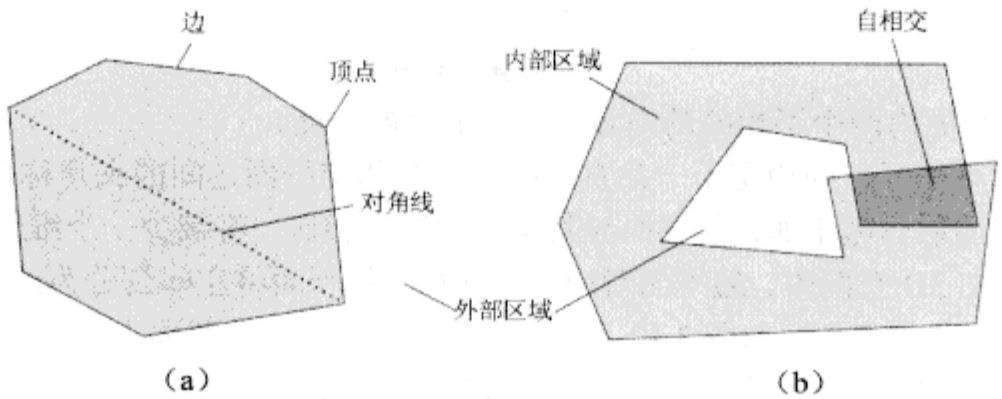


图 3.13　多边形元素。（a）简单多边形。（b）自相交的非简单多边形

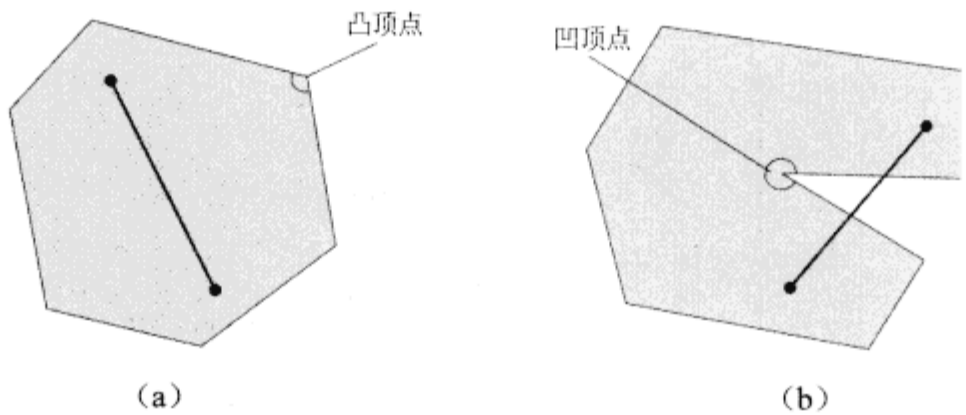


图 3.14　（a）针对凸多边形，连接任意两点的线段完全位于该多边形的内部。
（b）若连接两点的线段部分位于多边形的外部，则该多边形为凹多边形

如果多边形 P 任意两点间的所有线段皆位于多边形 P 内部，则称多边形 P 为凸多边形；否则，称为凹多边形。包含一个或多个凹顶点的多边形一定是一个凹多边形；但包含凸顶点的多边形不总是凸多边形（参见 3.7.1 节）。三角形是唯一可保证凸体性质的 n 边多边形。凸多边形可以看作是凸顶点集合的一个子集。一个凸顶点子集 S，S 内任意两点间的线段仍位于集合 S 中。当给定一个点集 S 时，S 的凸包是完全包含 S 的最小凸顶点集，记作 $CH(S)$，如图 3.15 所示。$CH(S)$也可描述为包含 S 的全部凸顶点集的交集。

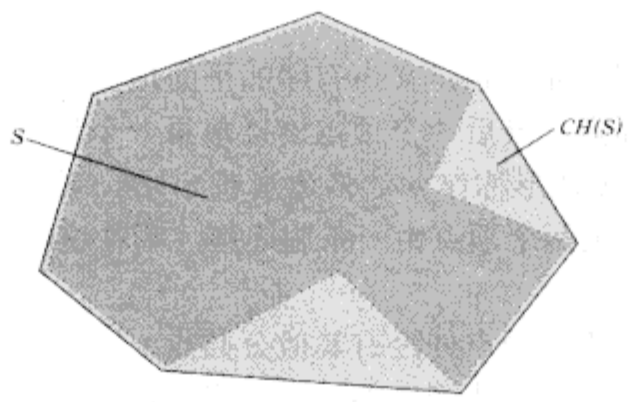


图 3.15　凹多边形的凸壳（这里，可将凸壳形容为箍在多边形对象上的橡皮圈）

与凸包相关的是仿射包，记作 $AH(S)$。仿射包是包含 S 全部顶点的、最小维度的超平面。即如果 S

只包含一个顶点，则 $AH(S)$也将是一点；如果 $AH(S)$包含两个点，则 $AH(S)$将是通过这两点的一条直线；如果 S 包含 3 个不共线点，则 $AH(S)$将是由这 3 个点确定的平面；如果 S 包含 4 个（或更多）非共面点，则 $AH(S)$将是全部 $\mathbb{R}^3$。

除了显式的多边形顶点表达方式之外，凸多边形还可以表述为有限数量的半空间的交集，这一定义方式简化了点包含测试。对于这种隐式的多边形表达方式，如果点位于所有的半空间之内，则该点也位于对应的多边形之内。图 3.16 显示了 3 个半空间交集形成的三角形。因此，另一种点集的凸性可以定义为：当且仅当点集 S 等于包含 S 的全部半空间的交集时，该点集为凸顶点集。对于多边形（以及多面体），这是一种有效的定义方式，且可以直接用于实现相应的凸测试。

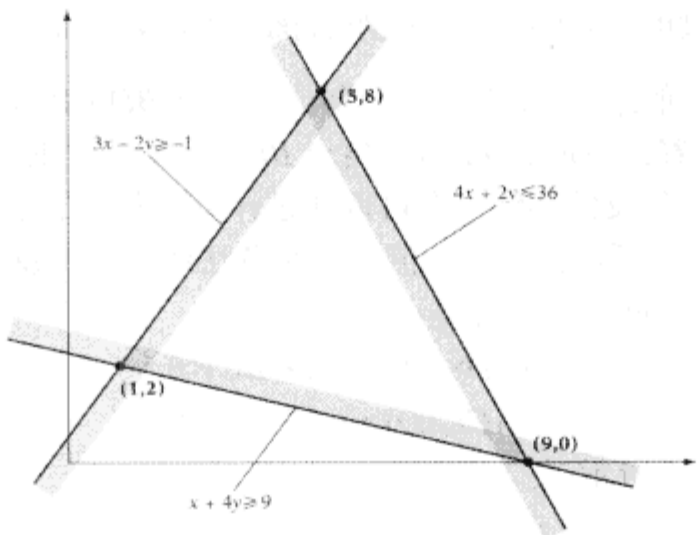


图 3.16　凸多边形可描述为一组（封闭）半空间的交集。这里，三角形(1,2),(9,0),(5,8)定义为半空间 $x+4y\geqslant 9$,$4x+2y\leqslant 36$ 和 $3x-2y\geqslant -1$ 的交集

两个或多个多边形可以通过公共边连接为多边形网格。在多边形网格中，顶点的“度”解释为：顶点所连接的边数量。若当全部多边形实现连接时，每一条边都是确定的两个多边形的公共边，则称构成的网格为封闭网格。

3.7.1　多边形凸性测试

在碰撞检测系统中，当采用凸多边形而非凹多边形时，往往会加快多边形相关操作和相交测试的速度。从这一角度来看，作为一种确定的凸体多边形类型，三角形是一个不错的选择。但是，对一个包含 n 边的凸多边形而非同一区域的多个三角形执行测试，往往会更加高效。为了确保碰撞几何数据中不包含凹多边形——其无法实现针对凸面编写的快速测试算法——无论是在建模阶段还是在运行阶段（也许还包括 debug 建立阶段），所有面都应执行凸面测试。

通常，除了三角形，四边形也是一种支持型图元。因此，可以专门采用应用于四边形的、更加简单的凸面测试方法，而非任意多边形的通用凸体测试。这里，设定四边形 $ABCD$ 的全部顶点共面，当且仅当两对角线完全位于四边形内部，则该四边形为凸面（如图 3.17（a）～图 3.17（c）所示）。相对于对角线，该测试对应于线段 AC 和 BD 的相交测试。如果二者相交，则该四边形为凸面；否则，该四边形为凹面或自相交面。如果两线段平行或重叠，四边形将产生退化现象（退化为一条直线），如图 3.17（d）所示。对于存在三点共线的凸面四边形，如果一条线段在另一条内部（而非延长线的一端

点）交迭，则认为两线段相交。

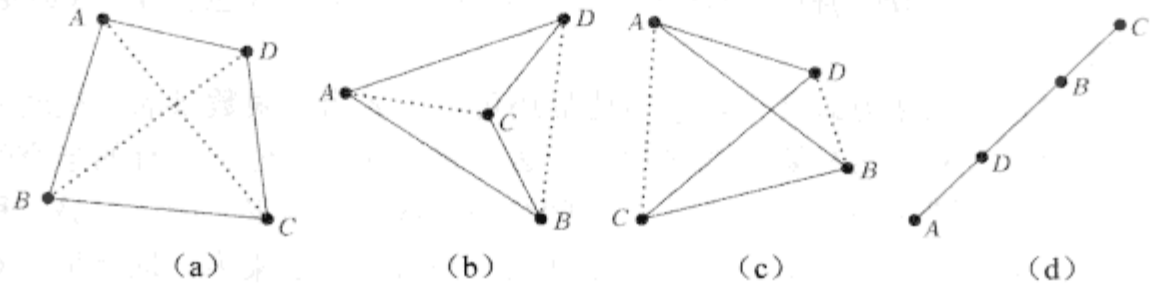


图 3.17　不同种类的四边形。（a）凸四边形。（b）凹四边形（镖状几何体）。（c）自相交多边形。（d）退化的多边形。其中，虚线表示多边形的对角线。当且仅当对角线相交时，四边形为凸形

可以看出，线段的相交等价于：顶点 A、C 分别位于直线 BD 的两侧；顶点 B、D 也位于直线 AC 的两侧。因此，该测试也意味着三角形 BDA 与三角形 BDC 有着相反的环绕方向；三角形 ACD 与三角形 ACB 有着相反的环绕方向。相反的环绕方向可以通过计算三角形的法线（利用叉积）并在两个法线之间执行点积值的符号测试：如果点积值为负值，则两条法线指向相反的方向，且三角形之间的环绕方向相反。综上所述，四边形为凸面的条件是：

$$(BD \times BA) \cdot (BD \times BC) < 0$$

$$(AC \times AD) \cdot (AC \times AB) < 0$$

一种较直接的实现方式如下所示：

```
// Test if quadrilateral (a, b, c, d) is convex
int IsConvexQuad(Point a, Point b, Point c, Point d)
{
    // Quad is nonconvex if Dot(Cross(bd, ba), Cross(bd, bc)) >=0
    Vector bda = Cross(d - b, a - b);
    Vector bdc = Cross(d - b, c - b);
    if (Dot(bda, bdc) >= 0.0f) return 0;
    // Quad is now convex iff Dot(Cross(ac, ad), Cross(ac, ab))<0
    Vector acd = Cross(c - a, d - a);
    Vector acb = Cross(c - a, b - a);
    return Dot(acd, acb) < 0.0f;
}
```

两条线段的相交测试将在 5.1.9.1 节加以讨论。

并非只是四边形，对于一般的 n 边多边形，一种简单的方法是：对于多边形的每一条边，测试所有其他顶点是否（严格地）位于该边的同一侧，如果计算结果为真，则该多边形为凸面，否则为凹面。另外，还需要检测重合的顶点，以使测试更加健壮。该算法虽然易于实现，但对于大型多边形，全部顶点的算法复杂度为 $O(n^2)$，其代价比较昂贵。碰撞检测系统中，过于复杂的多边形并不常见，因此复杂度为 $O(n^2)$往往意味着问题的出现。当然，快速的测试算法也是易于实现的。但是，这些算法往往只是能够正确地计算出凸多边形的一个子集，且还可能包含某些错误的凹多边形。例如，一个严格的凸多边形，其内角小于 180°。然而，该凸面测试只是一个“必要不充分”检测。因此，只测试多边形的内角将会得出错误的结论：五角星将会是一个凸多边形（如图 3.18（c）所示）。因此，该测试只是假设：已知当前多边形为非自相交多边形时才发挥作用。

另一个凸性测试中值得注意的因素是：当测试在凸多边形顶点间移动，并形成从首顶点到尾顶点的某种环绕方式时，沿任一轴向都只有两种方向性的变化。为了检测如图 3.18（d）所示的 Z 字形图元，需要针对这两种不同方向所产生的变化分别加以测试。例如，分别沿 x 轴、y 轴执行测试。该算法虽然貌似正确，但当凸多边形的所有顶点投影至一条直线上时，算法将出现错误。可以同时采用上述两种测试方案，从而实现一个健壮的组合测试，即两种方案彼此“保护”。组合测试的实现在[Schorn94]中加以描述。

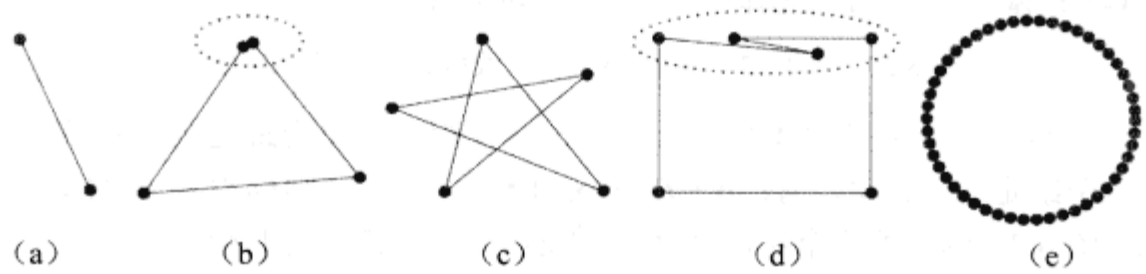


图 3.18　凸体测试时可能导致问题的输入数据。（a）线段。（b）顶点重叠的四边形。（c）五角星形。（d）包含两个额外共边顶点的四边形。（e）顶点共圆

3.8　多面体

多面体（Polyhedra）是多边形的 3D 对照物，并且以多面实体的方式体现了一个空间边界连通域。多面体的边界由多个连接的多边形面组成，每一条多边形的边都是两个面的公共边（如图 3.19 所示）。其他一些定义允许多面体无界，即多面体可以沿某一方向无限延伸。

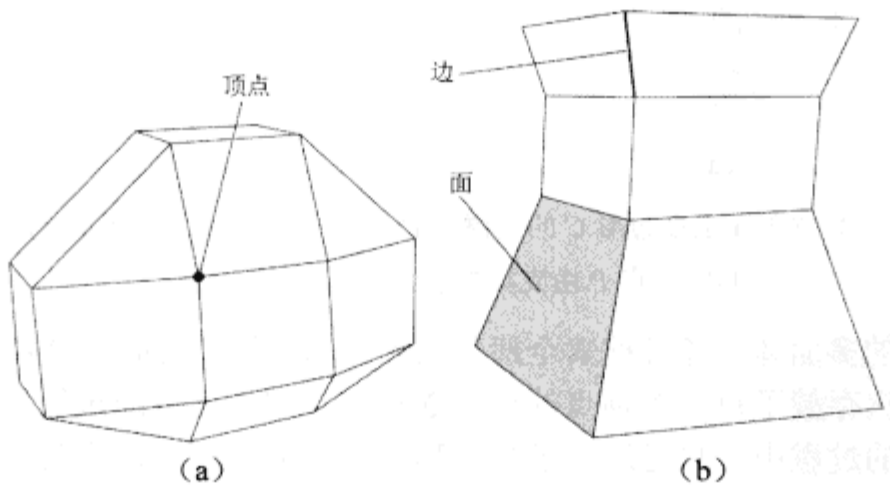


图 3.19　（a）凸多面体。（b）凹多面体。其中，相应的数据面、边以及顶点均已标出

相对于多边形，多面体边界将空间分割为两个分离区域：内部和外部。当多面体内部边界确定的点集为凸状时，多面体为凸体。边界凸多面体也称作多面体（Polytope）。类似于多边形，多面体也可描述为有限数量的半空间的交集。

d-单形体是指：d 维空间内，d+1 仿射无关的点集形成的凸包。单形体也称作给定 d 值的 d-单形体。例如，0-单形体是一个点；1-单形体是一条线段；2-单形体是一个三角形；3-单形体是一个四面体（如

图 3.20 所示）。单形体的一个性质是：从相关定义集合中除去一点，将使单形体的维度减 1。

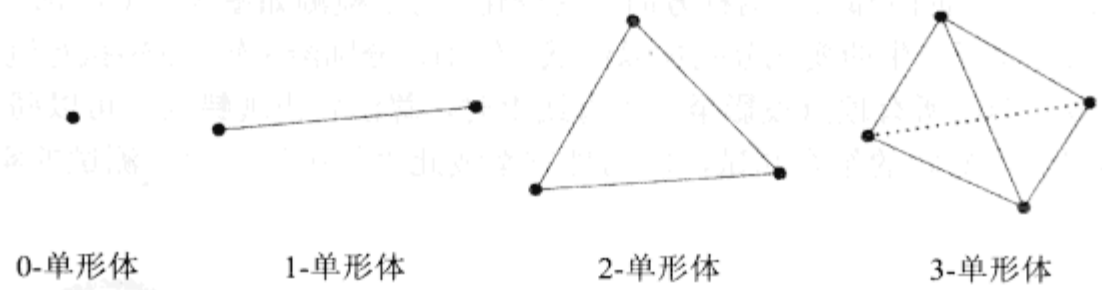


图 3.20 （0～3）-单形体：点、线段、三角形和四面体

对于凸体集 C（不必是一个多面体），沿给定方向且距离该集合最远的一点称作 C 的支撑点。特别地，如果对于一个给定的方向 $\mathbf{d}$，有：$\mathbf{d}\cdot P=\max\{\mathbf{d}\cdot V:V\in C\}$，则称 P 为 C 的支撑点，即点 P 使得 $\mathbf{d}\cdot P$ 存在最大值。图 3.21 显示了两个不同凸体集的支撑点。支撑点有时也称作端点，且不必是唯一的。对于一个多面体，顶点集中的相应一点，通常可作为某一给定方向上的支撑点。

当支撑点是一个顶点时，该顶点一般称作支撑顶点。

支撑映射（或称支撑函数）作为一个函数，记作 $S_C(\mathbf{d})$，且关联一个凸体集 C，将方向 $\mathbf{d}$ 映射至 C 的支撑点。对于简单的凸体（如球体、盒体、圆锥和圆柱体），支撑映射以闭合方式加以限定。例如，对于球心为 O、半径为 r 的球体 C，支撑映射定义为 $S_C(\mathbf{d})=O+r\mathbf{d}/\|\mathbf{d}\|$（如图 3.21（b）所示）。具有更高复杂度的凸体，则要求支撑映射函数利用相应的数值方法确定支撑点。

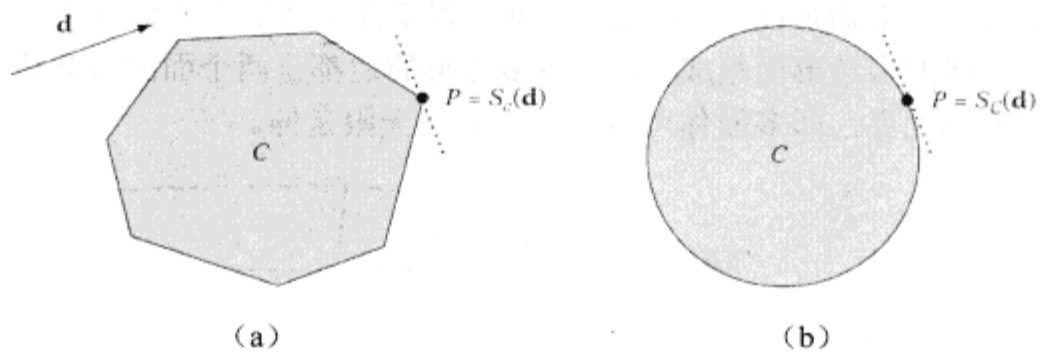


图 3.21 （a）方向 $\mathbf{d}$ 上多边形 C 的支撑点 P。（b）方向 $\mathbf{d}$ 上圆形 C 的支撑点 P。其中，点 P 由支撑映射函数 $S_C(\mathbf{d})$加以定义

对于包含 n 个顶点的多面体，通过搜索全部顶点来计算支撑点的时间消耗是 $O(n)$级别的。假设存在这样一个数据结构，它存储了每一个顶点的全部邻接点，端点可以通过简单的爬山法得到。该方案十分高效：在接近端点的过程中，只是遍历了各自顶点中的一个较小的点集。对于大型多边形，可以人工添加一个或多个邻居至顶点的邻接表中，从而加速爬山法的计算过程。通过预计算顶点的权值，定位支撑点的时间消耗可以达到 $O(\log n)$级别。上述加速支撑点搜索的相关算法将在第 9 章详细讨论。

当一个平面包含支撑点，并且以一个给定的方向作为法线，则该平面称作支撑面。另外，如果全部多面体的顶点（且多个顶点可以构造该多面体的质心）皆位于某一平面的同一侧，则称该平面“支撑”多面体。通过顶点集$\{P_1,P_2,\cdots,P_n\}$定义的多面体质心，则是顶点位置坐标的算术平均值$(P_1+P_2+\cdots+P_n)/n$。

两个凸体集的分割面是指，其中一个凸体集完全位于该面的正（开放）半空间；另一个凸体集则

完全位于该面的负（开放）半空间中。垂直于该分割面的轴（平行于分割面的法线）称作分离轴。对于两个不相交的凸体集，总是存在一个分离轴（或分割面），然而相同情况对于凹体集则不成立。关于分离轴的详细讨论可参见第 5 章。

多面体的表面常称作 2-流形。该拓扑术语表明表面各顶点的邻接性拓扑等价（同胚）于一个圆平面。因而，2-流形多面体表面意味着多面体的一条边必定连接两个面（或边上一顶点的邻接性将不会呈现圆平面状）。

多面体的顶点（V）、面（F）以及边（E）之间的数量关系遵循欧拉公式 $V+F-E=2$。另外，对于孔状多面体也存在对应的欧拉公式。关于欧拉公式，本书将在第 12 章加以讨论。

3.8.1　凸体测试

类似于多边形的凸面测试，当且仅当对于凸体 P 的每一个面，P 的所有顶点皆（严格）位于该面的同一侧时，P 为凸体。健壮的计算过程应包括对多面体各面的重合点和共线边进行独立测试，同时，对于点的重合度计算以及边的共线性计算，通常会设定一个误差范围。上述凸体测试的复杂度为 $O(n^2)$。

下面考虑一个复杂度为 $O(n)$的算法：对于多面体 P 的每一个面 F，计算该面的质心 C，同时，对于面 F 的每一个邻接面 G，测试质心 C 是否处于邻接面 G 的支撑面的背面。若存在质心 C 未能位于一个或多个邻接面的支撑面的背面，则 P 为凹体，否则 P 暂假定为凸体。应注意，由于相应的凸多边形测试对于五角星是无效的，因此上述凸体测试对于“五角星体”也将是无效的。

3.9　凸包计算

在某些应用中，凸包可视为碰撞几何体的最小（最紧凑）包围体，因此，存在多种求解凸包的算法。下面两个小节将简要地介绍两种算法：Andrew 算法和 Quickhull 算法。

3.9.1　Andrew 算法

一种比较健壮且易于实现的 2D 凸包算法是 Andrew 算法[Andrew79]。第一次遍历时，从左到右对给定的顶点集实现全顶点排序；在后续的第二次和第三次遍历中，可形成从最左到最右顶点的边集（chains of edges），分别对应凸包的上半部分边集和下半部分边集。最后，只需简单地连接两个边集的尾顶点，即可形成凸包。这一过程如图 3.22 所示。其中，核心步骤为凸包边集的生成。如图 3.22 中的左中图所示，该图给出了其中一个边集的构造过程，即构造上半部分边集。为了表述简便，假设顶点间的 x 坐标彼此不同（该假设稍后撤销）。

最初，两个最靠近左边的顶点 A、B 产生了边集中的第一条边，对其余的顶点依次加以处理，待确认后逐个加入到凸包的边集中。如果待考查的下一个顶点位于边集中当前边的右侧，则该点暂定为凸包上的顶点，并加入到边集中；若待考查的下一个顶点位于边集中当前边的左侧，则该点将位于凸包的外部并处于错误状态。因此，已加入边集中的最后一个顶点将被剔除，并开始下一次测试。顶点的剔除过程重复执行，直到出现位于边集中当前边右侧的顶点，且将该顶点加入至边集中。

示例中的下一个顶点为 C，且顶点 C 位于边 AB 的左侧，则临时凸包出现错误且从边集中剔除顶点 B。由于再没有顶点可供删除（顶点 A 作为最左顶点，须位于凸包中），顶点 C 加入至边集中，从而形成凸包边集 A-C。下一个顶点为 D 且位于边 AC 的右侧，因而加入至边集中。接下来，顶点 D 位于边 AC 的右侧且加入至边集中；顶点 E 位于边 CD 的右侧且加入至边集中；顶点 F 同理。顶点 G 位于边 EF 的左侧，则凸包处于错误状态，顶点 F 将从边集中剔除。下一步，顶点 G 也位于边 DE 的左侧，则顶点 E 将从边集中剔除。最终，顶点 G 位于边集中当前边 CD 的右侧，并将其加入至边集中——此处形成了 A-C-D-G。继续处理后续顶点，最终上半部分的边集如图 3.22 中的右中图所示。同时，可以采用类似的步骤构造下半部分边集。这里仍存在一个遗留问题：处理 x 坐标相同的顶点。最直接的方法是将上方的顶点加入至上半部分边集中，将下方的顶点加入至下半部分边集中。

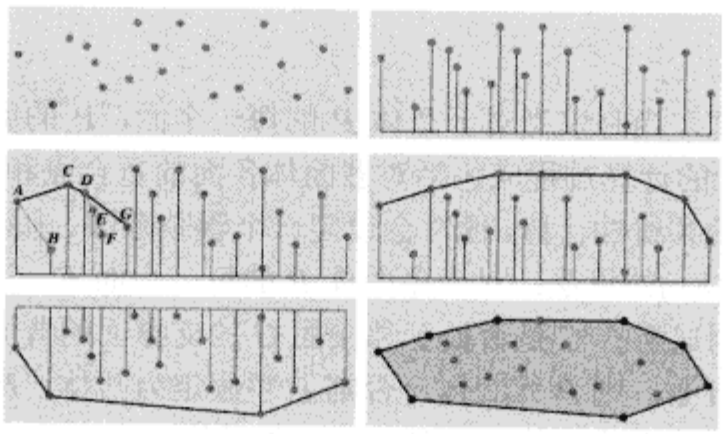


图 3.22　Andrew 算法。左上图：点集。右上图：从左向右（按照字典顺序）存储顶点。左中图：构造上半部分边集。右中图：构造完毕后的上半部分边集。左下图：下半部分边集。右下图：两个边集生成的最终凸体

实现上述版本的 Andrew 算法并不困难。这里，Andrew 算法更益于处理数组表达的顶点集。

3.9.2　Quickhull 算法

虽然 Andrew 算法在 2D 空间内工作良好，但对于 3D 空间内的扩展应用，该算法尚无法给出明确的解释。因此，另一个可以运行于 2D 空间和 3D 空间的替代算法是 Quickhull 算法（快包算法）。Quickhull 算法的基本概念十分简单，其 2D 空间示例如图 3.23 所示。

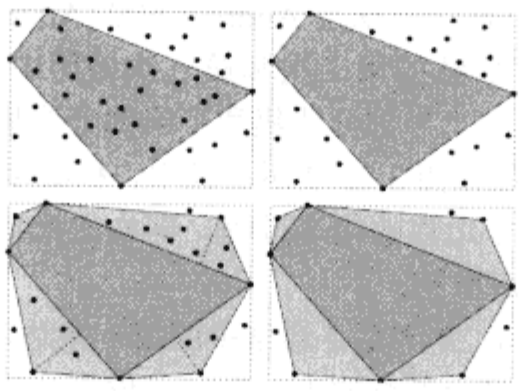


图 3.23　Quickhull 算法步骤分析。左上图：确定 4 个极值点（位于点集的包围盒上）。右上图：删除极值点内部区域中的顶点，这一类顶点并不位于凸体之上。左下图：针对该区域中的各边，计算相应的最远点。右下图：删除位于三角形区域中的全部顶点。随后，算法递归计算距三角形各边的最远点

首先获取顶点集的包围盒，一般情况下可以定位 4 个端点（分别位于包围盒的 4 条边上）。作为端点，上述 4 点定位于凸包上并构成了其“首次逼近”，如图 3.23 中的左上图所示。进一步考查，位于该四边形内部的顶点一定不在凸包之上，因此可以删除（如图 3.23 中的右上图所示）。位于该四边形之外的一些点，则可能位于凸包之上。这样，首次逼近得到修正。下面，对于四边形的每一条边，计算边外部距离最远的顶点（如果存在），由于这类点都是垂直于边的端点，因此必定是凸包上的点且需要加入到凸包中（如图 3.23 中的左下图所示），其中，边的两个顶点和这一个新的凸顶点形成了一个三角形，如前所述，三角形内部的点不会是凸包上的顶点且可以丢弃（如图 3.23 中的右下图所示）。对于凸包上的新边，递归地重复执行前述同一过程，直至没有顶点位于边外时终止。

虽然上述内容阐述了全部算法描述，但对于健壮的算法实现，还应考虑到两个细节。首先对于凸包的首次逼近计算，其计算过程可能无法构造四边形。例如，如果一个端点位于左下角，则初始形状将是一个三角形；如果另一个端点位于右上角，则结果可能只是一个包含两个顶点的、退化的凸包。为了避免发生上述问题，算法实现必须能够处理包含不同边数量的、凸包的首次逼近计算。第二，包围盒的某一条边上可能包含多个顶点，或者距离边最远的顶点也可能不止一个。在这种情况下，首先计算距离边首尾顶点最近的两个顶点 M、N，并选择其中之一作为端点。任何位于顶点 M、N 之间的顶点通常可以不计为凸包顶点：它们位于边上且与边首尾顶点共线。

给定由顶点 A、B 确定的一条边，点集 P 中距离该边最远的一点可按下列方式计算：将顶点投影到边 AB 的正交轴上，且正交轴上投影值最大的顶点为所求顶点，若这样的顶点不止一个，则再次计算边 AB 上的最大投影值。这一过程如下列代码所示：

```
// Return index i of point p[i] farthest from the edge ab, to the left of the edge
int PointFarthestFromEdge(Point2D a, Point2D b, Point2D p[], int n)
{
    // Create edge vector and vector (counterclockwise) perpendicular to it
    Vector2D e = b – a, eperp = Vector2D(-e.y, e.x);
    // Track index, 'distance' and 'rightmostness' of currently best point
    int bestIndex = -1;
    float maxVal = -FLT_MAX, rightMostVal = -FLT_MAX;

    // Test all points to find the one farthest from edge ab on the left side
    for (int i = 1; i < n; i++) {
        float d = Dot2D(p[i] – a, eperp);        // d is proportional to distance along eperp
        float r = Dot2D(p[i] – a, e);            // r is proportional to distance along e
        if (d > maxVal || (d == maxVal && r > rightMostVal)) {
            bestIndex = i;
            maxVal = d;
            rightMostVal = r;
        }
    }
    return bestIndex;
}
```

Quickhull 算法可应用于 3D 空间内（或更高维度，参见[Barber96]）。在 3D 空间中，首次逼近计算始于包含全部点集的、轴对齐包围盒之上的 6 个端点（或更多）。此处不再计算距离多边形边最远的顶点，而是获取与多面体中的面距离最远的顶点。类似地，无须为了获取凸包顶点而对边执行分解

计算——而可以将多面体上的面分解为两个或多个面。

健壮的 Quickhull 算法（Qhull，Brad Barber）可以从网络上获取。关于凸包算法的详细讨论可参见[O'Rourke98]。

3.10 域

与众多相交测试相关的一个重要概念是 Voronoi 域。其中，给定一个平面上的点集 S，S 内一点 P 的 Voronoi 域定义为：点集 S 内，距离 P 最近的面内点集。Voronoi 域和 Voronoi 图这两种概念来源于计算几何分支领域，并且应用于最近邻接节点的查询计算。

下面对 Voronoi 域这一概念稍加扩展，使其可以应用于碰撞检测系统。给定一个多面体 P，且令 P 的特征之一为顶点、边或面。则针对 P，其特征 F 的 Voronoi 域为：与 P 的其他特征相比，空间内距离 F 最近的点集。图 3.24 显示了由三角形特征确定的 Voronoi 域。图 3.25 显示了一个立方体 3 种类型的 Voronoi 特征域。需注意的是，本书常混合使用 Voronoi 域和 Voronoi 特征域这两个术语，读者应避免将 Voronoi 域和 3.4 节中讨论的质心域混淆。另外，Voronoi 域的包围面称作 Voronoi 面。

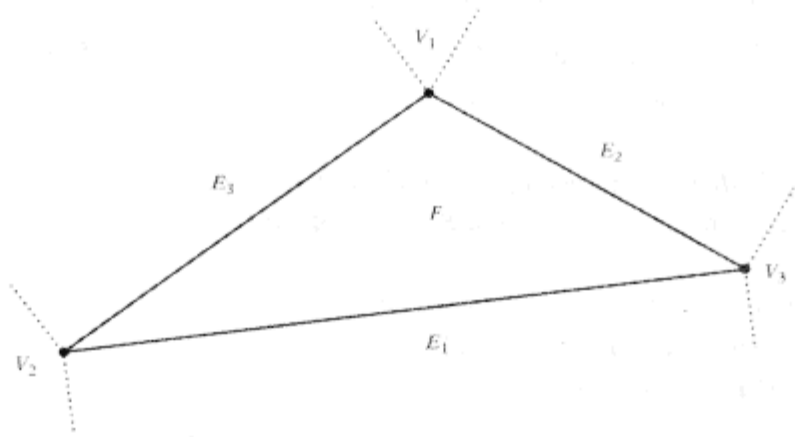


图 3.24　三角形将其支撑面划分为 7 个 Voronoi 特征域：一个面域（F）、3 个边域（E_1、E_2、E_3）以及 3 个顶点域（V_1、V_2、V_3）

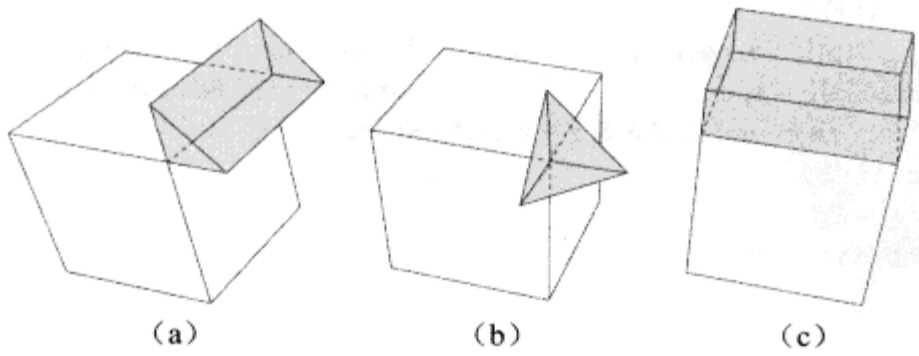


图 3.25　3D 立方体 Voronoi 特征域。（a）边域。（b）顶点域。（c）面域

下面给定一个凸多面体 P，则在空间内且位于 P 外部的全部顶点可分为位于 P 的顶点、边或面的 Voronoi 特征域，且存在一个由两个相邻 Voronoi 特征域形成的包围。需注意的是，该包围只属于两个相邻 Voronoi 特征域中的一个。由于 Voronoi 域构造了一个多面体外部空间划分，所以，对于空间内的

点 Q，利用 Voronoi 域可以计算凸多面体物体上的距离 Q 最近的一点。即可以遍历各特征域，直至在某一特征域内获取 Q。因此，该凸体物体上距离 Q 最近的一点，将是 Q 在该特征（且与其既定特征域相关）上的投影。对于重复查询，可以采用“连续查询”的方法，即记录上次最近点所处的特征域，这将从此处开始新一轮的查询。Voronoi 域的概念将应用于某些相交测试，并在第 5 章加以描述。另外，Voronoi 域也将在第 9 章中讨论，并用于计算凸多面体间的相交测试。

3.11　Minkowski 和与 Minkowski 差

有两个重要的点集操作将贯穿本书的多个章节：Minkowski 和与 Minkowski 差。令 A 和 B 为两个点集，且令 $\mathbf{a}$ 和 $\mathbf{b}$ 为两个位置向量，对应 A 和 B 中的顶点。Minkowski 和 $A \oplus B$ 定义为：

$$A \oplus B = \{\mathbf{a}+\mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$$

其中，$\mathbf{a}+\mathbf{b}$ 为位置向量的和。直观地看，Minkowski 和可视为由 A（转换至 B，反之亦然）扫掠后的相应区域，且 Minkowski 和的计算过程如图 3.26 所示。

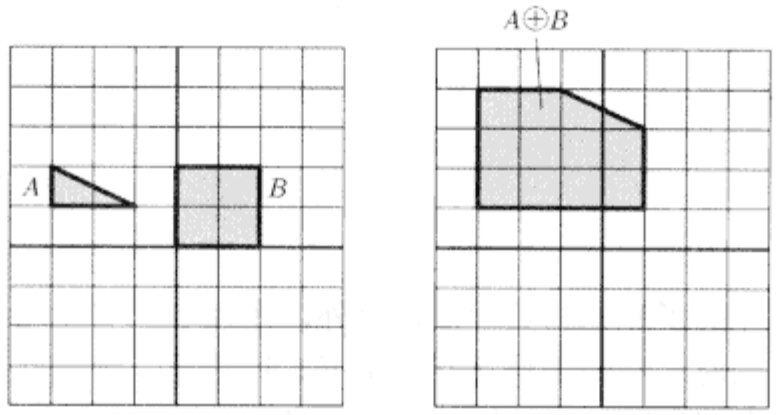


图 3.26　三角形 A 和正方形 B 的 Minkowski 和

类似地，两个点集 A 和 B 的 Minkowski 差定义为：

$$A \ominus B = \{\mathbf{a}-\mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$$

在几何学中，Minkowski 差可以看作 A 与 B 相对原点的反射向量之间的加法，即 $A \ominus B = A \oplus (-B)$，如图 3.27 所示。因此，上述两个术语常简称为 Minkowski 和。对于两个凸多边形 P 和 Q，Minkowski 和 $R=P \oplus Q$ 具有下列特征：R 也是一个凸多边形且其顶点为 P、Q 顶点之和。同理，两个凸多面体的 Minkowski 和也具有上述特征。

Minkowski 和常直接或间接地用于碰撞检测中。考查下列情景可能产生的问题：一个形状复杂的物体将穿越若干具有同等复杂形状的障碍物，为了避免对复杂物体执行耗时的测试，障碍物可以相对于物体“膨胀”，同时，物体“收缩”为一点，使得其计算过程转换为一个运动点与“膨胀”的障碍物之间的碰撞测试。这一概念将在第 8 章中结合 BSP 树进行深入的讨论。

从碰撞测试的角度来看，Minkowski 差是十分重要的概念：当且仅当 Minkowski 差 C（$C=A \ominus B$）包含原点时，顶点集 A 和 B 发生碰撞，如图 3.27 所示。实际上，可以构造更直接的结果：计算 A、B 之间的最小距离等价于 C 和原点之间的最小距离。这一结果将用于第 9 章涉及的 GJK 算法中，且有：

$$\text{distance}(A,B)=\min\{\|\mathbf{a}-\mathbf{b}\| : \mathbf{a} \in A, \mathbf{b} \in B\}=\min\{\|\mathbf{c}\| : \mathbf{c} \in A \ominus B\}$$

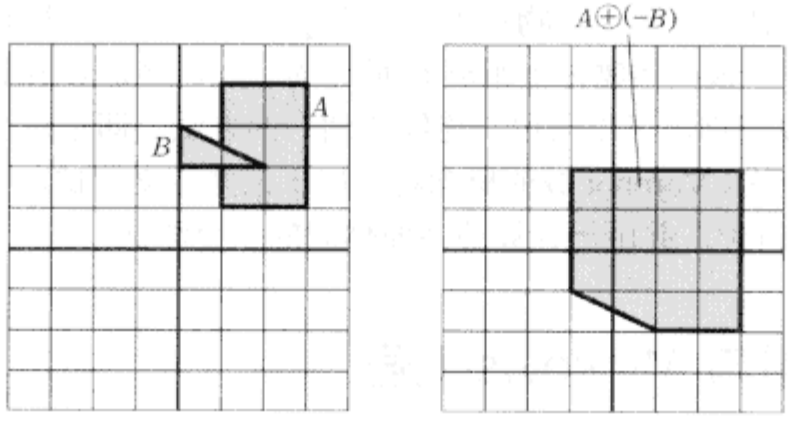


图 3.27　矩形 A 和三角形 B 相交，因而原点位于 Minkowski 差内

需注意的是，两个凸集的 Minkowski 差仍是一个凸集，且其最小范数顶点是唯一的。

相关算法介绍了如何显式地计算 Minkowski 和（如[Bekker01]）。在本书中，主要是利用 Minkowski 和这一概念，将碰撞问题转化为与其对应的另一解决方案。有时，两个物体的 Minkowski 和会执行隐式计算（如 GJK 算法）。

两个物体的 Minkowski 差有时也称作“平移位形空间障碍物（TCSO）”，TCSO 上的查询操作则称为位形空间计算。

3.12　小结

进行碰撞检测的计算工作除了基础的数学知识以外，还需要扎实的几何和线性代数功底。本章带领读者回顾了相关的概念，同时这些知识将贯穿于全书。特别地，读者需要深刻理解下列知识点：点积、叉积以及标量三重积，它们常用于图元相交测试的演算过程（参见第 5 章）。如果读者对上述知识不是十分了解，可以参考相应的线性代数教材。

同时，本章还复习了大量的几何概念，包括顶点、直线、光线、线段，可参考[Mortenson99]以了解更多的几何知识。

本章还浏览了与计算几何、凸集理论相关的概念，并讨论了计算最近点时常用的重要理论——Voronoi 域。另外，对于非相交的凸体对象，分割面和分离轴则保证了划分物体时的高效测试，其进一步讨论可参见第 5 章和第 9 章。Minkowski 和与 Minkowski 差操作使得碰撞检测问题以一种不同的方式加以计算，从而使相应求解过程更为简便。第 5 章和第 9 章将讨论这类转化。

作为计算几何学中一本优秀的入门指南，[O’Rourke98]可供读者参考。关于凸集理论（侧重于凸体优化）的讨论，读者可以阅读[Boyd04]。

第 4 章　包　围　体

直接对两个物体对象的几何体执行碰撞测试，其计算过程往往代价高昂，特别是当物体包含成百上千个多边形时。为了减少计算消耗，在几何相交测试之前，通常先执行物体的包围体测试。

包围体（BV）是一个简单的体空间，且包围了一个或多个具有复杂形状的物体。这种简单体（如包围盒和球体）测试与具有复杂形状的物体相比，具备一定的计算优势。利用包围体可以执行快速的剔除测试，因为只有当包围体产生碰撞时，才有必要进一步计算具有复杂形状的几何体测试，如图 4.1 所示。

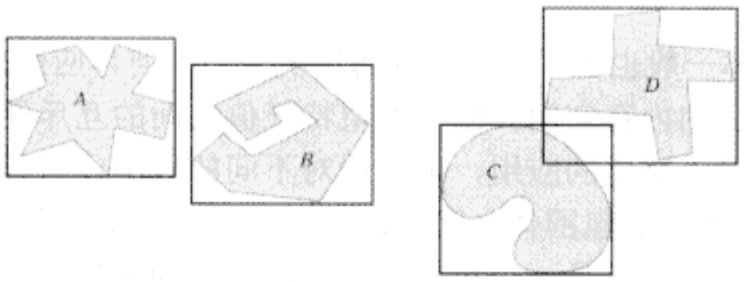


图 4.1　对象 A、B 间的包围体处于分离状态，因而对象 A、B 也不相交。
由于对象 C、D 的包围体相交，因而需进一步对二者执行相交计算

当然，当物体对象之间真正产生碰撞时，进一步的处理将导致计算量的显著增加。然而，在大多数情况下只是存在少数物体彼此靠近并可能产生碰撞。因此，包围体通常可以获取有效的性能改善；同时，复杂物体对象的前期剔除，更验证了为包围体测试所付出的较小代价是值得的。

对于某些应用程序，包围体测试可以提供足够的碰撞依据。对于未曾产生碰撞的某一部分，削减其包含对象的数量仍然是值得的——这样可以限制对包围体内部冗余多边形的进一步测试。物体对象 A、B 之间的多边形测试，其复杂度一般是 $O(n^2)$。因此，如果待测试多边形的数量能够减少一半，则相应的工作量将减少 75%。在第 6 章中，依据包围体的层次结构，可以将参与测试的物体对象和多边形数量减少至最低值，详细内容可参见第 6 章，本章所讨论的内容只限于一对包围体之间的测试。更进一步，这里所探讨的测试，主要包含同一类型的包围体之间的测试，但是，同时在多种类型的包围体之间执行测试并不常见。一些异类型 BV 之间的相交测试将在第 5 章中加以讨论。

存在多种几何形状适用于包围盒，本章将集中讨论应用较多的几种，包括球体、盒体和类凸体。对于某些不常用的包围体也将在 4.7 节中加以说明。

4.1　BV 期望特征

并非所有几何对象都可以充当有效的包围体。包围体的期望特征包括：

- ☑ 低耗费的相交测试。
- ☑ 实现紧密拟合。
- ☑ 计算耗费较少。
- ☑ 易于旋转和变换。
- ☑ 内存占用较少。

包围体背后的核心理念为：优先使用代价低廉的测试类型，并且尽量使测试提早退出。为了支持这种低消耗的相交测试，包围体应具有简单的几何形状。同时，为了使这种“提早退出”尽可能高效，包围体形状应尽量与物体对象之间实现较紧密的拟合，并且在紧密拟合与测试耗费之间形成某种平衡。另外，相交测试不应仅限于同类型包围体之间的比较，还应包含异类型包围体之间的测试。最后，相应的测试应涵盖下列计算查询：点包含、光线-体相交测试，以及面-多边形相交测试。

一般情况下，包围体计算须采用预处理方式而非实时计算。需注意的是，当包围体包含的物体移动时，一些包围体需要实现空间重对齐。因此，如果包围体的计算代价高昂，重对齐包围体就是一个更可取的方法。

除了几何体数据，包围体一般也占用内存。理想情况下，几何体应额外附加少许内存数据，因此，简单的几何体形状将占用较少的内存空间。由于期望特征值之间的互斥性，没有哪一种包围体适用于所有的情况，相比较而言，对于特定的应用，可以针对不同的包围体执行相关测试，最终确定较好的一个。图 4.2 所示为 5 种常用类型的包围体之间的比较。给定的顺序依次为边界简单、易于剔除、测试速度较快以及占用较少内存——这只是一个粗略的约定，而非绝对的准则。本章所涉及的第一个包围体实例为轴对齐的包围盒，将在 4.2 节中加以描述。

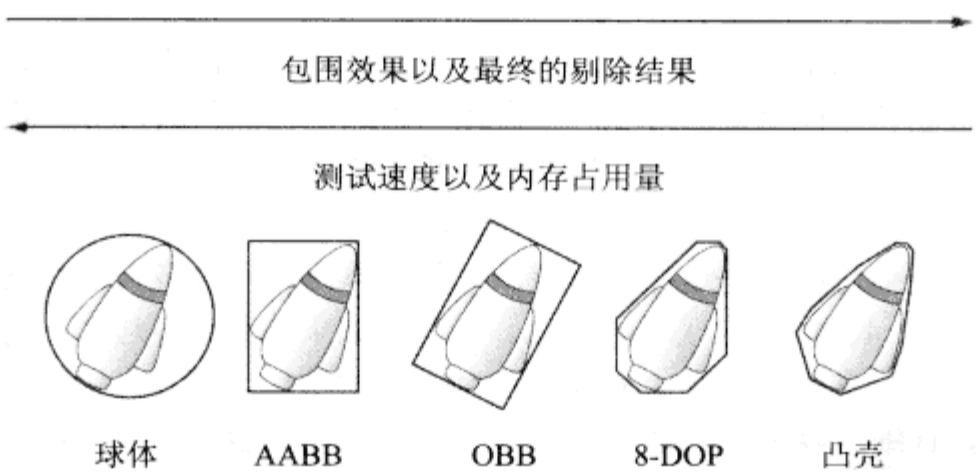


图 4.2　包围体类型：球体、轴对齐包围盒（AABB）、有向包围盒（OBB）、8-DOP 以及凸壳

4.2　轴对齐包围盒

轴对齐包围盒（AABB）是应用最为广泛的包围体之一。在 3D 空间中，AABB 是一个 6 面盒状长方体（在 2D 空间中是包含 4 条边的矩形），且其面方向性划分按如下方式进行：面法线皆平行于给定的坐标轴。AABB 的最大特点是能够实现快速的相交测试，即仅执行相应坐标值之间的比较。

AABB 存在 3 种常规表达方式，如图 4.3 所示。其中一种表达方式是采用各坐标轴上的最小值和

最大值，如下所示：

```
// region R = (x, y, z) | min.x<=x<=max.x, min.y<=y<=max.y, min.z<=z<=max.z
struct AABB {
        Point min;
        Point max;
};
```

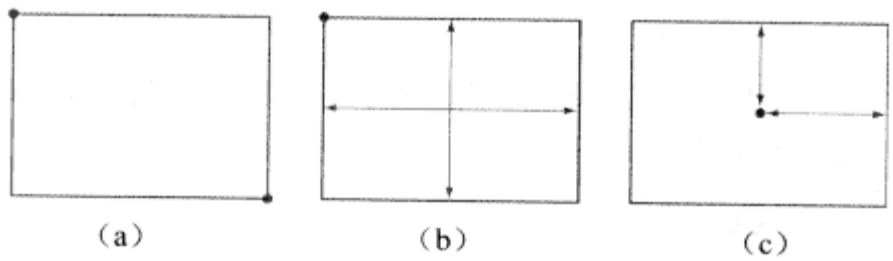


图 4.3　3 种常见的 AABB 表达方式。（a）最小值-最大值。（b）最小值-宽度值。（c）中心位置-半径

该表达方式将 BV 空间区域界定为两个对角顶点：*min* 和 *max*。另一种表达方式则利用最小顶点值和一个直径范围 *dx*、*dy* 和 *dz*，如下所示：

```
// region R = (x, y, z) | min.x<=x<=min.x+dx, min.y<=y<=min.y+dy, min.z<=z<=min.z+dz
struct AABB {
        Point min;
        float d[3];     // diameter or width extents (dx, dy, dz)
};
```

最后一种表达方式给定 AABB 的中心点 *C*，以及各轴向半径 *rx*、*ry*、*rz*，如下所示：

```
// region R =(x, y, z) | |c.x-x|<=rx, |c.y-y|<=ry, |c.z-z|<=rz
struct AABB {
        Point c;            // center point of AABB
        float r[3];         // radius or halfwidth extents (rx, ry, rz)
};
```

若考虑到存储需求，则“中心-半径”方式最为经济，与中心位置坐标值相比，半径则只需几个位；虽然存在较小的差异，但“最小值-直径”方式基本与“中心-半径”方式相同；最坏情况则是“最小值-最大值”方式，其中全部 6 个值都需要以相同的精确度加以存储，而为了降低存储空间，需要使用整数表示 AABB，而非浮点数。如果物体只是以平移方式运动，则与“最小值-最大值”相比，另外两种表达方式将更加方便——只需更新 6 个参数中的 3 个。而“中心-半径”方式的一个比较有用的性质是：可以作为一个包围球加以检测。

4.2.1　AABB 间的相交测试

AABB 之间的相交测试比较直观且无须考虑其表达方式。若 AABB 在 3 个轴向上都相交，则 AABB 相交，其中，AABB 沿每一个维度的有效范围可以看作是对应轴上的数值区间。则对于“最小值-最大值”方式，有下列区间相交测试：

```
int TestAABBAABB(AABB a, AABB b)
{
    // Exit with no intersection if separated along an axis
    if (a.max[0] < b.min[0] || a.min[0] > b.max[0]) return 0;
    if (a.max[1] < b.min[1] || a.min[1] > b.max[1]) return 0;
    if (a.max[2] < b.min[2] || a.min[2] > b.max[2]) return 0;
    // Overlapping on all axes means AABBs are intersecting
    return 1;
}
```

相比较而言，“最小值-直径”方式则不那么吸引人——即使以简洁的方式编写代码，就操作符数量而言，仍无法与上述测试相比，如下所示：

```
int TestAABBAABB(AABB a, AABB b)
{
    float t;
    if ((t = a.min[0] - b.min[0]) > b.d[0] || -t > a.d[0]) return 0;
    if ((t = a.min[1] - b.min[1]) > b.d[1] || -t > a.d[1]) return 0;
    if ((t = a.min[2] - b.min[2]) > b.d[2] || -t > a.d[2]) return 0;
    return 1;
}
```

最后，“中心-半径”方式的相交测试代码如下所示：

```
int TestAABBAABB(AABB a, AABB b)
{
    if (Abs(a.c[0] - b.c[0]) > (a.r[0] + b.r[0])) return 0;
    if (Abs(a.c[1] - b.c[1]) > (a.r[1] + b.r[1])) return 0;
    if (Abs(a.c[2] - b.c[2]) > (a.r[2] + b.r[2])) return 0;
    return 1;
}
```

在现代计算机体系结构中，函数 Abs()调用一般转化为一条指令，否则，可以通过判断浮点值二进制格式的符号位高效地实现该函数。当 AABB 域声明为整数而非浮点数时，上述测试可以替换为另一个版本。使用整数时，区间$[A, B]$和$[C, D]$可以按照下列表达式加以计算：

```
overlap = (unsigned int)(B - C) <= (B - A) + (D - C);
```

当 $C > B$ 时，无符号整数下溢，左边的计算结果将变得不可预测，从而使该表达式产生错误。而上溢则可替代绝对值函数调用，使得“中心-半径”方式的测试如下所示：

```
int TestAABBAABB(AABB a, AABB b)
{
    int r;
    r = a.r[0] + b.r[0]; if ((unsigned int)(a.c[0] - b.c[0] + r) > r + r) return 0;
    r = a.r[1] + b.r[1]; if ((unsigned int)(a.c[1] - b.c[1] + r) > r + r) return 0;
    r = a.r[2] + b.r[2]; if ((unsigned int)(a.c[2] - b.c[2] + r) > r + r) return 0;
    return 1;
}
```

利用整数也可以实现其他技巧，但其应用多依赖于平台的体系结构，如 SIMD 指令用几条代码即可实现 AABB 测试（相应的实例可参见第 13 章）。最后，对于需要执行大量相交测试的碰撞检测系统，可以根据状态的相似性对测试进行排序。例如，如果相应操作基本出现在 xz 平面，则 y 坐标测试应最后执行，从而使状态的变化降低到最小程度。

4.2.2　AABB 的计算与更新

包围体常采用其占据的局部模型空间，而为了执行两个包围体之间的相交测试，需要将它们转换为统一的坐标系统。这里存在两种选择：将两个包围体全部转换为世界坐标空间系统；将一个包围体转换为另一个包围体的局部坐标系统。局部空间转换的优势在于：只需执行一次世界坐标空间的转换，就能够产生更紧凑的包围体。图 4.4 阐述了这一概念。图 4.4（a）在世界坐标空间内计算了物体 A、B 的 AABB。应注意的是，如图 4.4（c）所示，在物体 B 的模型空间内，两个物体之间却是分离的。

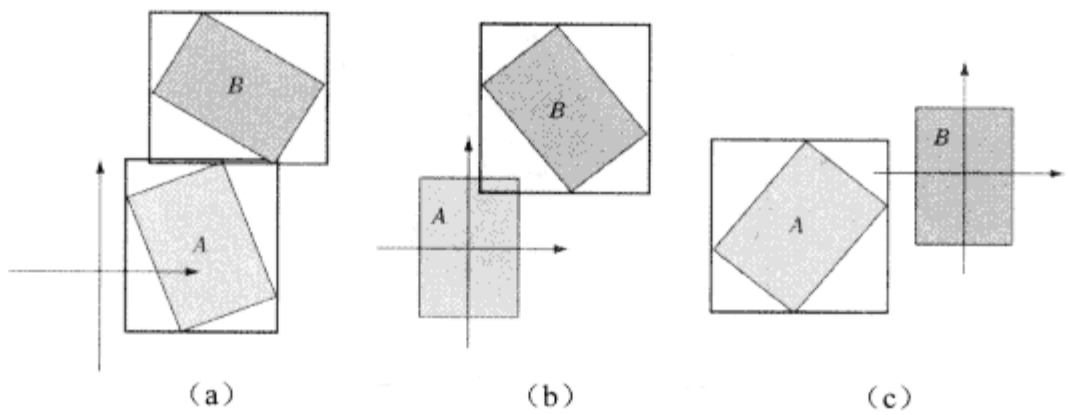


图 4.4　（a）世界空间内的 AABB：A 和 B。（b）局部空间 A 中的 AABB。（c）局部空间 B 中的 AABB

将包围体转换至另一个包围体的局部坐标空间时，精度是需要考虑的一个问题。当采用世界坐标空间执行测试时，可能会将物体移至距离原点较远处。因此，包围体近原点坐标转换时附加的平移操作将会使原值出现精度损失（丢位）。然而，当测试位于局部空间时，对象总是处于近原点位置，因而能够保持计算精度。需注意的是，通过调整平移操作，使得被转换物体的原点居中，则世界坐标转换也能够维护相应的精度。

在一个时间步内，当已更新的包围体可以被临时缓存时，值得考虑世界坐标空间的转换。通过缓存转换后的包围体，相对于某一给定的空间，全部包围体只需转换一次。当采取了世界空间坐标转换后，包围体将位于同一空间内，其中，物体的相交测试已被执行多次。相反，当转换至某一物体的局部空间时，缓存已更新的包围体不会带来任何帮助，因为全部变换只影响到当前新物体，或者是新的目标坐标系统。同时，缓存已更新的包围体还会导致存储空间成倍增长，因为在更新过程中，包围体中的大部分数据域都已发生了变化。

某些包围体（如球体和凸体），可以直接转换至任意一种坐标系统中，因为它们不受限于特定的方向，并称之为非对齐或自由定向的包围体。相反，对齐的包围体（如 AABB）则依赖于其既定的方向。另外，如果对齐的包围体在运动过程中存在旋转，且使得包围体处于非对齐状态，则需要重对齐包围体。而对于更新或重构 AABB，存在下列 4 条基本原则：

☑　使用固定尺寸且较为松散的 AABB 包围物体对象。

☑ 基于原点确定动态且紧凑的重构方式。
☑ 利用爬山法确定动态且紧凑的重构方式。
☑ 基于旋转后的 AABB，确定近似且动态的重构方式。

4.2.3～4.2.6 节将详细地介绍上述方案。

4.2.3 基于包围球的 AABB

第一种方法通过在任意方向上全包围物体对象，从而实现重构造 AABB。其中，物体 A 的包围球形成的包围盒将作为包含该 AABB 的固定尺寸加以计算，且物体可以围绕球心 P 旋转；半径 r 为球心距离物体最远顶点的距离，如图 4.5 所示。若点 P 与物体中心位置重合，则可保证当前球体半径为最小半径。

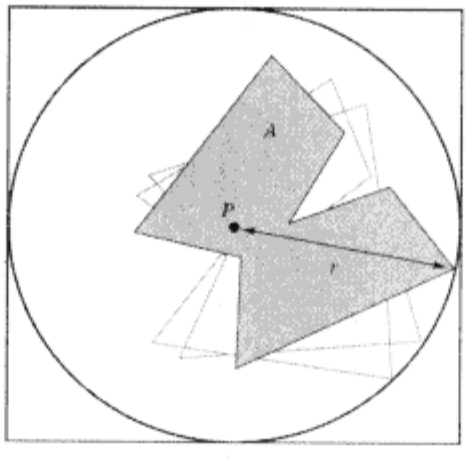


图 4.5 在任意方向上完全包含对象 A 的包围球 AABB

该表达方式的优点之一是：在更新 AABB 的过程中只需考虑平移变换（该变换也应用于物体之上），且可以忽略旋转变换。然而，包围球自身也具备上述特征（优于 AABB），因此，在此例中包围球可视为包围体的一个较好选择。

4.2.4 基于原点的 AABB 重构

本节所描述的更新策略（稍后将介绍其他两类更新策略）在 AABB 与各坐标轴重对齐时，将动态地调整其尺寸。对于一个紧密拟合的包围体，将检测其底层几何数据，同时沿坐标轴的 6 个方向搜索端点从而构造 AABB 的包围数据。一种比较直接的方法是遍历全部顶点，并记录各方向向量上的最大距离顶点。

该距离可以通过顶点向量在方向向量上的投影得到。由于需要进行比较，这里不必规范化方向向量。该计算过程如下列代码所示，其中记录了沿方向向量上的最近顶点和最远顶点：

```
// Returns indices imin and imax into pt[] array of the least and
// most, respectively, distant points along the direction dir
void ExtremePointsAlongDirection(Vector dir, Point pt[], int n, int *imin, int *imax)
{
    float minproj = FLT_MAX, maxproj = -FLT_MAX;
```

```
    for (int i = 0; i < n; i++) {
        // Project vector from origin to point onto direction vector
        float proj = Dot(pt[i], dir);
        // Keep track of least distant point along direction vector
        if (proj < minproj) {
            minproj = proj;
            *imin = i;
        }
        // Keep track of most distant point along direction vector
        if (proj > maxproj) {
            maxproj = proj;
            *imax = i;
        }
    }
}
```

当 n 值较大时，复杂度为 $O(n)$的算法会在运行期内付出高昂的代价，因此预处理顶点数据将会加速上述计算过程。可以忽略冗余顶点，这一简单方案基于下列事实：只有凸体上的顶点才会影响包围体形状（如图 4.6 所示）。在预处理过程中，相对于其他冗余顶点，凸体上的全部 k 个顶点将被提前存储。可以只检查这 k 个最初的顶点，从而构建紧凑的 AABB。

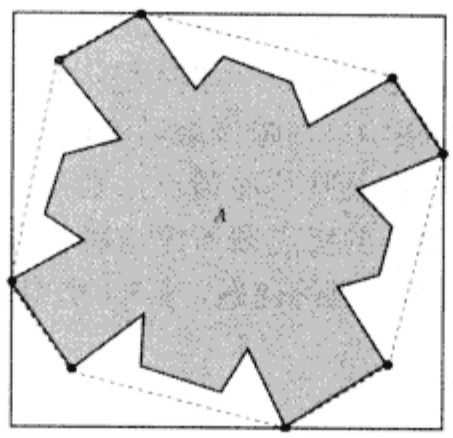


图 4.6　当计算紧凑 AABB 时，仅需考虑位于对象凸体之上的关键点

对于一般性凹体，该方法具有一定的优势，但针对凸体，其全部顶点已位于凸体之上，因此该方法收效甚微。

通过额外的特定预处理搜索结构，确定端点的时间消耗可降到 $O(\log n)$。例如，Dobkin-Kirkpatrick 层次结构（参见第 9 章）便用于此目的。但是，该结构将占用额外的内存空间且存在遍历时间消耗——这在多数情况下将是导致性能下降的直接原因。但是，如果紧凑包围体在计算中确实十分重要，可以考虑它而不考虑 AABB。

4.2.5　利用爬山法构造 AABB

另外一种加速 AABB 重对齐过程的处理方法是：快速计算物体顶点的邻接顶点。该方法只是通过简单的爬山法即可确定新 AABB 的各端点（如图 4.7 所示）。

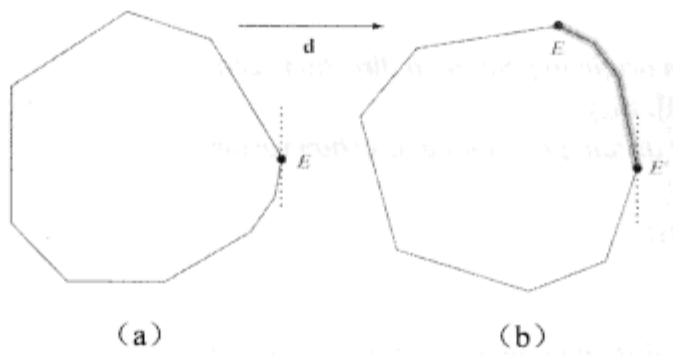


图 4.7 （a）方向 **d** 上的极值点 E。（b）待对象逆时针旋转后，可根据灰色标注的顶点路径并采用爬山法获取 **d** 向上的新极值点 E

这里不采用记录各轴向上最小值、最大值的方法，而是维护 6 个顶点指针，并指向各轴向的物体端点。爬山法将一个参考顶点和它的邻接顶点进行比较，并查看参考点是否仍为之前同一方向上的端点，如果不是，则端点替换为新的邻接顶点，重复该过程直至获得该方向上的最终端点。为了避免在局部最小值处阻塞，爬山法要求物体对象为凸体。因此，爬山法是基于非凸对象预计算后的凸体算法。综上所述，这种紧凑型 AABB 的计算方法是一种常量时间操作。

当爬山算法确实减少了相应的计算量时才考虑转换顶点。然而，相关实现可以进一步改善该算法：在某一给定轴上搜索端点时，可以仅使用 x、y、z 分量中的一个。例如，当在$+x$ 轴上获取端点时，只需计算被转换顶点的 x 分量——转换产生的消耗减少了 2/3。

为了实现健壮的爬山算法，还需要考虑其他一些问题。考查下列情形：沿任意轴上的一个端点以及围绕该轴的若干共面顶点。此时，如果物体对象围绕其余两轴中的任一轴旋转 180°，则该端点将位于同一轴上的反方向上，由于该端点存在某些共面顶点，则爬山法无法搜索更佳的邻接顶点，且止于当前搜索方向上的（最少）端点处。健壮的实现须考虑这种特殊情况：在预处理时可以先剔除共面顶点，具体内容可参见第 12 章。关于搜索端点的问题，9.5.4 小节中将加以讨论。

4.2.6 旋转 AABB 后的重计算

考虑前述 4 个 AABB 重构原则中的最后一项。其中，最简单的方法是利用一个新的 AABB 包围旋转后的 AABB，但这只是形成了一个逼近 AABB 而非紧凑 AABB。当结果 AABB 大于初始 AABB 时，需要注意的是，应在初始 AABB 的局部空间内计算旋转。否则，针对上一个时间步内的、旋转后的 AABB，重复旋转计算将使得 AABB 无限增长。

考查一个轴对齐的包围盒 A，且旋转矩阵 **M** 作用于 A 之上，结果将是具有某一方向的旋转包围盒 A。旋转矩阵的 3 个列（或行，取决于矩阵的优先规则）给出了 A 中的世界坐标轴（若向量为列向量且右乘矩阵，则矩阵 **M** 的列为坐标轴；若向量为行向量且左乘矩阵，则矩阵 **M** 的行为坐标轴）。

令包围盒 A 采用最小值-最大值形式，且 **M** 为列矩阵，则界定 A 的轴对齐包围盒 B 由下列方式确定：A 中 8 个旋转后的顶点在世界坐标轴上的投影间隔范围。例如，对于 B 的 x 值，只有矩阵 **M** 列向量中的 x 分量对其产生影响。因此，计算间隔范围对应于计算最小值、最大值与 **M** 中相应行的乘积。B 中的每一个顶点都是一个组合值，包含 3 个 A 中变换后的最小值或最大值。最小范围值为 min()项之和；而最大范围值为 max()项之和。由于平移操作并不影响新包围盒的尺寸，因而可以加入平移变换。例如，x 轴上的最大范围值计算如下：

```
B.max[0] = max(m[0][0] * A.min[0], m[0][0] * A.max[0])
           + max(m[0][1] * A.min[1], m[0][1] * A.max[1])
           + max(m[0][2] * A.min[2], m[0][2] * A.max[2]) + t[0];
```

因此，对于采用最小值-最大值形式的 AABB，其旋转后的包围盒可以参考下列代码：

```
// Transform AABB a by the matrix m and translation t,
// find maximum extents, and store result into AABB b.
void UpdateAABB(AABB a, float m[3][3], float t[3], AABB &b)
{
    // For all three axes
    for (int i = 0; i < 3; i++) {
        // Start by adding in translation
        b.min[i] = b.max[i] = t[i];
        // Form extent by summing smaller and larger terms respectively
        for (int j = 0; j < 3; j++) {
            float e = m[i][j] * a.min[j];
            float f = m[i][j] * a.max[j];
            if (e < f) {
                b.min[i] += e;
                b.max[i] += f;
            } else {
                b.min[i] += f;
                b.max[i] += e;
            }
        }
    }
}
```

相应地，对于采用中心-半径形式的 AABB，其旋转后的包围盒可以参考下列代码（[Arvo90]）：

```
// Transform AABB a by the matrix m and translation t,
// find maximum extents, and store result into AABB b.
void UpdateAABB(AABB a, float m[3][3], float t[3], AABB &b)
{
    for (int i = 0; i < 3; i++) {
        b.c[i] = t[i];
        b.r[i] = 0.0f;
        for (int j = 0; j < 3; j++) {
            b.c[i] += m[i][j] * a.c[j];
            b.r[i] += Abs(m[i][j]) * a.r[j];
        }
    }
}
```

需注意的是，计算旋转后的 AABB 等价于计算一个有向包围盒（关于有向包围盒及其相交测试，本书后续章节将详细讨论）。其中，主要的差别在于需要存储物体对象的有向包围盒，且仍然对这些重构的 AABB 执行相交测试。因此，需要占用额外的内存用于存储方向矩阵。同时，还将执行相应的矩阵乘法计算，并得到有向包围盒的旋转矩阵与转换矩阵 **M** 的组合矩阵。这种方法的优点是：针对初

始的有向盒体，重构的轴对齐盒体将更加紧凑。同时，对于有向盒体而言，这种轴对齐的测试方式也更加简洁、高效。

4.3 Spheres 球体

与包围盒相比，球体是另一类较为常用的包围体。如同包围盒，球体也具备快速相交测试这一特征。同时，球体基本不受旋转变换的影响，只需简单地平移至一个新位置。球体按照球心和半径的方式加以定义：

```
// Region R = (x, y, z) | (x-c.x)^2 + (y-c.y)^2 + (z-c.z)^2 <= r^2
struct Sphere {
    Point c;    // Sphere center
    float r;    // Sphere radius
};
```

该结构只包含了 4 个分量，因而包围球是一类节省内存的包围体。通常情况下，物体对象的中心或原点与球心基本相符，因此只需存储上述结构中的半径分量。与 AABB 一样，计算一个优化的包围球并非易事。下面几个小节将考查几种计算包围球的相关算法，其中，为了提高精度，还介绍了最小包围球的计算方法。这一类非优化的逼近算法也适用于其他类型的包围体。

4.3.1 其他相交测试

球体间的相交测试简单明了：计算两球体间的几何距离，并与半径和比较。可以采用距离的平方值进行相关计算，从而避免计算成本较高的平方根操作。如下所示：

```
int TestSphereSphere(Sphere a, Sphere b)
{
    // Calculate squared distance between centers
    Vector d = a.c - b.c;
    float dist2 = Dot(d, d);
    // Spheres intersect if squared distance is less than squared sum of radii
    float radiusSum = a.r + b.r;
    return dist2 <= radiusSum * radiusSum;
}
```

与 AABB 相比，虽然球体测试多出了几个算术操作，但包含较少的分支语句以及较少的数据存取。在现代计算机体系结构中，球体测试可能比 AABB 测试稍快一些，但是，对于这类简单测试，在进行算法选择时不能仅将速度作为考量的因素，数据计算的严谨性往往更加重要。

4.3.2 计算包围球

首先，可通过计算所有顶点的 AABB 获取一个逼近的包围球，则 AABB 的中心点将作为球心，而

球体的半径则可以通过计算最远顶点与球心的距离获得。需注意的是，采用所有顶点的几何中心点（平均值）而非 AABB 的中心点，将产生错误的包围球（半径的 2 倍）——这是因为各顶点的权值（贡献值）不同。虽然该算法速度较快，但与优化后的方法相比，其适用性仍非最佳。

一种替代算法可参见[Ritter90]，该算法初期即尝试计算一个近似完整的包围球，并通过后续步骤完善该球体，直至包围全部顶点。该算法分两步执行：第一步，获取各坐标轴上的 6 个端点，可以选择其中间隔距离最远的两个顶点（注意到这两个顶点不必是定义 AABB 最大边的那两个顶点），这两个顶点的中点即为球心，且两顶点距离的一半则为球体半径。针对第一步计算的代码 MostSeparatedPointsOnAABB()和 SphereFromDistantPoints()，代码如下：

```
// Compute indices to the two most separated points of the (up to) six points
// defining the AABB encompassing the point set. Return these as min and max.
void MostSeparatedPointsOnAABB(int &min, int &max, Point pt[], int numPts)
{
    // First find most extreme points along principal axes
    int minx = 0, maxx = 0, miny = 0, maxy = 0, minz = 0, maxz = 0;
    for (int i = 1; i < numPts; i++) {
        if (pt[i].x < pt[minx].x) minx = i;
        if (pt[i].x > pt[maxx].x) maxx = i;
        if (pt[i].y < pt[miny].y) miny = i;
        if (pt[i].y > pt[maxy].y) maxy = i;
        if (pt[i].z < pt[minz].z) minz = i;
        if (pt[i].z > pt[maxz].z) maxz = i;
    }
    // Compute the squared distances for the three pairs of points
    float dist2x = Dot(pt[maxx] - pt[minx], pt[maxx] - pt[minx]);
    float dist2y = Dot(pt[maxy] - pt[miny], pt[maxy] - pt[miny]);
    float dist2z = Dot(pt[maxz] - pt[minz], pt[maxz] - pt[minz]);
    // Pick the pair (min,max) of points most distant
    min = minx;
    max = maxx;
    if (dist2y > dist2x && dist2y > dist2z) {
        max = maxy;
        min = miny;
    }
    if (dist2z > dist2x && dist2z > dist2y) {
        max = maxz;
        min = minz;
    }
}

void SphereFromDistantPoints(Sphere &s, Point pt[], int numPts)
{
    // Find the most separated point pair defining the encompassing AABB
    int min, max;
    MostSeparatedPointsOnAABB(min, max, pt, numPts);
```

```
    // Set up sphere to just encompass these two points
    s.c = (pt[min] + pt[max]) * 0.5f;
    s.r = Dot(pt[max] - s.c, pt[max] - s.c);
    s.r = Sqrt(s.r);
}
```

在第二步中，全部顶点再次循环。对于位于当前球体外部的所有顶点，该步骤将更新为包含原球体及其外部顶点的新球体。即相对于原球心，新球体的直径将延伸至外部顶点。

```
// Given Sphere s and Point p, update s (if needed) to just encompass p
void SphereOfSphereAndPt(Sphere &s, Point &p)
{
    // Compute squared distance between point and sphere center
    Vector d = p - s.c;
    float dist2 = Dot(d, d);
    // Only update s if point p is outside it

    if (dist2 > s.r * s.r) {
        float dist = Sqrt(dist2);
        float newRadius = (s.r + dist) * 0.5f;
        float k = (newRadius - s.r) / dist;
        s.r = newRadius;
        s.c += d * k;
    }
}
```

因此，计算逼近包围球的全部代码如下所示：

```
void RitterSphere(Sphere &s, Point pt[], int numPts)
{
    // Get sphere encompassing two approximately most distant points
    SphereFromDistantPoints(s, pt, numPts);

    // Grow sphere to include all points
    for (int i = 0; i < numPts; i++)
        SphereOfSphereAndPt(s, pt[i]);
}
```

如果前期使用一个较好的逼近球体，则其最终的计算结果将会是一个更加紧凑的球体。如何使用逼近球体将在 4.3.3 节中阐述。

4.3.3 最大离散方向上的包围球

这里不采用 4.3.2 节中介绍的 AABB 中的一对间隔顶点，而是利用统计方法分析顶点云并确定最大离散方向（[Wu92]）。若给定该方向，将距离最远的两顶点投影至此轴上，即可得到初始球体的球心和半径。图 4.8 显示了同一顶点云在两个不同轴上扩散时的区别。

图 4.8　点集在各轴上的投影结果。（a）较小的轴离散度。（b）较大的轴离散度。当投影点集在当前轴上具有最大离散度时，即可确定包围球体

正如数据集的平均值（即数据和除以数据的数量）是数值居中趋势的一种计算方法，方差则是计算数值的离中趋势（或称为离散度）的一种方法。均值 u 和方差 σ^2 的计算方法如下所示：

$$u = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - u)^2 = \frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right) - u^2$$

方差的平方根记为标准偏差。对于沿某一轴上的离散值，方差可通过下列方式计算：首先得到数据值的平均值 u，然后求得标准偏差的平方值，最后得到其平均值即方差值，如下列代码所示：

```
// Compute variance of a set of 1D values
float Variance(float x[], int n)
{
    float u = 0.0f;
    for (int i = 0; i < n; i++)
        u += x[i];
    u /= n;
    float s2 = 0.0f;
    for (int i = 0; i < n; i++)
        s2 += (x[i] - u) * (x[i] - u);
    return s2 / n;
}
```

通常很少对方差和标准偏差进行直观的解释，但它们是一种重要的比较计算法。对于两个变量，方差将通过下列方式计算其协同变化的方式：均值数据标准偏差平方的平均值。对于多个变量，数据的方差一般采用矩阵计算，即协方差矩阵。

对于包含 n 个顶点 $P_1, P_2, \cdots, P_n$ 的点集，其协方差矩阵 $\mathbf{C} = [c_{ij}]$ 为：

$$c_{ij} = \frac{1}{n}\sum_{k=1}^{n}(P_{k,i} - u_i)(P_{k,j} - u_j)$$

或

$$c_{ij} = \frac{1}{n}\left(\sum_{k=1}^{n} P_{k,i}P_{k,j}\right) - u_i u_j$$

其中，u_i（及 u_j）项为顶点第 i 个坐标值的均值，如下所示：

$$u_i = \frac{1}{n}\sum_{k=1}^{n} P_{k,i}$$

这里考查上述第一个协方差矩阵，并讨论其运作方式。当两个变量距各自均值发生同向偏离时，下列乘积值趋向于正值：

$$(P_{k,i} - u_i)(P_{k,j} - u_j)$$

否则，若发生异向偏离，则乘积值趋于负值。乘积值的和则确定了变量间协同变化的方式。当采用单精度浮点数时，上述公式通过保留更多的精度位可以获得满意的精确度；若采用双精度浮点数，其计算结果与单精度浮点数相比基本上没有变化。上述第一个协方差矩阵的代码实现如下所示：

```
void CovarianceMatrix(Matrix33 &cov, Point pt[], int numPts)
{
    float oon = 1.0f / (float)numPts;
    Point c = Point(0.0f, 0.0f, 0.0f);
    float e00, e11, e22, e01, e02, e12;

    // Compute the center of mass (centroid) of the points
    for (int i = 0; i < numPts; i++)


        c += pt[i];
    c *= oon;

    // Compute covariance elements
    e00 = e11 = e22 = e01 = e02 = e12 = 0.0f;
    for (int i = 0; i < numPts; i++) {
        // Translate points so center of mass is at origin
        Point p = pt[i] - c;
        // Compute covariance of translated points
        e00 += p.x * p.x;
        e11 += p.y * p.y;
        e22 += p.z * p.z;
        e01 += p.x * p.y;
        e02 += p.x * p.z;
        e12 += p.y * p.z;
    }
    // Fill in the covariance matrix elements
    cov[0][0] = e00 * oon;
    cov[1][1] = e11 * oon;
    cov[2][2] = e22 * oon;
    cov[0][1] = cov[1][0] = e01 * oon;
    cov[0][2] = cov[2][0] = e02 * oon;
    cov[1][2] = cov[2][1] = e12 * oon;
}
```

获取协方差矩阵可以更好地解释方差的工作原理，通过计算该矩阵的特征值和特征向量，可以实现对矩阵的分解。对于特征值和特征向量，二者的关系为：若特征向量为最大特征向量，则特征值即为沿该轴的顶点数据具有的最大方差值。类似地，若为最小特征向量，则特征值为沿该轴的顶点数据

具有的最小方差值。通常，计算矩阵健壮的特征值和特征向量并非易事，一般采取某种（迭代）数值计算技术（可参见[Golub96]）。

通过上述定义可知，协方差矩阵通常都是对称的，并分解为一个实数（而非复数）特征值以及一个特征向量表示的正交基。对于常规尺寸矩阵，Jacobi 方法可以工作得很好，但对于其具体描述则超出了本书的内容范围。简而言之，该算法对输入矩阵将执行一系列的变换，每一步都包含了对相关矩阵的旋转变换，并使结果矩阵逐步逼近对角矩阵（除了对角线上的元素，其他元素皆为 0）。对于最终的对角矩阵，对角线上的元素即为特征值；多个旋转矩阵将连接为一个结果矩阵，其中包含了相应的特征向量。理想情况下，该分解过程宜采用双精度浮点数，从而减少数值误差。下列与 Jacobi 方法相关的代码基于[Golub96]，其中第一段代码用于辅助计算旋转矩阵。

```
// 2-by-2 Symmetric Schur decomposition. Given an n-by-n symmetric matrix
// and indices p, q such that 1 <= p < q <= n, computes a sine-cosine pair
// (s, c) that will serve to form a Jacobi rotation matrix.
//
// See Golub, Van Loan, Matrix Computations, 3rd ed, p428
void SymSchur2(Matrix33 &a, int p, int q, float &c, float &s)
{
    if (Abs(a[p][q]) > 0.0001f) {
        float r = (a[q][q] - a[p][p]) / (2.0f * a[p][q]);
        float t;
        if (r >= 0.0f)
            t = 1.0f / (r + Sqrt(1.0f + r*r));
        else
            t = -1.0f / (-r + Sqrt(1.0f + r*r));
        c = 1.0f / Sqrt(1.0f + t*t);
        s = t * c;
    } else {
        c = 1.0f;
        s = 0.0f;
    }
}
```

实现了上述辅助函数后，Jacobi 方法的相关实现如下所示：

```
// Computes the eigenvectors and eigenvalues of the symmetric matrix A using
// the classic Jacobi method of iteratively updating A as A = J^T * A * J,
// where J = J(p, q, theta) is the Jacobi rotation matrix.
//
// On exit, v will contain the eigenvectors, and the diagonal elements
// of a are the corresponding eigenvalues.
//
// See Golub, Van Loan, Matrix Computations, 3rd ed, p428
void Jacobi(Matrix33 &a, Matrix33 &v)
{
    int i, j, n, p, q;
    float prevoff, c, s;
    Matrix33 J, b, t;
```

```
// Initialize v to identify matrix
for (i = 0; i < 3; i++) {
        v[i][0] = v[i][1] = v[i][2] = 0.0f;
        v[i][i] = 1.0f;
}

// Repeat for some maximum number of iterations
const int MAX_ITERATIONS = 50;
for (n = 0; n < MAX_ITERATIONS; n++) {
        // Find largest off-diagonal absolute element a[p][q]
        p = 0; q = 1;
        for (i = 0; i < 3; i++) {
                for (j = 0; j < 3; j++) {
                        if (i == j) continue;
                        if (Abs(a[i][j]) > Abs(a[p][q])) {
                                p = i;
                                q = j;
                        }
                }
}

// Compute the Jacobi rotation matrix J(p, q, theta)
// (This code can be optimized for the three different cases of rotation)
SymSchur2(a, p, q, c, s);
for (i = 0; i < 3; i++) {
        J[i][0] = J[i][1] = J[i][2] = 0.0f;
        J[i][i] = 1.0f;
}
J[p][p] = c; J[p][q] = s;
J[q][p] = -s; J[q][q] = c;

// Cumulate rotations into what will contain the eigenvectors
v = v * J;

// Make 'a'more diagonal, until just eigenvalues remain on diagonal
a = (J.Transpose() * a) * J;

// Compute "norm" of off-diagonal elements
float off = 0.0f;
for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
                if (i == j) continue;
                off += a[i][j] * a[i][j];
        }
}
/* off = sqrt(off); not needed for norm comparison */

// Stop when norm no longer decreasing
if (n > 2 && off >= prevoff)
```

```
        return;

    prevoff = off;
}
```

对于这里使用的 3×3 矩阵，可以不采用 Jacobi 方法，特征值可以通过一个简单的 3 次方程直接求解，特征向量则利用高斯消元法计算得到。该算法可参见[Cromwell94]。有了上面定义的函数，通过最远两点计算球体的函数如下所示：

```
void EigenSphere(Sphere &eigSphere, Point pt[], int numPts)
{
    Matrix33 m, v;

    // Compute the covariance matrix m
    CovarianceMatrix(m, pt, numPts);
    // Decompose it into eigenvectors (in v) and eigenvalues (in m)
    Jacobi(m, v);

    // Find the component with largest magnitude eigenvalue (largest spread)
    Vector e;
    int maxc = 0;
    float maxf, maxe = Abs(m[0][0]);
    if ((maxf = Abs(m[1][1])) > maxe) maxc = 1, maxe = maxf;
    if ((maxf = Abs(m[2][2])) > maxe) maxc = 2, maxe = maxf;
    e[0] = v[0][maxc];
    e[1] = v[1][maxc];
    e[2] = v[2][maxc];

    // Find the most extreme points along direction 'e'
    int imin, imax;
    ExtremePointsAlongDirection(e, pt, numPts, &imin, &imax);
    Point minpt = pt[imin];
    Point maxpt = pt[imax];

    float dist = Sqrt(Dot(maxpt - minpt, maxpt - minpt));
    eigSphere.r = dist * 0.5f;
    eigSphere.c = (minpt + maxpt) * 0.5f;
}
```

改进后的逼近包围球代码如下所示：

```
void RitterEigenSphere(Sphere &s, Point pt[], int numPts)
{
    // Start with sphere from maximum spread
    EigenSphere(s, pt, numPts);

    // Grow sphere to include all points
    for (int i = 0; i < numPts; i++)
    SphereOfSphereAndPt(s, pt[i]);
}
```

这里所讨论的协方差分析一般用于降维计算和数据的统计分析，即“主成分分析（PCA）”。关于 PCA 的更多信息可参考[Jolliffe02]。协方差矩阵的特征向量也可用于定向一个方向包围盒，相关内容将在 4.4.3 节中加以描述。

4.3.4 采用迭代修正的包围球

对于 4.3.2 节中所描述的算法，其主要思想是：针对一个最小真实球体采取某种较为完美的逼近，而后对其扩展直至包含全部顶点。可以很自然地想到，算法的输出值可以以一种迭代的方式再次作为输入值。一次迭代后的球体将缩小一定量且再次减少其半径以供下一次调用。

```
void RitterIterative(Sphere &s, Point pt[], int numPts)
{
    const int NUM_ITER = 8;
    RitterSphere(s, pt, numPts);
    Sphere s2 = s;
    for (int k = 0; k < NUM_ITER; k++) {
        // Shrink sphere somewhat to make it an underestimate (not bound)
        s2.r = s2.r * 0.95f;

        // Make sphere bound data again
        for (int i = 0; i < numPts; i++) {
            // Swap pt[i] with pt[j], where j randomly from interval [i+1,numPts-1]
            DoRandomSwap();
            SphereOfSphereAndPt(s2, pt[i]);
        }

        // Update s whenever a tighter sphere is found
        if (s2.r < s.r) s = s2;
    }
}
```

为了进一步改进上述结果，可以考查随机顶点，而非迭代过程中的固定顺序顶点。通常，该算法要优于 Wu 方法（参见前述小节）生成的结果，并且其消耗仅体现在对输入数据的迭代计算。如果相同的迭代方案应用于 Wu 算法中，二者的计算结果相当。而对于同一类型的迭代爬山算法（如梯度下降法、模拟退火法或禁忌搜索法），搜索结果可能阻塞于局部最小值，且不能保证为最优结果。然而最终返回结果通常接近最优且能满足相应的健壮性要求。

4.3.5 最小包围球

球体可以通过 4 个（非共面）顶点唯一确定。因此，可以采用蛮力法计算顶点集的最小包围球，即考查 4 个顶点的全部可能性组合（依次为 3 个顶点的可能性组合，2 个顶点的可能性组合），且能够得到当前组合测试中的最小包围球并加以记录。在记录的最小包围球集合中，半径最小的球体即为最终结果。该蛮力算法的时间复杂度为 $O(n^5)$，因此难以在实际计算过程中使用。但这一相关问题已经在

计算几何领域得到了很好的解决，一种线性时间相关的随机算法已经在[Welzl91]中给出了相应的答案。

假定有顶点集 P，现计算其最小包围球 S。如果一个新顶点 Q 加入集合 P 中，仅当 Q 位于 S 之外，球体 S 才存在重计算的必要。不难看出，对于顶点集 $P \cup \{Q\}$，顶点 Q 一定位于新增最小包围球的边界之上。Welzl 算法便是基于上述观察结果而形成的一种递归算法。该算法维护两个顶点集——输入顶点集和支撑顶点集，后者包含了输入顶点集中位于最小球体边界上的顶点。下列代码片段显示了 Welzl 主算法：

```
Sphere WelzlSphere(Point pt[], unsigned int numPts, Point sos[], unsigned int numSos)
{
    // if no input points, the recursion has bottomed out. Now compute an
    // exact sphere based on points in set of support (zero through four points)
    if (numPts == 0) {
        switch (numSos) {
        case 0: return Sphere();
        case 1: return Sphere(sos[0]);
        case 2: return Sphere(sos[0], sos[1]);
        case 3: return Sphere(sos[0], sos[1], sos[2]);
        case 4: return Sphere(sos[0], sos[1], sos[2], sos[3]);
        }
    }
    // Pick a point at "random" (here just the last point of the input set)
    int index = numPts - 1;
    // Recursively compute the smallest bounding sphere of the remaining points
    Sphere smallestSphere = WelzlSphere(pt, numPts - 1, sos, numSos);    // (*)
    // If the selected point lies inside this sphere, it is indeed the smallest
    if(PointInsideSphere(pt[index], smallestSphere))
        return smallestSphere;
    // Otherwise, update set of support to additionally contain the new point
    sos[numSos] = pt[index];
    // Recursively compute the smallest sphere of remaining points with new s.o.s.
    return WelzlSphere(pt, numPts - 1, sos, numSos + 1);
}
```

虽然函数中的两个递归调用似乎使算法付出了额外的代价，但该算法表明，假设的顶点将随机地从输入集中删除，从而保证算法的时间消耗为线性的。需注意的是，如代码所示，当输入超过几千个顶点时，第一个递归调用（代码中带有*号的注释语句）有可能导致栈的上溢，而对该代码稍加修改即可避免这一问题，参见[Gärtner99]。该算法在[Capens01]中给出了其全部实现过程。另外，也可从网络上获取该算法的复杂实现，可参见“计算几何算法库（CGAL）”中的相关内容。实现健壮的 Welzl 算法，需要上述 4 个支持函数（分别对应 1～4 个顶点时的真实球体计算）能够正确处理输入顶点集的退化现象，例如顶点共面。

Welzl 算法可应用于包围圆以及更高维度的球体计算。然而，该算法并不直接计算包含一个球体集的最小球体。关于第 2 个问题可参见[Fischer03]。在详细讨论了球体问题之后，下面将计算任意方向的包围盒。

4.4 方向包围盒

方向包围盒（OBB）是一个长方体，类似于 AABB，但具有方向性。存在多种 OBB 的表达方式：8 个顶点的点集、6 个面的面集、3 组平行面集合、一个顶点和 3 个彼此正交的边向量，以及中心点、一个旋转矩阵和 3 个 1/2 边长。通常，最后一种表达方式最为常用，并且与其他表达方式相比，其 OBB 间相交测试更加便捷。该测试算法基于分离轴理论，相关内容将在第 5 章加以讨论。下列代码给出了 OOB 的数据结构：

```
// Region R = x | x = c+r*u[0]+s*u[1]+t*u[2], |r|<=e[0], |s|<=e[1], |t|<=e[2]
struct OBB {
    Point c;        // OBB center point
    Vector u[3];    // Local x-, y-, and z-axes
    Vector e;       // Positive halfwidth extents of OBB along each axis
};
```

对于 IEEE 标准的单精度浮点数，上述结构包含了 15 个浮点数（即 60 个字节），对于内存消耗来讲，OBB 无疑是包围体中较大的一类。可以直接存储欧拉角或四元数而非矩阵作为方位数据，从而达到降低内存消耗的目的，这就只利用了 3～4 个浮点数而非 9 个。但是，对于 OBB 相交测试而言，在分离轴测试中，上述存储方式最终还需要转换回矩阵方式，这将是非常耗时的操作。一种较好的妥协方案是：只存储旋转矩阵中的两个轴，只是在测试时利用叉积计算第 3 个轴。相对来讲，这能够降低 CPU 的操作开销，同时还可以节省 3 个浮点数分量，即降低了 20%的内存消耗。

4.4.1 相交测试

不同于前述的包围体相交测试，两个方向包围盒之间的相交测试则十分复杂。首先，该测试可看作：某一个盒体是否完全位于另一个盒体的某一个面外。这一方式可转化为：盒体 A 的顶点是否完全位于盒体 B 的面外，反之亦然。该算法在 2D 空间内运行正常，却无法在 3D 空间内正确地执行：因为算法无法正确处理 A、B 之间的边-边碰撞。这里，盒体可能并非位于另一个盒体面的同一侧，上述算法也仍视为相交，即使两个盒体并未相交。尽管可能会出现这种错误情况，上述算法也并非毫无意义：这虽然是一个保守的测试算法，但不会遗漏相应的碰撞结果——可能包含了冗余的碰撞测试结果。在某些情况下，该算法确实会将分离的盒体错误地报告为相交。鉴于此，该算法可以作为某些复杂测试的预测试算法。

OBB 相交测试可以采用“分离轴测试”加以实现，该测试方法将在第 5 章加以讨论。这里需要注意的是，对于某一轴 L，如果两盒体投影半径之和小于中心点之间的投影距离，则 OBB 处于分离状态，如图 4.9 所示。且有：

$$|T \cdot L| > r_A + r_B$$

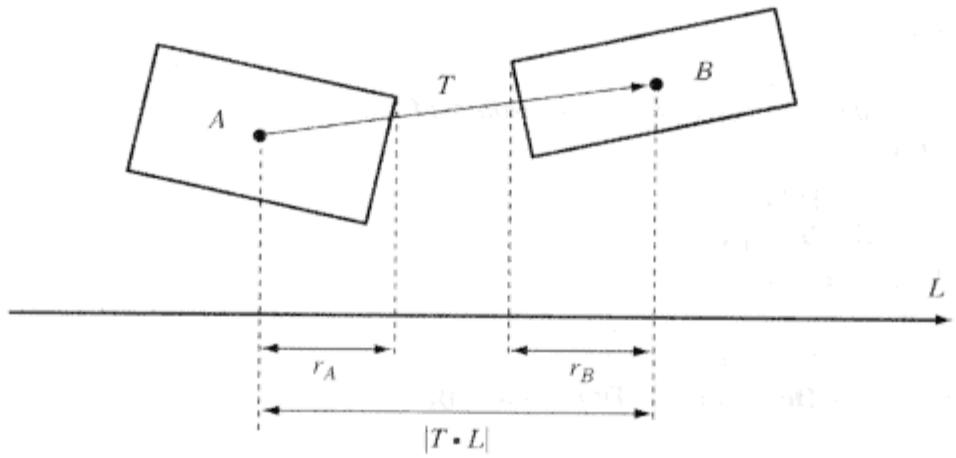


图 4.9　针对某一轴 L，若两个 OBB 的投影半径和小于投影中心间的距离，则二者处于分离状态

对于 OBB，需要测试最多 15 个分离轴以确定 OBB 的相交状态。这些分离轴分别对应着 A 的 3 个坐标轴、B 的 3 个坐标轴以及垂直于每一轴的 9 个轴。若盒体间在上述 15 个轴上的任一轴上都未产生重叠，则两 OBB 不相交；否则即是在某一轴上产生了重叠，且测试提前退出，从而确定盒体间相交。

可以将 B 转换至 A 的坐标系统中，从而减少操作数量。若 $\mathbf{t}$ 为由 A 至 B 的位移向量，$\mathbf{R}$ 为$[r_{ij}]$（将 B 转换至 A 的坐标系统中的旋转矩阵），则在各轴 L 上的测试过程如表 4.1 所示。

表 4.1　15 个分离轴测试以确定 OBB-OBB 间的相交状态（上角标表示相应的 OBB）

L	$\lvert T \cdot L \rvert$	r_A	r_B
$\mathbf{u}_0^A$	$\lvert \mathbf{t}_0 \rvert$	$\mathbf{e}_0^A$	$\mathbf{e}_0^B \lvert r_{00} \rvert + \mathbf{e}_1^B \lvert r_{01} \rvert + \mathbf{e}_2^B \lvert r_{02} \rvert$
$\mathbf{u}_1^A$	$\lvert \mathbf{t}_1 \rvert$	$\mathbf{e}_1^A$	$\mathbf{e}_0^B \lvert r_{10} \rvert + \mathbf{e}_1^B \lvert r_{11} \rvert + \mathbf{e}_2^B \lvert r_{12} \rvert$
$\mathbf{u}_2^A$	$\lvert \mathbf{t}_2 \rvert$	$\mathbf{e}_2^A$	$\mathbf{e}_0^B \lvert r_{20} \rvert + \mathbf{e}_1^B \lvert r_{21} \rvert + \mathbf{e}_2^B \lvert r_{22} \rvert$
$\mathbf{u}_0^B$	$\lvert \mathbf{t}_0 r_{00} + \mathbf{t}_1 r_{10} + \mathbf{t}_2 r_{20} \rvert$	$\mathbf{e}_0^A \lvert r_{00} \rvert + \mathbf{e}_1^A \lvert r_{10} \rvert + \mathbf{e}_2^A \lvert r_{20} \rvert$	$\mathbf{e}_0^B$
$\mathbf{u}_1^B$	$\lvert \mathbf{t}_0 r_{01} + \mathbf{t}_1 r_{11} + \mathbf{t}_2 r_{21} \rvert$	$\mathbf{e}_0^A \lvert r_{01} \rvert + \mathbf{e}_1^A \lvert r_{11} \rvert + \mathbf{e}_2^A \lvert r_{21} \rvert$	$\mathbf{e}_1^B$
$\mathbf{u}_2^B$	$\lvert \mathbf{t}_0 r_{02} + \mathbf{t}_1 r_{12} + \mathbf{t}_2 r_{22} \rvert$	$\mathbf{e}_0^A \lvert r_{02} \rvert + \mathbf{e}_1^A \lvert r_{12} \rvert + \mathbf{e}_2^A \lvert r_{22} \rvert$	$\mathbf{e}_2^B$
$\mathbf{u}_0^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_2 r_{10} - \mathbf{t}_1 r_{20} \rvert$	$\mathbf{e}_1^A \lvert r_{20} \rvert + \mathbf{e}_2^A \lvert r_{10} \rvert$	$\mathbf{e}_1^B \lvert r_{02} \rvert + \mathbf{e}_2^B \lvert r_{01} \rvert$
$\mathbf{u}_0^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_2 r_{11} - \mathbf{t}_1 r_{21} \rvert$	$\mathbf{e}_1^A \lvert r_{21} \rvert + \mathbf{e}_2^A \lvert r_{11} \rvert$	$\mathbf{e}_0^B \lvert r_{02} \rvert + \mathbf{e}_2^B \lvert r_{00} \rvert$
$\mathbf{u}_0^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_2 r_{12} - \mathbf{t}_1 r_{22} \rvert$	$\mathbf{e}_1^A \lvert r_{22} \rvert + \mathbf{e}_2^A \lvert r_{12} \rvert$	$\mathbf{e}_0^B \lvert r_{01} \rvert + \mathbf{e}_1^B \lvert r_{00} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_0 r_{20} - \mathbf{t}_2 r_{00} \rvert$	$\mathbf{e}_0^A \lvert r_{20} \rvert + \mathbf{e}_2^A \lvert r_{00} \rvert$	$\mathbf{e}_1^B \lvert r_{12} \rvert + \mathbf{e}_2^B \lvert r_{11} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_0 r_{21} - \mathbf{t}_2 r_{01} \rvert$	$\mathbf{e}_0^A \lvert r_{21} \rvert + \mathbf{e}_2^A \lvert r_{01} \rvert$	$\mathbf{e}_0^B \lvert r_{12} \rvert + \mathbf{e}_2^B \lvert r_{10} \rvert$
$\mathbf{u}_1^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_0 r_{22} - \mathbf{t}_2 r_{02} \rvert$	$\mathbf{e}_0^A \lvert r_{22} \rvert + \mathbf{e}_2^A \lvert r_{02} \rvert$	$\mathbf{e}_0^B \lvert r_{11} \rvert + \mathbf{e}_1^B \lvert r_{10} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_0^B$	$\lvert \mathbf{t}_1 r_{00} - \mathbf{t}_0 r_{10} \rvert$	$\mathbf{e}_0^A \lvert r_{10} \rvert + \mathbf{e}_1^A \lvert r_{00} \rvert$	$\mathbf{e}_1^B \lvert r_{22} \rvert + \mathbf{e}_2^B \lvert r_{21} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_1^B$	$\lvert \mathbf{t}_1 r_{01} - \mathbf{t}_0 r_{11} \rvert$	$\mathbf{e}_0^A \lvert r_{11} \rvert + \mathbf{e}_1^A \lvert r_{01} \rvert$	$\mathbf{e}_0^B \lvert r_{22} \rvert + \mathbf{e}_2^B \lvert r_{20} \rvert$
$\mathbf{u}_2^A \times \mathbf{u}_2^B$	$\lvert \mathbf{t}_1 r_{02} - \mathbf{t}_0 r_{12} \rvert$	$\mathbf{e}_0^A \lvert r_{12} \rvert + \mathbf{e}_1^A \lvert r_{02} \rvert$	$\mathbf{e}_0^B \lvert r_{21} \rvert + \mathbf{e}_1^B \lvert r_{20} \rvert$

该测试的实现方法如下列代码所示：

```
int TestOBBOBB(OBB &a, OBB &b)
{
    float ra, rb;
```

```
Matrix33 R, AbsR;

// Compute rotation matrix expressing b in a's coordinate frame
for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
        R[i][j] = Dot(a.u[i], b.u[j]);
// Compute translation vector t
Vector t = b.c - a.c;
// Bring translation into a's coordinate frame
t = Vector(Dot(t, a.u[0]), Dot(t, a.u[2]), Dot(t, a.u[2]));

// Compute common subexpressions. Add in an epsilon term to
// counteract arithmetic errors when two edges are parallel and
// their cross product is (near) null (see text for details)
for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
                AbsR[i][j] = Abs(R[i][j]) + EPSILON;

// Test axes L = A0, L = A1, L = A2
for (int i = 0; i < 3; i++) {
        ra = a.e[i];
        rb = b.e[0] * AbsR[i][0] + b.e[1] * AbsR[i][1] + b.e[2] * AbsR[i][2];
        if (Abs(t[i]) > ra + rb) return 0;
}

// Test axes L = B0, L = B1, L = B2
for (int i = 0; i < 3; i++) {
        ra = a.e[0] * AbsR[0][i] + a.e[1] * AbsR[1][i] + a.e[2] * AbsR[2][i];
        rb = b.e[i];
        if (Abs(t[0] * R[0][i] + t[1] * R[1][i] + t[2] * R[2][i]) > ra + rb) return 0;
}

// Test axis L = A0 × B0
ra = a.e[1] * AbsR[2][0] + a.e[2] * AbsR[1][0];
rb = b.e[1] * AbsR[0][2] + b.e[2] * AbsR[0][1];
if (Abs(t[2] * R[1][0] - t[1] * R[2][0]) > ra + rb) return 0;

// Test axis L = A0 × B1
ra = a.e[1] * AbsR[2][1] + a.e[2] * AbsR[1][1];
rb = b.e[0] * AbsR[0][2] + b.e[2] * AbsR[0][0];
if (Abs(t[2] * R[1][1] - t[1] * R[2][1]) > ra + rb) return 0;

// Test axis L = A0 × B2
ra = a.e[1] * AbsR[2][2] + a.e[2] * AbsR[1][2];
rb = b.e[0] * AbsR[0][1] + b.e[1] * AbsR[0][0];
if (Abs(t[2] * R[1][2] - t[1] * R[2][2]) > ra + rb) return 0;

// Test axis L = A1 × B0
ra = a.e[0] * AbsR[2][0] + a.e[2] * AbsR[0][0];
```

```
    rb = b.e[1] * AbsR[1][2] + b.e[2] * AbsR[1][1];
    if (Abs(t[0] * R[2][0] - t[2] * R[0][0]) > ra + rb) return 0;

    // Test axis L = A1 × B1
    ra = a.e[0] * AbsR[2][1] + a.e[2] * AbsR[0][1];
    rb = b.e[0] * AbsR[1][2] + b.e[2] * AbsR[1][0];
    if (Abs(t[0] * R[2][1] - t[2] * R[0][1]) > ra + rb) return 0;

    // Test axis L = A1 × B2
    ra = a.e[0] * AbsR[2][2] + a.e[2] * AbsR[0][2];
    rb = b.e[0] * AbsR[1][1] + b.e[1] * AbsR[1][0];
    if (Abs(t[0] * R[2][2] - t[2] * R[0][2]) > ra + rb) return 0;

    // Test axis L = A2 × B0
    ra = a.e[0] * AbsR[1][0] + a.e[1] * AbsR[0][0];
    rb = b.e[1] * AbsR[2][2] + b.e[2] * AbsR[2][1];
    if (Abs(t[1] * R[0][0] - t[0] * R[1][0]) > ra + rb) return 0;

    // Test axis L = A2 × B1
    ra = a.e[0] * AbsR[1][1] + a.e[1] * AbsR[0][1];
    rb = b.e[0] * AbsR[2][2] + b.e[2] * AbsR[2][0];
    if (Abs(t[1] * R[0][1] - t[0] * R[1][1]) > ra + rb) return 0;

    // Test axis L = A2 × B2
    ra = a.e[0] * AbsR[1][2] + a.e[1] * AbsR[0][2];
    rb = b.e[0] * AbsR[2][1] + b.e[1] * AbsR[2][0];
    if (Abs(t[1] * R[0][2] - t[0] * R[1][2]) > ra + rb) return 0;

    // Since no separating axis is found, the OBBs must be intersecting
    return 1;
}
```

为了使 OBB 间的测试更加高效，需要注意的是，各轴的测试顺序按表 4.1 中的内容执行。首先需要执行 3 个正交轴的测试，从而消除空间冗余并快速覆盖全部空间。其次，根据前述规定，*A* 位于原点且与自身的坐标轴系统对齐，则与测试 *B* 中的各轴相比，测试 *A* 中的各轴将使计算消耗减低约一半。虽然此处并未采取该方案，但针对前 3 个测试，仍需交替计算 **R** 和 **AbsR**，当 OBB 测试在前几个 *if* 语句处退出时，无须执行全部计算。

在某些应用场合下，如果 OBB 只有一个轴与当前场景世界实现向上对齐（如物体对象在地面上漫游），则需要单独考查这些“垂直对齐”的 OBB。这一简化过程可以快速实现相交测试，即只测试 4 个分离轴并执行一次垂直方向的快速测试。

有时，只需测试 15 个分离轴中的前 6 项即可迅速得到全部测试结果。结合以往的经验，[Bergen97] 指出：在 OBB 相交测试代码中，最后 9 项测试在确定非相交结果时约占据全部几率的 15%。也许半数查询计算都是正确的，但忽略最后 9 项查询却会带来 6%～7%的误报率。因此，对于边界几何体的精确测试而言，OBB 测试可以作为一种预测试计算方案，虽然这是一种保守测试算法，但不会遗漏每一次碰撞的发生。

4.4.2 健壮的分离轴测试

在某些流行的分离轴解决方案中，测试的健壮性这一重要问题常被忽略。为了按照既定的工作流程得到相应的结果，需要精心地安排其测试的代码实现。当利用各自包围盒上的边计算叉积并构造分离轴时，两条边有可能平行。其结果：叉积的值为 0 向量，而在 0 向量上的投影结果也为 0，则有 0 > 0。从严格的数学意义上讲，该表达式一般不正确。但在具体实现中，代码需要利用浮点运算来处理这类误差。

对于前述经过优化的不等式，平行边对应着旋转矩阵 **R** 中的 0 元素。理论上讲，这仍然能够产生不等式 0 > 0。在实际操作中，由于存在误差累积，旋转矩阵无法实现完全正交，其中的 0 元素也很难保证恰好为 0。因此，不等式两边不会为 0——虽然仍然存在少量误差。由于误差累积可能导致不等式一边符号位的改变，或者产生某一偏移量，因此，其结果存在某种随机性。如果不等式未曾采取相应的防范措施，上述运算误差仍可错误地导致分离轴为（近似）0 向量：两个重叠的 OBB 因而误报为分离状态。

当两个 OBB 彼此相交时，可知不等式右边的值较大，一种简单的解决方法是：获取不等式右侧矩阵元素的绝对值，并增加一个较小的 ε 值。此时对于接近 0 的数值项，ε 成为主控项，并且与（近）平行边对应的轴测试将归于前述描述的保守算法。对于其他非 0 的分离轴测试，该 ε 值可简单地忽略不计。需注意的是，由于旋转矩阵中的分量绝对值位于[0，1]之间，采用固定的 ε 值可保证上述测试工作良好，而无须考虑测试盒体的尺寸。关于分离轴的健壮性问题还将在第 5 章中加以讨论。

4.4.3 计算紧凑的 OBB

计算紧密拟合的方向包围盒具有一定的难度，对于对齐性较差的 OBB 和对齐性良好的 OBB，二者存在着较大的差异，这将使计算过程变得更加复杂（如图 4.10 所示）。[O'Rourke85]给出了计算多面体最小包围盒的相关算法，该算法暗含的核心思想源于：对于给定多面体的一个面和一条边抑或 3 条边，它们将位于多面体包围盒上的不同面中。因此，可对相关的边、面进行系统的搜索，该算法的复杂度为 $O(n^3)$。虽然此算法是一项值得关注的理论结果，但过于复杂且执行速度慢，没有太多的应用价值。

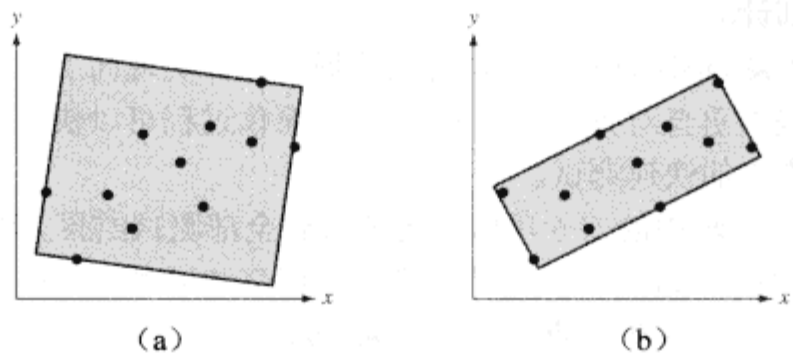


图 4.10 （a）非对齐 OBB。（b）具有良好对齐特征的 OBB

关于计算拥有最小体空间的包围盒逼近算法，[Barequet99]给出了两种理论上的解决方案。但算法

的实现者也承认，该算法难于实现且不具备现实意义。因此，基于当前理论算法的不可应用性，只能采取逼近算法或蛮力算法。

[Barequet99]给出了一种简化算法并对点集实现了最佳 OBB 的粗略逼近：首先计算点集的最小 AABB。其中，选择包围盒中两个间距最大的平行面上的两个点，用以确定 OBB 的长度方向。于是，点集投影至垂直于 OBB 长度方向的平面上。采用相同的方法计算最小的轴对齐矩形，且利用其中两个间距最大的平行面上的顶点计算 OBB 的第 2 个轴。OBB 的第 3 个轴则正交于前两个轴。虽然该算法易于编码，但实际应用中，常采用具有类似复杂度的其他算法获取（接近）最优的包围盒，这一类算法将在后文加以描述。

对于瘦长类型的物体对象，OBB 轴应与物体的方向对齐；而对于扁平的物体对象，OBB 轴应与扁平物体的法线方向对齐。上述方向对应于物体的主方向，而且 4.3.3 节中描述的主成分分析也可以应用于此处。

对于在空间内顶点均匀分布的模型来说，基于模型顶点的包围盒计算一般能够正常运行。然而，内部顶点产生的影响往往使协方差计算产生偏移，并使得 OBB 呈现任意方向——即使存在极值顶点。因此，应避免使用基于加权顶点位置的包围体计算方法。足以引起注意的一点是：最小包围体的特征定义（中心位置、维度以及方向）皆独立于物体顶点的点簇（clustering）。这一点可以通过对包围体添加（或去除）额外的内部或边界偏心顶点得到验证，该方法并不影响包围体的特征定义，因此也不会影响其计算结果。然而，以上述方式增加额外顶点却会改变顶点的协方差矩阵，从而改变了依据此矩阵计算出的 OBB 特征。这种情况可以通过只考查极值点加以改善，即只考虑使用物体对象模型上的凸顶点，从而消除内部顶点导致的 OBB 非对齐问题。然而，即使全部剩余顶点皆为极值顶点，基于顶点的分布状态，最终的 OBB 仍可出现不可预料的错误，其结果为：点簇仍使得指向自身的某一轴产生偏移。从另一个角度来看，即孤立地对顶点实施计算将无法生产稳定、可靠的协方差矩阵。

一种可行的解决方案是，考查图元中的全部面并计算相应的协方差矩阵（[Gottschalk00]）。其中，凸体仍应用于相关计算过程中，否则，某些无关的小型几何元素将会扩展当前包围盒，却不会对盒体的对齐计算产生任何价值；另外，内部的几何元素还会导致协方差偏差。如果凸体为当前已知，则该算法的复杂度为 $O(n)$；考虑到计算凸体的额外代价，该算法的复杂度为 $O(n \log n)$。

给定凸体中的 n 个三角形（p_k，q_k，r_k），其中 $0 \leqslant k < n$，则协方差矩阵为：

$$c_{ij} = \left(\frac{1}{a_H} \sum_{0 \leqslant k < n} \frac{a_k}{12} (9 m_{k,i} m_{k,j} + p_{k,i} p_{k,j} + q_{k,i} q_{k,j} + r_{k,i} r_{k,j} \right) - m_{H,i} m_{H,j}$$

其中，三角形 k 的面积 $a_k = \|(q_k - p_k) \times (r_k - p_k)\|/2$；三角形 k 的质心 $m_k = (p_k + q_k + r_k)/3$。

凸体的全面积及其质心分别为：

$$a_H = \sum_{0 \leqslant k < n} a_k$$

$$m_H = \frac{1}{a_H} \sum_{0 \leqslant k < n} a_k m_k$$

其中，凸体的质心为三角形面积加权质心的均值；下标 i 和 j 表明为所采用的坐标分量（即 x、y 或 z）。该代码可以从公共碰撞检测包 RAPID 中得到；[Eberly01]中也给出了该协方差矩阵的另一个类似版本，相应的代码可以在 www.tup.com.cn 上找到。

上述方法总是将相应的多面体视为空心体，并利用表面积计算其协方差矩阵。在[Mirtich96a]中阐

述了另一类算法，并将多面体视为实体。设定一个密度均匀的多面体，Mirtich 多面体质量特征算法对多面体实现了整合，并计算其 3×3 惯性张量矩阵（即惯性矩阵或质量矩阵）。该对称矩阵的特征向量称为惯性主轴，且特征值称为主转动惯量。正如前述讨论的协方差矩阵，Jacobi 方法可以用于计算各主轴并作为 OBB 的方向矩阵。关于计算惯性矩阵的详细伪代码，可参考 Mirtich 的相关论文；也可以通过网络下载一个共享的 C 语言版本的实现代码。另外，[Eberly03]中也再次提到了 Mirtich 的这篇论文，并给出了更加高效的计算解决方案和相应的伪代码。

需注意的是，无论是协方差矩阵版本的方向包围盒，还是惯性矩阵版本的方向包围盒，它们都不是最优结果。例如，考查对象 A 及其方向包围盒 B，向 A 中增加某些几何元素并扩展，但仍使其处于 B 的范围内。对于上述两个版本的解决方案，一般针对更新的物体对象 A'将会产生另一个 B'。该构造过程使得 B 和 B'分别包围物体对象 A 和 A'。但一般情况下，由于两个 OBB 存在尺寸差异，其中一个必为次优的 OBB。

4.4.4 基于 PCA 的 OBB 优化

由于以协方差格式对齐的 OBB 并非最优解，因而对算法进行稍许修改即可实现性能的改进——这一点是值得思考的。例如，OBB 可以围绕某一轴进行旋转并对最小包围体加以定位。一类改进的 OBB 拟合算法为：仅沿一个主分量实现包围盒对齐。对于所有顶点在选定轴上的垂直面中的投影，可以计算得到具有最小面积的包围矩形，从而确定其余两个分量的方向。针对包含最小体空间的 OBB，该算法给出了相对于某一轴的最佳旋转方向。

[Barequet96]中给出了这一解决方案，其中分别探讨了下列 3 种不同的方法：

- ☑ 全主分量包围盒。
- ☑ 最大主分量包围盒。
- ☑ 最小主分量包围盒。

全主分量包围盒使用了 3 个主分量用于对齐 OBB，且其具体算法等价于前述小节中给出的方法。对于最大主分量包围盒，对应于最大特征值的特征向量将作为 OBB 的长度方向。因此，全部顶点将投影至垂直于该方向的面上，并能够计算投影点形成的最小包围矩形的面积，从而得到 OBB 其余两个分量上的方向。最后，最小主分量包围盒选择了最小的主分量作为初始 OBB 的方向，而后依据前面讨论过的方法加以处理。根据以往经验，[Barequet96]得出下列结论——最小主向量算法具有最佳执行效果而非最大主向量算法，并给出了一个具有说服力的解释：最大主向量显示了最大方差方向且提供最长边，因而可能产生较大的 OBB 空间。

对于一个给定的初始 OBB，可以通过迭代该方法实现一个局部最小空间包围体。具体计算过程包括：将全部顶点投影至 OBB 某一方向上的垂直面上，并与其中具有最小面积的包围矩形对齐以更新该 OBB。该迭代过程循环执行，直至相应的顶点投影（沿 OBB 的某一方向）不再有明显的改善效果——对应点处即可得到局部最小值。结合其他预处理算法（如 Mirtich 算法），该方法可作为包围盒计算的最优算法。

最后一个问题即是如何计算针对面内点集的、具有最小面积的包围矩形。这里，核心问题仍源于计算几何中的相应结果：凸多边形的最小面积包围矩形应至少存在一条边，且与多边形的某一边共线（[Freeman75]）。

因此，可以利用这一类简单的算法计算最小矩形。首先，需要计算顶点集的凸体并获取其中的凸多边形；其次，将多边形的一条边视为包围盒的方向向量，并可以得到垂直于该方向的正交轴。将全部多边形的顶点投影至这两个轴上，即可计算出矩形面积。当全部多边形中的各条边测试完毕后，具有最小面积的相应边（及其正交轴）即确定了最小面积包围矩形的方向。对于每一条边，矩形面积的时间消耗为 $O(n)$，而算法全部的时间消耗为 $O(n^2)$。算法的具体实现如下所示：

```
// Compute the center point, 'c', and axis orientation, u[0] and u[1], of
// the minimum area rectangle in the xy plane containing the points pt[].
float MinAreaRect(Point2D pt[], int numPts, Point2D &c, Vector2D u[2])
{
    float minArea = FLT_MAX;

    // Loop through all edges; j trails i by 1, modulo numPts
    for (int i = 0, j = numPts - 1; i < numPts; j = i, i++) {
        // Get current edge e0 (e0x,e0y), normalized
        Vector2D e0 = pt[i] - pt[j];
        e0 /= Length(e0);

        // Get an axis e1 orthogonal to edge e0
        Vector2D e1 = Vector2D(-e0.y, e0.x);    // = Perp2D(e0)

        // Loop through all points to get maximum extents
        float min0 = 0.0f, min1 = 0.0f, max0 = 0.0f, max1 = 0.0f;
        for (int k = 0; k < numPts; k++) {
            // Project points onto axes e0 and e1 and keep track
            // of minimum and maximum values along both axes
            Vector2D d = pt[k] - pt[j];
            float dot = Dot2D(d, e0);
            if (dot < min0) min0 = dot;
            if (dot > max0) max0 = dot;
            dot = Dot2D(d, e1);
            if (dot < min1) min1 = dot;
            if (dot > max1) max1 = dot;
        }
        float area = (max0 - min0) * (max1 - min1);

        // If best so far, remember area, center, and axes
        if (area < minArea) {
            minArea = area;
            c = pt[j] + 0.5f * ((min0 + max0) * e0 + (min1 + max1) * e1);
            u[0] = e0; u[1] = e1;
        }
    }
    return minArea;
}
```

采用旋转卡尺算法（rotating caliper，[Toussaint83]），可以将凸多边形最小面积包围矩形的时间消耗降至 $O(n \log n)$。旋转卡尺算法利用通过多边形极值点的 4 条直线将多边形包围起来，从而构成了一

个矩形。至少应存在一条直线与多边形的某一边重合。对于算法的每一次迭代过程，直线皆围绕其支撑点同步地顺时针旋转，直至获取一条直线与多边形的某一条边重合。因此，多条直线将构造出一个多边形的包围矩形。该算法重复执行，直至直线方向旋转 90°。最终，具有最小面积的包围矩形则对应着全部迭代计算过程中产生的直线构成的最小面积矩形。该算法的时间复杂度受限于凸体的计算过程，若凸体已知，则旋转卡尺算法的时间复杂度为 $O(n)$。

4.4.5 蛮力法实现 OBB 的拟合

关于实现 OBB 的紧密拟合，最后一种解决方案曾提到了以蛮力法计算 OBB。实现蛮力拟合的一种方法是以某种方式参数化 OBB 的方向向量，其中，对方向空间实现参数化的固定区间采样，并保留全部采样旋转方位中的最优 OBB。区间采样形成了较高子间隔分辨率处的最优 OBB，从而使原始 OBB 的方向得到了修正。该爬山算法重复执行且对区间实施细分，直至当前最佳 OBB 的方向无明显变化。

针对每一个执行测试的坐标系统，计算候选 OBB 时需要将其全部顶点转换至当前坐标系统中。由于这一转换过程比较耗时，因此，一旦发现候选 OBB 并不优于当前最优 OBB，应立即结束当前搜索。这样，其计算过程不必付出高昂的代价，同时还可以获得较好的拟合结果。对于主分量分析（PCA）的拟合 OBB，由于制定了一个较好的初始预测方案，因而在顶点转换过程中增加了提前退出的几率（[Miettinen02a]）。为了进一步增加提前退出的几率，可以对定义前述 OBB 的 6 个顶点（或更多）率先进行变换。正如[Miettinen02b]中所阐述的，当采用这类优化方案时，超过 90%的测试可以实现提前退出。与基于 PCA 拟合的 OBB 相比，蛮力拟合通常能够产生更加紧凑的 OBB。

前面所描述的爬山算法在更新当前最优 OBB 之前，需要考查方向空间内的多个采样点。需注意的是，[Lahanas00]中所描述的基于 OBB 拟合的优化算法，每次只是计算搜索空间内的一个采样点，它并没有利用多重采样技术并辅助优化工具消除局部最小值。

4.5 球扫掠体

在讨论完球体与盒体之后，下面将考查圆柱体类型的包围体。但是，相关的数学计算表明，圆柱体的相交测试代价昂贵而且很少应用。然而，如果将圆柱体两端加上半球体“封口”，则该包围体将具有较强的实用价值。这里，由顶点 A 和 B（形成圆柱体的中间轴）以及半径 r 构造圆柱体。半径为 r 的球体沿线段 AB 扫掠，并形成最终的包围体——球柱体。作为基本球体的扩展，该类型的体空间也是包围体族的一员。

回忆一下，两个球体的相交测试首先计算两个球心的距离，并与其半径和进行比较（可以计算半径的平方值，从而避免平方根运算产生的高昂代价）。可以利用任意类型的内部图元或中间表示结构来替换球心处的顶点，并构造出新类型的包围体。最终结果等价于：半径 r 的球体沿内部图元进行扫掠（或者从技术角度上讲，形成了球体与对应图元的 Minkowski 和）。因而，全部包围体族皆可视为球扫掠体（SSV）。内部中间图元结构中、r 范围内的全部顶点皆位于 SSV 中。图 4.11 显示了 3 种类型的球扫掠包围体。

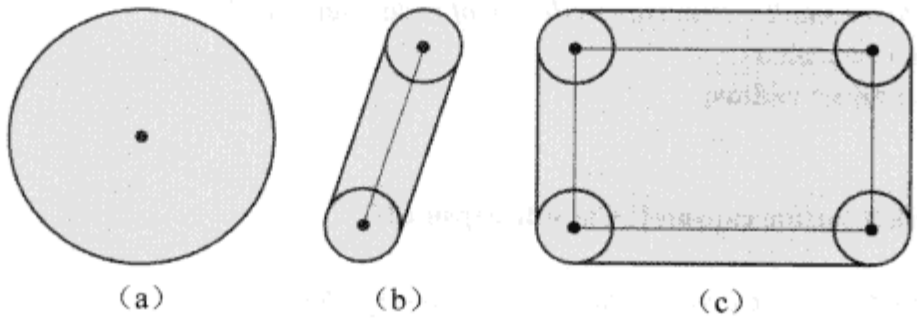


图 4.11　（a）球体扫掠点（SSP）。（b）球体扫掠直线（SSL）。（c）球体扫掠矩形（SSR）

鉴于球体的相交测试，两个 SSV 之间的相交测试变为：计算两个内部图元之间的（平方）距离，并与它们的半径（平方）和进行比较。SSV 的测试消耗完全取决于对应距离函数的消耗。为了使距离测试尽可能实现低消耗，内部图元常限于点、线段或矩形。最终的 SSV——点-球体扫掠体、直线-球体扫掠体、矩形-球体扫掠体——分别称之为球体、胶囊体（capsule）和菱形体（lozenge）。矩形-球体扫掠体类似于带有圆角边的 OBB；胶囊体也称为半球体封口圆柱体或球柱体。胶囊体和菱形体的数据结构定义如下所示：

```
// Region R =x | (x - [a + (b - a)*t])^2 <= r, 0 <= t <= 1
struct Capsule {
    Point a;        // Medial line segment start point
    Point b;        // Medial line segment endpoint
    float r;        // Radius
};
// Region R =x | (x - [a + u[0]*s + u[1]*t])^2 <= r, 0 <= s,t <= 1
struct Lozenge {
    Point a;        // Origin
    Vector u[2];    // The two edges axes of the rectangle
    float r;        // Radius
};
```

由于外形上的类似性，菱形体常作为 OBB 的有效替代物。当具有较高相似度时，与 OBB 的分离轴测试算法相比，菱形体间的距离计算消耗往往更低。

4.5.1　球扫掠体的相交测试

由前述构造过程可知，所有的球扫掠体测试都可以以相同的方式加以表达。首先计算内部图元结构之间的距离，然后用该距离与半径和进行比较。任意两类球扫掠体测试的唯一区别在于：两个体空间中的、内部图元结构之间的距离计算方法不同。球扫掠体的一个比较有用的性质是：内部图元结构之间的距离计算并不依赖其类型，可以很容易地构造混合类型之间的测试。对于球体-胶囊体与胶囊体-胶囊体而言，这两种类型之间的测试代码如下所示：

```
int TestSphereCapsule(Sphere s, Capsule capsule)
{
    // Compute (squared) distance between sphere center and capsule line segment
    float dist2 = SqDistPointSegment(capsule.a, capsule.b, s.c);
```

```
    // If (squared) distance smaller than (squared) sum of radii, they collide
    float radius = s.r + capsule.r;
    return dist2 <= radius * radius;
}

int TestCapsuleCapsule(Capsule capsule1, Capsule capsule2)
{
    // Compute (squared) distance between the inner structures of the capsules
    float s, t;
    Point c1, c2;
    float dist2 = ClosestPtSegmentSegment(capsule1.a, capsule1.b,
    capsule2.a, capsule2.b, s, t, c1, c2);

    // If (squared) distance smaller than (squared) sum of radii, they collide
    float radius = capsule1.r + capsule2.r;
    return dist2 <= radius * radius;
}
```

这里，函数 **SqDistPointSegment( )**和 **ClosestPtSegmentSegment( )**可参见第 5 章（5.1.2.1 节和 5.1.9 节）。基于半空间测试的 SSR 距离测试（通过适当简化，也适用于 SSL 和 SSP）可参见[Larsen99]和[Larsen00]。

4.5.2 球体扫掠体包围体的计算

关于 SSV 的计算方法已在 4.5.1 节中介绍过。例如，通过计算主轴并选择最大轴作为胶囊体的长度方向，可以实现胶囊体的拟合；同时，选择次长轴作为胶囊体的半径。或者，为了获取一条与顶点集相匹配的直线，该长度也可采用最小二乘法获得。对于 SSR，将会使用到全部的 3 个轴，其中最短轴为矩形的面法线。详细内容可参见[Eberly01]、[Larsen99]和[Larsen00]。

4.6 半空间相交体

显然，除了球体，多数包围体皆为凸多面体。这些多面体包围体可视为半空间集合的交集，其中，半空间对应包围体的各个面。例如，AABB 和 OBB 为 6 个半空间的交集；4 个半空间的交集可以构造出四面体——形成封闭包围体所需的最小半空间数量。一般情况下，半空间的数量越多，其最终的交集体空间将达到的物体匹配效果越好。如果被包围的物体为多面体，则潜在的紧密凸包围体将是该物体对象的凸体。在这种情况下，凸体所包含的面数量则限定了一个上限值，即构造包围体所需的半空间数量。凸体是一类重要的包围体，本书第 9 章将详细讨论凸体碰撞检测算法。

虽然凸体可以构造较为紧密的包围体，但并不是处理包围体的最佳选择。凸体仍包含某些缺陷，例如，代价高昂且难于计算，显示结构消耗大量内存，操作复杂。通过限制相交体中的半空间数量，可以构造某些简化的、可替代的包围体。几种不同类型的包围体将在后续内容中加以描述。

4.6.1　Kay-Kajiya 平行平面空间包围体

Kay-Kajiya 包围体因其可加速计算光线-物体相交测试而得到首次引用。Kay-Kajiya 包围体也称作基于平行平面空间的平行六面体族（[Kay86]）。所谓平行平面空间（slab）是指两平行面之间的无限区域（如图 4.12 所示）。该面集使用了一个方向向量 **n**（面集法线）加以表达，并设定了两个标量用于记录两个面（沿 **n** 方向）距原点的有符号距离。

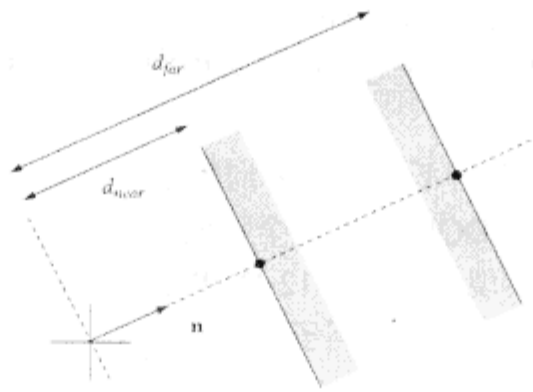


图 4.12　平行平面空间是指两平行面之间的无限区域，并由法线 **n** 和距原点的两个有符号距离加以定义

```
// Region R = {(x, y, z) | dNear <= a*x + b*y + c*z <= dFar }
struct Slab {
    float n[3];        // Normal n = (a, b, c)
    float dNear;       // Signed distance from origin for near plane (dNear)
    float dFar;        // Signed distance from origin for far plane (dFar)
};
```

构造包围体时，可以选择多条法线。对于每一条法线，定位几组平面并沿法线方向在其两边包围物体对象。对于多边形物体对象，平面的位置可以通过计算法线与物体顶点的点积值得到。其中，相应的最小值和最大值即为定义平面位置的两个标量值。

至少需要 3 个平行平面空间才可形成一个封闭的 3D 包围体。AABB 和 OBB 即是 3 个平行平面空间交集构造的包围体的两个实例。通过增加平面空间的数量，可构造任意紧密程度的平行平面空间来匹配物体对象的凸体。

在最初的光线相交测试中，快速测试方案基于下列情景：当且仅当光线同时位于全部（3 个）包含一定间隔的平行平面空间之内，或者光线-平行平面空间相交测试区间的交集不为 0 时，参数化光线与平行平面空间相交。由于几组平面共享同一法线，因而两个光线-平行平面之间也可以共享同一计算过程，从而改善了测试性能。

虽然平行平面空间包围体可以加速光线的相交测试过程，但在物体间的相交测试中，不可以简单地照搬使用该算法。然而，通过共享法线，算法可以执行快速的物体相交测试（具体内容参见 4.6.2 节）。

4.6.2　离散有向多面体（*k*-DOP）

与 Kay-Kajiya 算法基于同一理念，[Konečný97]和[Klosowski98]建议，平行平面包围体也可称作离

散有向多面体（*k*-DOP）或固定方向凸体（FDH）。对于这两个术语，后者可能更具有描述性，但前者具有更广泛的使用性并被本书所采纳。*k*-DOP 即是凸多面体，除了法线定义为固定的轴集合且共享于全部 *k*-DOP 包围体之中，它几乎等同于平行平面空间包围体。法线分量一般限定于集合{−1, 0, 1}内，且法线通常不加以范化。这里，法线将有助于 *k*-DOP 实现更加快捷的计算过程，其重要性将体现在 *k*-DOP 的动态重对齐过程中。通过在物体间共享法线，*k*-DOP 将占用很少的内存存储空间——只需存储每个轴的 min-max 区间。例如，8-DOP 则改写为：

```
struct DOP8 {
    float min[4];        // Minimum distance (from origin) along axes 0 to 3
    float max[4];        // Maximum distance (from origin) along axes 0 to 3
};
```

6-DOP 一般称作多面体且各个面与下列 6 个方向轴对齐：(±1, 0, 0)、(0, ±1, 0)和(0, 0, ±1)。该 6-DOP 即所谓的 AABB，但 AABB 只是 6-DOP 中的一个特例。任何有向包围盒都可描述为 6-DOP；8-DOP 中的各个面与(±1, ±1, ±1)8 个方向实现轴对齐；12-DOP 包含(±1, ±1, 0)、(±1, 0, ±1)和(0, ±1, ±1)12 个方向。图 4.13 显示了一个 2D 空间内 8-DOP 的例子。

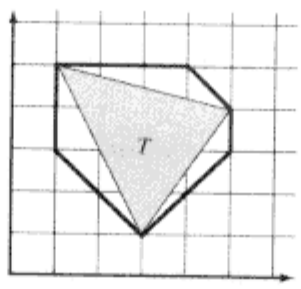


图 4.13　针对轴(1, 0)、(0, 1)、(1, 1)、(1, −1)，三角形(3, 1)、(5, 4)、(1, 5)的 8-DOP 可表示为{1, 1, 4, −4, 5, 5, 9, 2}

可以利用 6-DOP 和 8-DOP 的组合来定义 14-DOP；18-DOP、20-DOP 以及 26-DOP 可以采用类似的方式加以构造。其中，14-DOP 对应着一个切掉 8 个角的轴对齐包围盒，18-DOP 则为切割了 12 条边的 AABB。在[Crosnier99]中，18-DOP 也称作 tribox。

需注意的是，*k*-DOP 不仅是平行平面空间的交集，还构造了最紧密的体空间。例如，三角形(0, 0)，(1, 0)，(0, 1)可以利用多个平面空间的交集加以表达，但只存在唯一一个 *k*-DOP 实现其最终描述。当且仅当平面空间中的平面与平面空间体存在公共点时，才可定义为 *k*-DOP 平面空间。这种紧密性测试标准是非常重要的，因为 *k*-DOP 相交测试并不作用于相交体之上，而仅作用于平行平面定义之上。如果缺乏这种限制，相交测试将变得十分低效。

相对于有向包围盒，由于存在固定方向平面，*k*-DOP 相交测试将会显著地提高测试速度，即使是对更大 *k* 值的 DOP 测试也是如此。内存方面，OBB 的内存消耗大约对应一个 14-DOP 的内存消耗量。通常，随着 *k* 值增大，*k*-DOP 将会更加接近物体的凸体，且一般要比 OBB 具有更紧密的拟合性。但对于与 *k*-DOP 轴无法实现良好对齐性的几何体来说，OBB 往往更具紧密性。同时，OBB 在近似逼近计算方面也具有较好的性能。

即使空间体之间很少发生碰撞，*k*-DOP 也执行更新或“翻转”操作——这是 *k*-DOP 最大的缺陷。由于该操作代价很高，无论何时都应该避免这类“翻转”操作。因此，常常采用包围球来表现物体对象（即如果包围球测试失败，就不必执行 *k*-DOP 测试）。一般情况下，若干动态物体与多个静态物体

执行碰撞测试时（即需更新若干 *k*-DOP），或同一物体涉及多次碰撞测试时（即更新 *k*-DOP 的计算代价根据测试分期执行），*k*-DOP 往往具有更佳的表现力。

[Zachmann00]中考查了一个类似的 *k*-DOP。其中，采用了模拟退火算法对 *k*-DOP 中的各轴加以选择，且 *k* 个顶点随机分布在一个单位球上，并规定：对于每一个顶点 *P*，−*P* 也位于当前集合中。

4.6.3　*k*-DOP—*k*-DOP 相交测试

由于物体对象间共享固定轴集合，*k*-DOP 相交测试类似于两个 AABB 之间的相交测试，同时，其计算难易程度也不会超过 AABB 之间的相交测试。通过该相交测试简单地查看 *k*/2 个区间，若区间内不存在相交，则 *k*-DOP 之间也不相交；只有当全部区间内接相交时，*k*-DOP 才相交。

```
int TestKDOPKDOP(KDOP &a, KDOP &b, int k)
{
    // Check if any intervals are non-overlapping, return if so
    for (int i = 0; i < k / 2; i++)
        if (a.min[i] > b.max[i] || a.max[i] < b.min[i])
            return 0;

    // All intervals are overlapping, so k-DOPs must intersect
    return 1;
}
```

类似于有向包围盒，轴的测试顺序也可能对计算性能产生一定的影响。由于沿“接近”方向上的区间可能产生较类似的相交状态，在不同（正交）方向上应采取连续区间测试。一种实现方法是利用贪心算法处理轴的测试顺序：首先测试任意轴——这将排序各轴，即后续轴为与当前轴点积值约为 0 的某一轴。

4.6.4　*k*-DOP 的计算与重对齐

物体对象的 *k*-DOP 计算可看作是 AABB 计算的一般性法则，因此 *k*-DOP 间的相交测试即是 AABB 间相交测试的一般化表示。基于此，可以简单地计算沿 *k*-DOP 各定义轴上的对象顶点的投影分布。这与 AABB 的唯一区别在于投影轴的数量。这里，投影的方向并不是任意的，由于最多只考虑 3 个方向的轴上顶点投影，将投影轴各分量限制在{−1, 0, 1}集合之间可使硬编码过程更具简洁性。例如，8-DOP 的计算方式如下所示：

```
// Compute 8-DOP for object vertices v[] in world space
// using the axes (1,1,1), (1,1,-1), (1,-1,1) and (-1,1,1)
void ComputeDOP8(Point v[], int numPts, DOP8 &dop8)
{
    // Initialize 8-DOP to an empty volume
    dop8.min[0] = dop8.min[1] = dop8.min[2] = dop8.min[3] = FLT_MAX;
    dop8.max[0] = dop8.max[1] = dop8.max[2] = dop8.max[3] = -FLT_MAX;

    // For each point, update 8-DOP bounds if necessary
```

```
    float value;
    for (int i = 0; i < numPts; i++) {
        // Axis 0 = (1,1,1)
        value = v[i].x + v[i].y + v[i].z;
        if (value < dop8.min[0]) dop8.min[0] = value;
        else if (value > dop8.max[0]) dop8.max[0] = value;

        // Axis 1 = (1,1,-1)
        value = v[i].x + v[i].y - v[i].z;
        if (value < dop8.min[1]) dop8.min[1] = value;
        else if (value > dop8.max[1]) dop8.max[1] = value;

        // Axis 2 = (1,-1,1)
        value = v[i].x - v[i].y + v[i].z;
        if (value < dop8.min[2]) dop8.min[2] = value;
        else if (value > dop8.max[2]) dop8.max[2] = value;

        // Axis 3 = (-1,1,1)
        value = -v[i].x + v[i].y + v[i].z;
        if (value < dop8.min[3]) dop8.min[3] = value;
        else if (value > dop8.max[3]) dop8.max[3] = value;
    }
}
```

虽然 *k*-DOP 在平移过程中保持恒定，但旋转操作将使之与原始轴呈现非对齐状态。如同 AABB 中所示，*k*-DOP 旋转后需要实现重对齐。然而，由于 *k*-DOP 的重计算过程涉及将对象顶点转换至新的空间内，当顶点数量巨大时，这一步骤将十分耗时。一种比较高效的重对齐方案类似于 4.2.5 节计算 AABB 时所描述的爬山算法，唯一的区别在于记录 *k* 个面方向上的顶点指针而非 6 个。当帧间相关一致性要求较高且物体旋转量较小时，跟踪记录顶点将是一类较好的解决方案。然而，该算法在最坏情况下的复杂度为 $O(n^2)$，当帧间相关一致性要求较低时，算法的效率较低。要知道，爬山法可以生成较为紧凑的包围体。

另外一种近似算法的实现方式为：在预处理时计算并存储 *k*-DOP 初始状态下的各顶点，*k*-DOP 在运行时刻重计算上述顶点，并利用当前旋转矩阵将顶点转换至世界坐标空间中。该算法等价于 4.2.6 节中针对 AABB 所描述的方法。

初始方向状态下的 *k*-DOP 顶点集可利用对偶映射变换加以计算。这里，顶点(a, b, c)针对平面 $ax + by + cz = 1$ 实现对偶映射，反之亦然。利用平面方程表示 *k*-DOP 中定义的平面，于是通过计算该平面的对偶映射凸体，凸体面（边和顶点）映射为相交顶点（边和面）。为了使得该对偶转换能够正常工作，*k*-DOP 需要进行适当的平移以使之包含原点。对于半空间交集形成的包围体，体内一点可通过线性规划方法得到（参见 9.4.1 节）。

另一种计算包围体内部顶点的简易方法为交替投影算法（MAP）。该算法始于空间内部中的任一顶点，并以任意顺序遍历全部半空间，如果顶点位于半空间外部，则将其投影至当前半空间的边界超平面上并更新该顶点。该循环过程遍历全部半空间，直至顶点位于半空间内部，从而保证了计算过程的收敛性。若顶点位于体外，则最终顶点将位于体边界上，即某一特定的超平面之上（若顶点位于体内，则算法返回该顶点）。对于多个各异的初始顶点，可重复使用交替投影算法获取一个内部顶点，

直至出现位于不同包围超平面上的另一个顶点，两个边界顶点的中点即为内部顶点。对于特定的顶点输入，MAP 可能具有较慢的收敛速度。然而，对于广泛用于包围体的体空间而言，较慢的收敛速度并不是什么问题。MAP 的更多相关内容可参见[Deutsch01]；对偶（或极性）变换的详细讨论可参见[Preparata85]、[O'Rourke98]或[Berg00]。

一种计算初始顶点集的简单方案是：针对 *k*-DOP 包围面的输入集合，计算 3 个非共面平面的组合，每 3 个平面相交于一点（相交点的计算可参见 5.4.5 节）。待全部相交顶点计算完毕，可剔除所有位于 *k*-DOP 边界面正面外的顶点，剩余顶点即为所求。

也可以采用基于线性规划技术的算法实现 *k*-DOP 的重对齐问题，具体内容可参见[Konečný97]和[Konečný98]。[Fünfzig03]中描述了更加复杂的重对齐策略。线性规划技术的相关内容将在 9.4 节中加以介绍。

4.6.5　近似凸体相交测试

两个凸多边形的分离测试易于实现（例如，可采用 4.4.4 节中描述的旋转卡尺算法）。但是，该问题对于多面体而言却并非易事。关于该问题的精确算法将在第 9 章加以阐述。

在多种场合下，多面体的精确碰撞检测既无必要也不是目的。通过近似算法可以实现轻量级的测试方案，而且实现这一类简单而快速的算法并非遥不可及。

一类解决方案可通过维护凸体的既定平面和顶点加以实现。为了在两个凸体间执行碰撞测试，可以在全部顶点与另一凸体面间进行空间位置测试：查看顶点是否位于某一面的外部。若是，则两凸体分离否则可初步认为两凸体之间存在碰撞。根据分离轴测试原则，上述步骤对应着凸体面法线间的分离测试，而非凸体间的边-边测试。

另外一种方案仅利用球体集代替顶点集。需要对球体加以选择以使球体集对当前凸体实现最佳近似。因此，测试过程转变为球体（而非顶点）与平面之间的测试。该理念源于：与顶点数量相比，只需对较少的球体加以测试。虽然该算法比较快速，但缺乏相应的精确度。通过降低球体集合的尺寸而提高精确度，该集合通常可以得到优化。

而对于 *k*-DOP，可以排序已存储的平面并使后续平面间尽可能地保持彼此垂直，从而加速测试过程。类似地，通过消除簇导致的退化现象，顶点（或球体）可呈现随机顺序。采用球体包围顶点，并在顶点测试之前利用球体与平面进行测试，通常可以保证测试有效地提前退出。

与其他类型的包围体测试相比，上述测试过程仍成本高昂，并往往辅以简单的包围体预测试（如球体-球体测试）以排除非相交的凸体测试对象。另外，第 9 章中给出的一致性算法也将有助于降低凸体测试的数量。

4.7　其他类型的包围体

除了这里所提及的包围体类型，其他包围体还包括圆锥体（[Held97]、[Eberly02]）、圆柱体（[Held97]、[Eberly00]、[Schömer00]）、球壳（[Krishnan98]）、椭圆体（[Rimon92]、[Wang01]、[Choi02]、[Wang02]、[Chien03]）以及 zonotope 体（[Guibas03]）。圆锥体、圆柱体以及椭圆体自不必多说，球壳是指两同心

球体以及一个圆锥体（锥顶位于球心）之间的空间交集部分。zonotope 体是一类具有某种特征的且呈中心对称的多面体。需要说明的是，作为包围体来讲，上述形状的应用范围并不广泛，皆因其相交测试的计算开销过于庞大。因此，这里并不过多地讨论这一话题。

需要指出的是，尽管椭圆体-椭圆体之间的相交测试开销较大，但通过对坐标空间进行不规则缩放，椭圆体与三角形或其他类型的多边形间的测试可转换为球体与斜三角形之间的测试。因此，针对某些测试集而言，椭圆体作为包围体还是可行的。

4.8 小结

包围体可视为一类简单的几何体，用于包围一个或多个具有复杂几何形状的对象。其中，较为常见的包围体是球体与盒体。另外，若包围体与对象间需要实现紧密拟合，则也可采用平行平面体或凸壳。包围体常用于前期相交剔除测试，以避免对原几何体进行成本高昂的直接测试。需要指出的是，就包围体的选择而言，也存在相应的折中方案，如 4.1 节所述。通过使用紧密拟合的包围体，相应的剔除效果也就更加显著；但与此用时，对应包围体的测试开销也会更加高昂并将消耗更多的存储空间。通常情况下，包围体往往在预处理阶段加以计算，必要时，可基于被包围对象并在运行期内实施相应的转换操作，以更好地配合对象的位移。

本章不仅详细介绍了常见的包围体类型及其计算方法，还进一步讲解了（相同类型体空间的）具有结构相似性的包围体间的相交测试，其目的在于为第 5 章中的相关内容打下良好的基础。第 5 章将详细介绍基本几何图元（不同种类）间的相交测试和距离计算，包括直线、线段、球体、盒体、三角形、多边形以及多面体。

第 5 章　基本图元测试

当高层系统在碰撞检测计算过程中剔除了大量分离物体后，碰撞系统需要在图元或包围体之间执行底层测试，以确定相交状态。有时，仅判断是否发生碰撞是远远不够的，某些情况下还需要获取准确的碰撞点的信息。本章将对如何高效地实现底层测试进行详细的分析。另外，本章还将给出特定的数学概念，并利用数学检测手段对测试过程进行深入的推导。

需要注意的是，在具体实现过程中，某些数学表达式将会受限于浮点运算导致的数值精确度问题。这一问题会在本章中加以扼要的阐述，基于数值精确度的健壮性问题，将在第 11 章进行深入探讨。

5.1　最近点计算

最近点计算是碰撞查询中强有力的工具。获取两物体对象之间的最近点后，即可得到它们之间的距离。如果两对象的合成最大位移量小于二者间的距离，则对象间处于分离状态。另外，在层次结构表达方式中，通过最近点计算可适当地排除距离过大且不会产生碰撞的部分层次结构。

两对象间的最近点计算可视为一类极小值问题。这里可采用微积分方法（如拉格朗日乘子法）对该问题进行求解。在本章中，则推荐使用相关的几何算法。下面将对各类几何对象的最近点计算进行深入的分析。

需要注意的是，两对象间的最近点计算有时可以较低代价的渐进方式实现，这将有助于执行快速的碰撞测试。最近点的渐进式计算将在第 9 章中针对凸体对象间的碰撞测试进行更深入的讨论。

5.1.1　点到面的最近点

一点 P 以及法线 $\mathbf{n}$ 定义了一个平面 π，所有该平面上的点 X 都满足方程 $\mathbf{n}\cdot(X-P)=0$（即从点 P 指向 X 的向量垂直于 $\mathbf{n}$）。现令 Q 为空间内任意一点，则面内距 Q 的最近点 R 为点 Q 在该面上的正交投影，即从顶点 Q 处向平面垂直（依据法线 $\mathbf{n}$）移动。同时，对于值 t，有 $R=Q-t\mathbf{n}$，如图 5.1 所示。将该表达式带入平面方程并求解 t：

$$\mathbf{n}\cdot((Q-t\mathbf{n})-P)=0 \Leftrightarrow \text{（将平面方程中的 } X \text{ 替换为 } R\text{）}$$

$$\mathbf{n}\cdot Q-t(\mathbf{n}\cdot\mathbf{n})-\mathbf{n}\cdot P=0 \Leftrightarrow \text{（展开点积操作）}$$

$$\mathbf{n}\cdot(Q-P)=t(\mathbf{n}\cdot\mathbf{n}) \Leftrightarrow \text{（收集同类项并将表达式移至 } RHS\text{）}$$

$$t=\mathbf{n}\cdot(Q-P)/(\mathbf{n}\cdot\mathbf{n}) \Leftrightarrow \text{（两边同时除以 } \mathbf{n}\cdot\mathbf{n}\text{）}$$

将 t 值带入到 $R=Q-t\mathbf{n}$ 并得到投影点 R：

$$R=Q-(\mathbf{n}\cdot(Q-P)/(\mathbf{n}\cdot\mathbf{n}))\mathbf{n}$$

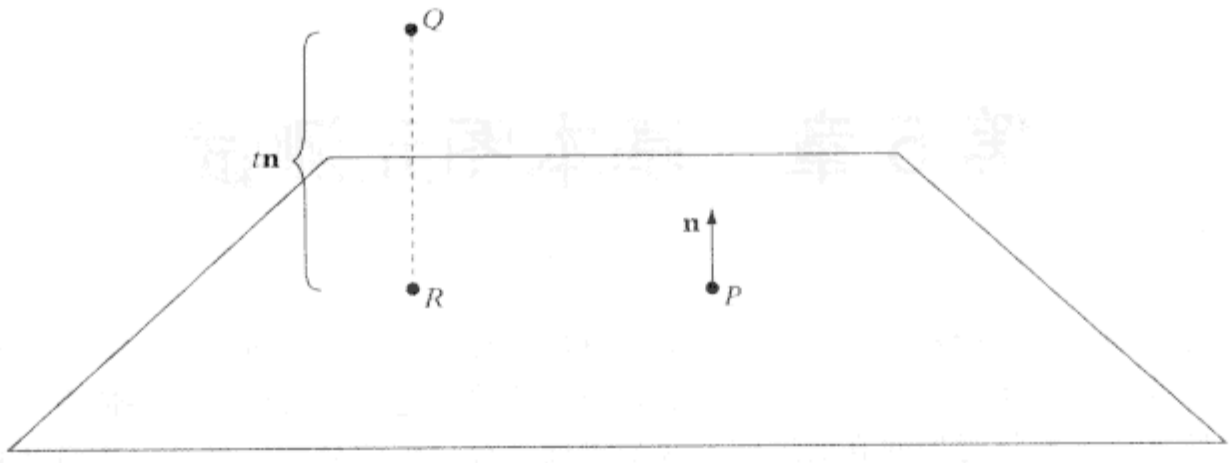


图 5.1　由点 P 和 $\mathbf{n}$ 定义的平面 π。点 Q 在 π 上的正交投影为 R，即平面 π 上距点 Q 的最近点

当法线 $\mathbf{n}$ 为单位向量时，上述 t 值简化为 $t = \mathbf{n} \cdot (Q - P)$；$R = Q - (\mathbf{n} \cdot (Q - P))\mathbf{n}$。从等式中不难发现，对于任意一点 Q，$t = \mathbf{n} \cdot (Q - P)$对应着 Q 距离该面 $\mathbf{n}$ 个长度单位的有向距离。如果 t 为正值，则 Q 位于该面的正面（如果 t 为负值，则 Q 位于该面的背面）。

当平面按照 $\mathbf{n} \cdot X = d$ 格式给出时，则有表达式 $t = ((\mathbf{n} \cdot Q) - d)/(\mathbf{n} \cdot \mathbf{n})$。因此，一点距平面最近点的代码如下所示：

```
Point ClosestPtPointPlane(Point q, Plane p)
{
    float t = (Dot(p.n, q) - p.d) / Dot(p.n, p.n);
    return q - t * p.n;
}
```

如果平面方程中采用了单位法向量，则 t 简化为 $t = (\mathbf{n} \cdot Q) - d$，如下所示：

```
Point ClosestPtPointPlane(Point q, Plane p)
{
    float t = Dot(p.n, q) - p.d;
    return q - t * p.n;
}
```

则 Q 至平面的有符号距离可通过 t 的计算返回值得到：

```
float DistPointPlane(Point q, Plane p)
{
    // return Dot(q, p.n) - p.d; if plane equation normalized (||p.n||==1)
    return (Dot(p.n, q) - p.d) / Dot(p.n, p.n);
}
```

5.1.2　点至线段的最近点

令 AB 为两端点 A、B 定义的线段，给定任意一点 C，该问题为：确定线段 AB 上距 C 的最近点 D。如图 5.2 所示，可以将点 C 投影至 AB 的延长线上，如果投影点 P 位于线段内，则点 P 为所求；如果点 P 位于线段外部，则线段上的近 C 端点为最近点。

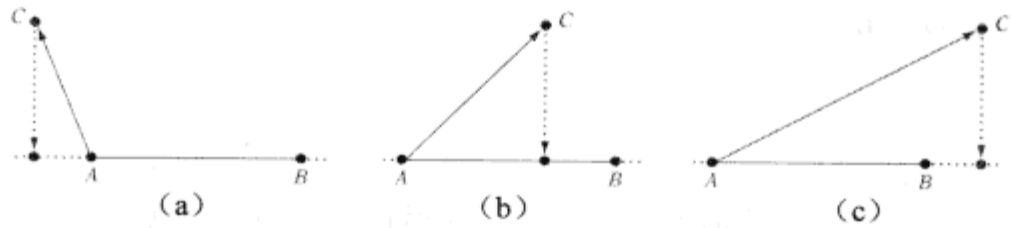


图 5.2　点 C 在 AB 上的投影。（a）靠近端点 A 且位于 AB 外部。（b）位于 AB 内部。（c）靠近端点 B 且位于 AB 外部

直线 AB 上的一点可参数化为 $P(t) = A + t(B - A)$。利用点积的投影性质，t 可表示为 C 在直线上的投影，且 $t = (C - A) \cdot \mathbf{n}/\|B - A\|$，其中 $\mathbf{n} = (B - A)/\|B - A\|$ 且为 AB 方向上的单位向量。

若在线段内获取最近点，则有 $0 \leqslant t \leqslant 1$，且在参数方程中带入 t 值即可得到最近点 D。实现代码如下所示：

```
// Given segment ab and point c, computes closest point d on ab.
// Also returns t for the position of d, d(t) = a + t*(b - a)
void ClosestPtPointSegment(Point c, Point a, Point b, float &t, Point &d)
{
    Vector ab = b – a;
    // Project c onto ab, computing parameterized position d(t) = a + t*(b – a)
    t = Dot(c – a, ab) / Dot(ab, ab);
    // If outside segment, clamp t (and therefore d) to the closest endpoint
    if (t < 0.0f) t = 0.0f;
    if (t > 1.0f) t = 1.0f;
    // Compute projected position from the clamped t
    d = a + t * ab;
}
```

如果除法操作成本高昂，则可预先保存非负的分母值并延迟该除法操作。优化后的代码如下所示：

```
// Given segment ab and point c, computes closest point d on ab.
// Also returns t for the parametric position of d, d(t) = a + t*(b - a)
void ClosestPtPointSegment(Point c, Point a, Point b, float &t, Point &d)
{
    Vector ab = b – a;
    // Project c onto ab, but deferring divide by Dot(ab, ab)
    t = Dot(c – a, ab);
    if (t <= 0.0f) {
        // c projects outside the [a,b] interval, on the a side; clamp to a
        t = 0.0f;
        d = a;
    } else {
        float denom = Dot(ab, ab);        // Always nonnegative since denom = ||ab||∧2
        if (t >= denom) {
            // c projects outside the [a,b] interval, on the b side; clamp to b
            t = 1.0f;
            d = b;
        } else {
            // c projects inside the [a,b] interval; must do deferred divide now
            t = t / denom;
```

```
                d = a + t * ab;
            }
        }
    }
```

同一算法也可用于计算光线上的最近点或直线上的最近点。对于光线，当 t 出现负值时，则需要对该值进行截取；对直线则无此限制。

5.1.2.1 点到线段的距离

一点 C 与线段 AB 之间的（平方）距离可以直接获取，而无须显式计算线段 AB 上距 C 的最近点 D。如前所述，需要考查 3 种情形：当 $AC \cdot AB \leqslant 0$ 时，点 A 为距 C 的最近点且平方距离为 $AC \cdot AC$；当 $AC \cdot AB \geqslant AB \cdot AB$ 时，点 B 为距 C 的最近点且平方距离为 $BC \cdot BC$；最后一种情况，当 $0 < AC \cdot AB < AB \cdot AB$ 时，平方距离为 $CD \cdot CD$，其中：

$$D = A + \frac{AC \cdot AB}{AB \cdot AB} AB$$

然而，由于 $CD \cdot CD$ 可简化为：

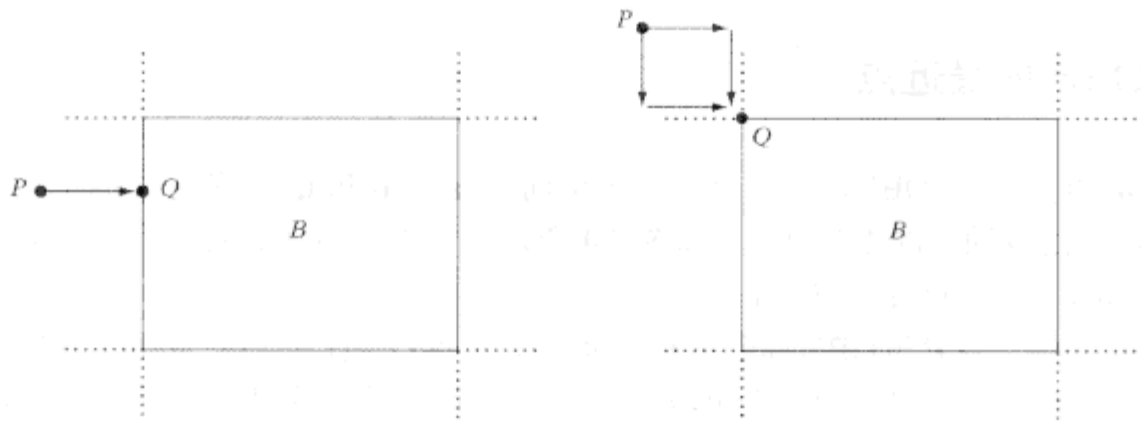
$$AC \cdot AC - \frac{(AC \cdot AB)^2}{AB \cdot AB}$$

则点 D 无须计算。可预计算某些重复的子项，实现代码如下所示：

```
// Returns the squared distance between point c and segment ab
float SqDistPointSegment(Point a, Point b, Point c)
{
    Vector ab = b – a, ac = c – a, bc = c – b;
    float e = Dot(ac, ab);
    // Handle cases where c projects outside ab
    if (e <= 0.0f) return Dot(ac, ac);
    float f = Dot(ab, ab);
    if (e >= f) return Dot(bc, bc);
    // Handle cases where c projects onto ab
    return Dot(ac, ac) – e * e / f;
}
```

5.1.3 点至 AABB 的最近点

令 B 为轴对齐的包围盒且 P 为空间任意一点，则 B 上（或之内）距 P 的最近点 Q 可以通过下列方式获得：在某一轴上将 P 限定在 B 的边界上。存在 4 种情况限定 P 的最终结果：如果 P 位于 B 的内部，则截取点为点 P 自身——即 B 上的最近点；如果 P 位于 B 的某一 Voronoi 域表面上，则截取操作将 P 限定在 B 中的该面之上（关于 Voronoi 域的相关内容，可参见第 3 章）；截取操作依据点 P 在 B 上的正交投影并生成 B 上的最近点，如果点 P 为 B 中 Voronoi 域顶点，则 P 的截取操作将视该顶点为最终结果——即 B 上的最近点；最后，如果 P 位于 Voronoi 域中的某一边上，P 的截取操作结果为该边上的正交投影，从而得到 B 上距 P 的最近点。上述计算过程可工作于 2D 和 3D 环境下。在 2D 中，截取过程如图 5.3 所示。



（a）　　（b）

图 5.3　在包围盒 B 的边界上对 P 进行截取操作。（a）针对某一边的 Voronoi 域。（b）针对某一顶点的 Voronoi 域

当点 P 位于 Voronoi 域中的某一条边上，或为 Voronoi 域中的某一顶点时，其最近点计算的实现代码如下所示：

```
// Given point p, return the point q on or in AABB b that is closest to p
void ClosestPtPointAABB(Point p, AABB b, Point &q)
{
    // For each coordinate axis, if the point coordinate value is
    // outside box, clamp it to the box, else keep it as is
    for (int i = 0; i < 3; i++) {
        float v = p[i];
        if (v < b.min[i]) v = b.min[i];          // v = max(v, b.min[i])
        if (v > b.max[i]) v = b.max[i];          // v = min(v, b.max[i])
        q[i] = v;
    }
}
```

在支持 SIMD 指令的 CPU 体系结构中，上述函数常实现为两条指令：max 指令及其后的 min 指令。

5.1.3.1　点到 AABB 的距离

对于顶点 P，当获取包围盒 B 上的最近点 Q 并计算二者距离时，实际上无须显式计算最近点 Q，如下列代码所示（为了简化计算过程并避免昂贵的开方计算，代码中使用了平方距离）：

```
// Computes the square distance between a point p and an AABB b
float SqDistPointAABB(Point p, AABB b)
{
    float sqDist = 0.0f;
    for (int i = 0; i < 3; i++) {
        // For each axis count any excess distance outside box extents
        float v = p[i];
        if (v < b.min[i]) sqDist += (b.min[i] - v) * (b.min[i] - v);
        if (v > b.max[i]) sqDist += (v - b.max[i]) * (v - b.max[i]);
    }
    return sqDist;
}
```

5.1.4 点至 OBB 的最近点

设定 B 为中心点为 C 的 OBB，且 3 个正交单位向量 $\mathbf{u}_0$、$\mathbf{u}_1$ 和 $\mathbf{u}_2$ 定义了 B 的 x、y 及 z 轴。3 个标量 e_0、e_1、e_2 确定了沿各轴向的 1/2 轴长（如图 5.4 所示）。在该表达方式中，所有 B 中的点 S 可记为 $S = C+a\mathbf{u}_0 +b\mathbf{u}_1 +c\mathbf{u}_2$，其中$|a|\leqslant e_0$，$|b|\leqslant e_1$，$|c|\leqslant e_2$。

世界坐标空间中的一点 P 与 OBB 局部坐标空间中的一点 $Q = (x, y, z)$之间存在下列关系：$P = C + x\mathbf{u}_0 +y\mathbf{u}_1+z\mathbf{u}_2$。给定一点 P，其 OBB 空间坐标可按下列方式加以计算。这里，只给出了坐标 x 的推导过程，其他坐标分量也可采用类似的计算方式。

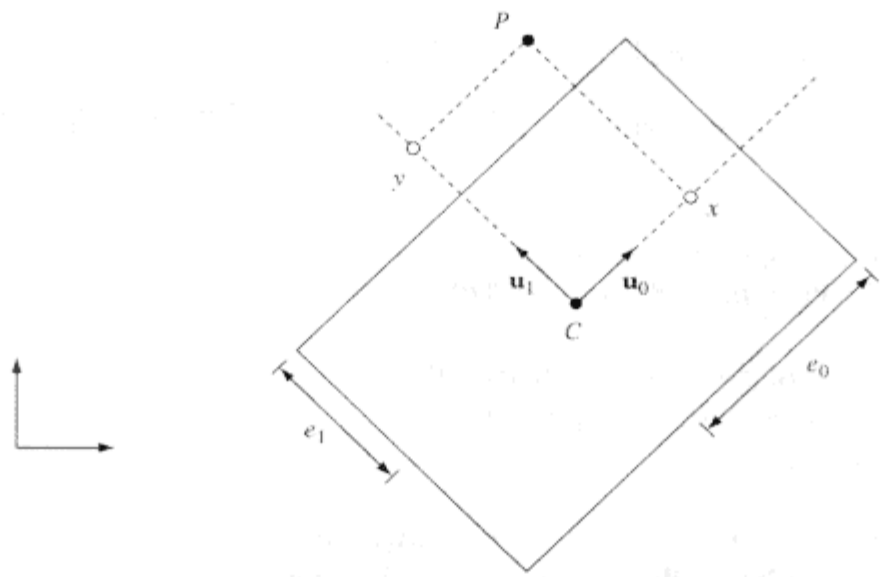


图 5.4　世界空间中的顶点 P 可表示为 2D 环境下 OBB 坐标系统中的点(x, y)

$$P = C + x\mathbf{u}_0+y\mathbf{u}_1+z\mathbf{u}_2 \Leftrightarrow \text{（初始表达式）}$$

$$P - C = x\mathbf{u}_0+ y\mathbf{u}_1+ z\mathbf{u}_2 \Leftrightarrow \text{（将 } C \text{ 移至左边）}$$

$$(P - C)\cdot\mathbf{u}_0=(x\mathbf{u}_0+y\mathbf{u}_1+z\mathbf{u}_2)\cdot\mathbf{u}_0 \Leftrightarrow \text{（两边乘以 } \mathbf{u}_0\text{）}$$

$$(P - C)\cdot\mathbf{u}_0=x(\mathbf{u}_0\cdot\mathbf{u}_0) + y(\mathbf{u}_1\cdot\mathbf{u}_0) + z(\mathbf{u}_2\cdot\mathbf{u}_0) \Leftrightarrow \text{（扩展点积表达式）}$$

$$(P - C)\cdot\mathbf{u}_0= x \quad \text{（利用 } \mathbf{u}_0\cdot\mathbf{u}_0= 1,\ \mathbf{u}_1\cdot\mathbf{u}_0=\mathbf{u}_2\cdot\mathbf{u}_0=0 \text{ 进行化简）}$$

因此，OBB 的坐标定义为：$x = (P - C)\cdot\mathbf{u}_0$，$y = (P -C)\cdot\mathbf{u}_1$，$z = (P - C)\cdot\mathbf{u}_2$。当计算 B 上（或内部）距 P 的最近点 R 时，可参照 AABB 中的同一算法并应用于 OBB 中，并将点 P 首先转换为 OBB 坐标系统中的点 Q，同时将 Q 限定在 e_0、e_1、e_2 范围之内，最后将点 Q 转换回世界坐标系统中。代码如下所示：

```
// Given point p, return point q on (or in) OBB b, closest to p
void ClosestPtPointOBB(Point p, OBB b, Point &q)
{
    Vector d = p - b.c;
    // Start result at center of box; make steps from there
    q = b.c;
    // For each OBB axis...
    for (int i = 0; i < 3; i++) {
```

```
        // ...project d onto that axis to get the distance
        // along the axis of d from the box center
        float dist = Dot(d, b.u[i]);
        // If distance farther than the box extents, clamp to the box
        if (dist > b.e[i]) dist = b.e[i];
        if (dist < -b.e[i]) dist = -b.e[i];
        // Step that distance along the axis to get world coordinate
        q += dist * b.u[i];
    }
}
```

从数学角度上讲，上述方法等价于将顶点 P 转换至 OBB 的局部坐标系统中，并计算 OBB 上距该转换点的最近点，最后将结果转换回世界坐标系统。

5.1.4.1　点到 OBB 的距离

计算点 P 与 OBB 上最近点之间的平方距离时，前述函数可以按照下列方式加以调用：

```
// Computes the square distance between point p and OBB b
float SqDistPointOBB(Point p, OBB b)
{
    Point closest;
    ClosestPtPointOBB(p, b, closest);
    float sqDist = Dot(closest - p, closest - p);
    return sqDist;
}
```

仅当计算平方距离而无须获取最近点时，该代码还可以进一步简化。通过将中心 B 与点 P 形成的向量 **v** 投影至 OBB 各自的 3 个轴上，即可获得各轴上的点 P 距中心的距离 d。由于各轴间正交，对于一个给定轴，可以计算出 d 相对于包围盒范围的超额部分，随后获取其平方值并累加至点 P 的距离平方值上，上述计算皆独立于其他两轴。

```
// Computes the square distance between point p and OBB b
float SqDistPointOBB(Point p, OBB b)
{
    Vector v = p - b.c;
    float sqDist = 0.0f;
    for (int i = 0; i < 3; i++) {
        // Project vector from box center to p on each axis, getting the distance
        // of p along that axis, and count any excess distance outside box extents
        float d = Dot(v, b.u[i]), excess = 0.0f;
        if (d < -b.e[i])
            excess = d + b.e[i];
        else if (d > b.e[i])
            excess = d - b.e[i];
        sqDist += excess * excess;
    }
    return sqDist;
}
```

5.1.4.2 点至 3D 矩形的最近点

对于给定点 P，计算 3D 矩形 R 上的最近点 Q 实际上等同于计算 OBB 上的最近点，其中 3D 矩形可看作是 z 向为 0 的 OBB。

由此，矩形定义为：中心点 C、确定 R 中 x、y 轴的正交单位向量 $\mathbf{u}_0$ 和 $\mathbf{u}_1$，以及用以确定矩形半轴长的两个标量 e_0 和 e_1。在该表达方式中，包含于 R 的点集 S 定义为 $S= C + a\mathbf{u}_0 + b\mathbf{u}_1$，其中$|a|\leqslant e_0$，$|b|\leqslant e_1$。该矩形的数据结构表示为：

```
struct Rect {
    Point c;            // center point of rectangle
    Vector u[2];        // unit vectors determining local x and y axes for the rectangle
    float e[2];         // the halfwidth extents of the rectangle along the axes
};
```

令 z 轴为 0 并重写函数 **ClosestPtPointOBB()**，则 3D 矩形最近点计算如下所示：

```
// Given point p, return point q on (or in) Rect r, closest to p
void ClosestPtPointRect(Point p, Rect r, Point &q)
{
    Vector d = p - r.c;
    // Start result at center of rect; make steps from there
    q = r.c;
    // For each rect axis...
    for (int i = 0; i < 2; i++) {
        // ...project d onto that axis to get the distance
        // along the axis of d from the rect center
        float dist = Dot(d, r.u[i]);
        // If distance farther than the rect extents, clamp to the rect
        if (dist > r.e[i]) dist = r.e[i];
        if (dist < -r.e[i]) dist = -r.e[i];
        // Step that distance along the axis to get world coordinate
        q += dist * r.u[i];
    }
}
```

3D 矩形也可以通过 3 个点（A、B、C）加以定义，则向量 $B - A$ 和 $C - A$ 跨越该矩形。R 中的所有顶点定义为 $S=A + u(B - A) + v(C - A)$，其中 $0\leqslant u\leqslant 1$，$0\leqslant v\leqslant 1$。在此方案中，仍可采用类似的投影方法（但需对新的限定区间加以调整）。其优化的实现代码如下所示：

```
// Return point q on (or in) rect (specified by a, b, and c), closest to given point p
void ClosestPtPointRect(Point p, Point a, Point b, Point c, Point &q)
{
    Vector ab = b - a;          // vector across rect
    Vector ac = c - a;          // vector down rect
    Vector d = p - a;
    // Start result at top-left corner of rect; make steps from there
    q = a;
    // Clamp p' (projection of p to plane of r) to rectangle in the across direction
    float dist = Dot(d, ab);
```

```
    float maxdist = Dot(ab, ab);
    if (dist >= maxdist)
        q += ab;
    else if (dist > 0.0f)
        q += (dist / maxdist) * ab;
    // Clamp p' (projection of p to plane of r) to rectangle in the down direction
    dist = Dot(d, ac);
    maxdist = Dot(ac, ac);
    if (dist >= maxdist)
        q += ac;
    else if (dist > 0.0f)
        q += (dist / maxdist) * ac;
}
```

与最初的方案相比，由于未曾受益于预计算过程中 *ab*、*ac* 的单位向量，该函数的耗费稍显庞大。

5.1.5　点至三角形的最近点

给定三角形 *ABC* 及一点 *P*，且令 *ABC* 上距 *P* 的最近点为 *Q*。一种解决方案是：如果 *P* 正交投影后位于三角形内部，则投影点即为最近点 *Q*；如果 *P* 的正交投影位于三角形 *ABC* 的外部，则最近点位于三角形的某一条边上。此时，需要遍历每一条边（*AB*、*BC*、*CA*），计算并返回 *P* 的最近点。虽然这可以得到相应的计算结果，但并不是一种高效的解决方案。一种较好的算法是：考查 *P* 位于三角形的哪一个 Voronoi 特征域中。一旦确定，只须将 *P* 正交投影至该特征域中即可获取最近点 *Q*。

下面考查 *P* 位于包含顶点 *A* 的 Voronoi 域的计算过程。该域由穿越 *A* 的两个平面的负半空间的交集构成，其法线分别为 $B-A$ 和 $C-A$（如图 5.5 所示）。

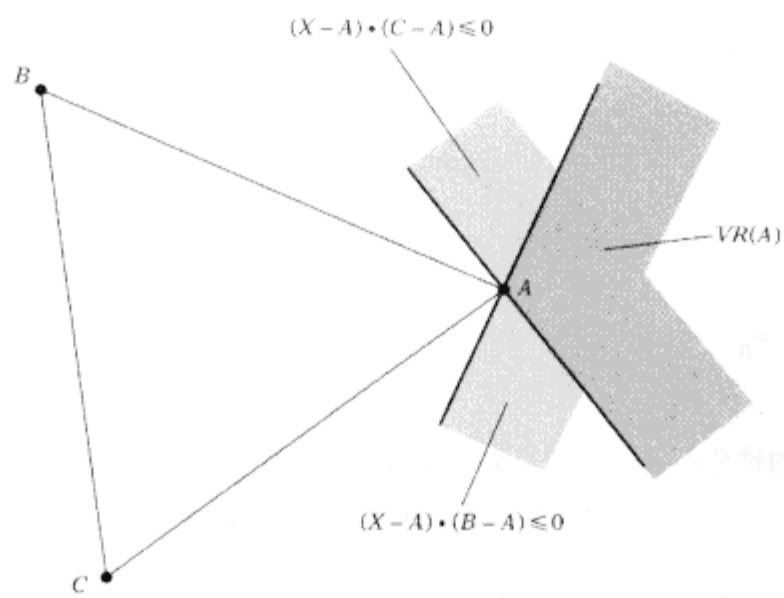


图 5.5　顶点 *A* 的 Voronoi 域 *VR*(A)可视为平面$(X-A)\cdot(B-A)=0$ 和$(X-A)\cdot(C-A)=0$ 负半空间的交集

可以采取多种方法确定 *P* 位于某一边的 Voronoi 域。由前述内容可知，一种有效的测试方案是：计算 *P* 在 *ABC* 上正交投影的质心坐标。回忆一下 3.4 节，*R* 的质心坐标为三角形 *RAB*、*RBC* 和 *RCA* 的有符号面积与三角形 *ABC* 有符号面积之比。令 **n** 为三角形 *ABC* 的法线，且对于某 *t* 值，有 $R=P-t\mathbf{n}$，

则 $R = uA + vB + wC$ 的质心坐标(u, v,w)为：$u = rbc/abc$，$v = rca/abc$ 以及 $w= rab/abc$。通过某些向量计算，可以达到一定程度上的简化目的。例如，表达式 rab 可以按如下方式进行简化：

```
Vector n = Cross(b - a, c - a);
float rab = Dot(n, Cross(a - r, b - r));        // proportional to signed area of RAB
float rbc = Dot(n, Cross(b - r, c - r));        // proportional to signed area of RBC
float rca = Dot(n, Cross(c - r, a - r));        // proportional to signed area of RCA
float abc = rab + rbc + rca;                    // proportional to signed area of ABC
```

$$\mathbf{n} \cdot ((A - R) \times (B - R)) = \quad \text{（初始表达式）}$$

$$\mathbf{n} \cdot (A \times (B - R) - R \times (B - R)) = \quad \text{（叉积展开）}$$

$$\mathbf{n} \cdot (A \times B - A \times R - R \times B) = \quad \text{（展开叉积；移除 } \mathbf{n} \cdot (R \times R) = 0 \text{ 项）}$$

$$\mathbf{n} \cdot (A \times B - A \times (P - t\mathbf{n}) - (P - t\mathbf{n}) \times B) = \quad \text{（将 } R = P - t\mathbf{n} \text{ 代入）}$$

$$\mathbf{n} \cdot (A \times B - A \times P + tA \times \mathbf{n} - P \times B + t\mathbf{n} \times B) = \quad \text{（叉积展开）}$$

$$\mathbf{n} \cdot (A \times B - A \times P - P \times B + t\mathbf{n} \times (B - A)) = \quad \text{（合并同类项）}$$

$$\mathbf{n} \cdot (A \times B - A \times P - P \times B) = \quad \text{（移除 } \mathbf{n} \cdot (t\mathbf{n} \times (B - A)) = 0 \text{ 项）}$$

$$\mathbf{n} \cdot ((A - P) \times (B - P)) \quad \text{（加入 } \mathbf{n} \cdot (P \times P) = 0 \text{ 项后获取叉积）}$$

即 R 的质心坐标可以直接通过 P 获得，而无须计算 R。

对于位于边 Voronoi 域中的 P 而言，由于 rab≤0，对应边 AB-P 位于 AB 的外部或之上；同时，还位于平面$(X - A) \cdot (B - A) = 0$ 及$(X - B) \cdot (A - B) = 0$ 正半空间之内。需注意的是，仅计算 P 是否位于 AB 的外部将导致不充分的测试结果。例如，对于一个在点 A 处为钝角的三角形，P 位于 AB 的外部却位于边 CA 的 Voronoi 域内（如图 5.6 所示，类似地一类常见错误例如，仅判定 P 位于 AB 外部且$(P - A) \cdot (B - A) < 0$，就得出结论：$A$ 为距 P 的最近点）。若点 P 不属于任一顶点或边的 Voronoi 域，则 Q 必位于三角形 ABC 的内部，实际上为正交投影 R。下面即可利用这些信息生成其代码实现。

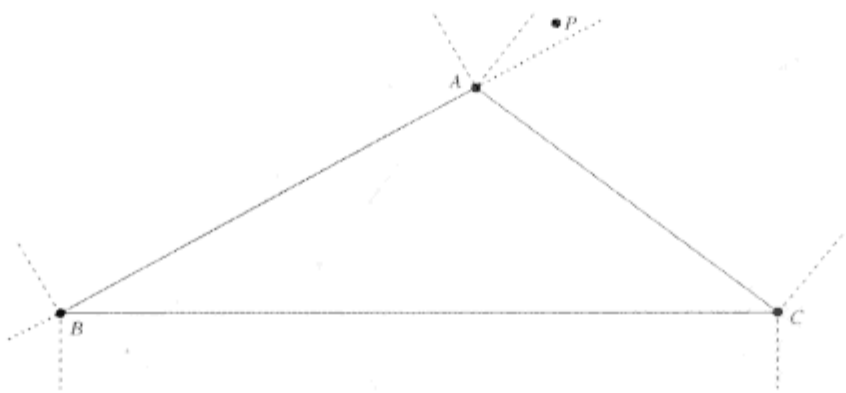


图 5.6　对于点 A 处的钝角而言，即使点 P 位于 AB 边的外部且不位于点 A 或 B 的 Voronoi 内部，点 P 仍有可能在 CA 边的 Voronoi 内

```
Point ClosestPtPointTriangle(Point p, Point a, Point b, Point c)
{
    Vector ab = b - a;
    Vector ac = c - a;
    Vector bc = c - b;

    // Compute parametric position s for projection P' of P on AB,
```

```
    // P' = A + s*AB, s = snom/(snom+sdenom)
    float snom = Dot(p - a, ab), sdenom = Dot(p - b, a - b);

    // Compute parametric position t for projection P' of P on AC,
    // P' = A + t*AC, s = tnom/(tnom+tdenom)
    float tnom = Dot(p - a, ac), tdenom = Dot(p - c, a - c);

    if (snom <= 0.0f && tnom <= 0.0f) return a;          // Vertex region early out

    // Compute parametric position u for projection P' of P on BC,
    // P' = B + u*BC, u = unom/(unom+udenom)
    float unom = Dot(p - b, bc), udenom = Dot(p - c, b - c);

    if (sdenom <= 0.0f && unom <= 0.0f) return b;        // Vertex region early out
    if (tdenom <= 0.0f && udenom <= 0.0f) return c  ;    // Vertex region early out

    // P is outside (or on) AB if the triple scalar product [N PA PB] <= 0
    Vector n = Cross(b - a, c - a);
    float vc = Dot(n, Cross(a - p, b - p));
    // If P outside AB and within feature region of AB,
    // return projection of P onto AB
    if (vc <= 0.0f && snom >= 0.0f && sdenom >= 0.0f)
        return a + snom / (snom + sdenom) * ab;

    // P is outside (or on) BC if the triple scalar product [N PB PC] <= 0
    float va = Dot(n, Cross(b - p, c - p));
    // If P outside BC and within feature region of BC,
    // return projection of P onto BC
    if (va <= 0.0f && unom >= 0.0f && udenom >= 0.0f)
        return b + unom / (unom + udenom) * bc;

    // P is outside (or on) CA if the triple scalar product [N PC PA] <= 0
    float vb = Dot(n, Cross(c - p, a - p));
    // If P outside CA and within feature region of CA,
    // return projection of P onto CA
    if (vb <= 0.0f && tnom >= 0.0f && tdenom >= 0.0f)
        return a + tnom / (tnom + tdenom) * ac;

    // P must project inside face region. Compute Q using barycentric coordinates
    float u = va / (va + vb + vc);
    float v = vb / (va + vb + vc);
    float w = 1.0f - u - v;          // = vc / (va + vb + vc)
    return u * a + v * b + w * c;
}
```

如代码所示，其中调用了 4 次叉积运算。与点积运算相比，叉积运算通常成本较高。因此，采用某种更加经济的表达式是值得考虑的。根据拉格朗日恒等式：

$$(\mathbf{a} \times \mathbf{b}) \cdot (\mathbf{c} \times \mathbf{d}) = (\mathbf{a} \cdot \mathbf{c})(\mathbf{b} \cdot \mathbf{d}) - (\mathbf{a} \cdot \mathbf{d})(\mathbf{b} \cdot \mathbf{c})$$

可表示为 3 个标量三重积：

```
Vector n = Cross(b - a, c - a);
float va = Dot(n, Cross(b - p, c - p));
float vb = Dot(n, Cross(c - p, a - p));
float vc = Dot(n, Cross(a - p, b - p));
```

并利用 6 个点积计算对以下内容进行表达：

```
float d1 = Dot(b - a, p - a);
float d2 = Dot(c - a, p - a);
float d3 = Dot(b - a, p - b);
float d4 = Dot(c - a, p - b);
float d5 = Dot(b - a, p - c);
float d6 = Dot(c - a, p - c);
```

最终有：

```
float va = d3*d6 - d5*d4;
float vb = d5*d2 - d1*d6;
float vc = d1*d4 - d3*d2;
```

实际上，**d1**～**d6** 也可以用于计算 **snom**、**sdenom**、**tnom**、**tdenom**、**unom** 以及 **udenom** 项：

```
float snom = d1;
float sdenom = -d3;
float tnom = d2;
float tdenom = -d6;
float unom = d4 - d3;
float udenom = d5 - d6;
```

其中，向量 **n** 则不再需要。代码优化后的最终版本为：

```
Point ClosestPtPointTriangle(Point p, Point a, Point b, Point c)
{
    // Check if P in vertex region outside A
    Vector ab = b - a;
    Vector ac = c - a;
    Vector ap = p - a;
    float d1 = Dot(ab, ap);
    float d2 = Dot(ac, ap);
    if (d1 <= 0.0f && d2 <= 0.0f) return a;      // barycentric coordinates (1,0,0)

    // Check if P in vertex region outside B
    Vector bp = p - b;
    float d3 = Dot(ab, bp);
    float d4 = Dot(ac, bp);
    if (d3 >= 0.0f && d4 <= d3) return b;        // barycentric coordinates (0,1,0)

    // Check if P in edge region of AB, if so return projection of P onto AB
    float vc = d1*d4 - d3*d2;
    if (vc <= 0.0f && d1 >= 0.0f && d3 <= 0.0f) {
        float v = d1 / (d1 - d3);
```

```
        return a + v * ab;      // barycentric coordinates (1-v,v,0)
    }

    // Check if P in vertex region outside C
    Vector cp = p - c;
    float d5 = Dot(ab, cp);
    float d6 = Dot(ac, cp);
    if (d6 >= 0.0f && d5 <= d6) return c;          // barycentric coordinates (0,0,1)
    // Check if P in edge region of AC, if so return projection of P onto AC
    float vb = d5*d2 - d1*d6;
    if (vb <= 0.0f && d2 >= 0.0f && d6 <= 0.0f) {
        float w = d2 / (d2 - d6);
        return a + w * ac;      // barycentric coordinates (1-w,0,w)
    }

    // Check if P in edge region of BC, if so return projection of P onto BC
    float va = d3*d6 - d5*d4;
    if (va <= 0.0f && (d4 - d3) >= 0.0f && (d5 - d6) >= 0.0f) {
        float w = (d4 - d3) / ((d4 - d3) + (d5 - d6));
        return b + w * (c - b);         // barycentric coordinates (0,1-w,w)
    }

    // P inside face region. Compute Q through its barycentric coordinates (u,v,w)
    float denom = 1.0f / (va + vb + vc);
    float v = vb * denom;
    float w = vc * denom;
    return a + ab * v + ac * w;         // = u*a + v*b + w*c, u = va * denom = 1.0f - v - w
}
```

获取最近点的第 3 种方案为[Eberly01]建议的向量微积分方法。简单地讲，参数化后的三角形 ABC 为 $T(s, t) = A + s(B - A) + t(C - A)$，其中 $s \geqslant 0$，$t \geqslant 0$，$s + t \leqslant 1$。P 的最近点为二次函数 $d(s, t) = \|T(s, t) - P\|^2$ 的最小值，该函数的最小值包含 3 种情形：某一个顶点处、某一条边上以及三角形内部。首先针对上述 3 种情形对 $d(s, t)$进行微分计算（即必要时替换 $s = 0$、$t = 0$ 或 $t = 1 - s$），设定导数为 0 并求解，最后将 s、t 的结果值与三角形边界比较，即可得到最小值与上述 3 种情形所对应的状态。

对于这类特定的应用，向量微积分算法实际操作起来将会更加复杂。然而，探讨二次极小化问题的通用算法仍然是有意义的，同一思想也可用于计算直线或线段与三角形之间的距离之中。向量微积分的其他信息及其伪代码在[Eberly01]中给出。

5.1.6　点到四面体的最近点

给定点 P，当前问题为：计算四面体 $ABCD$ 上（或之内）距点 P 的最近点 Q（如图 5.7 所示）。一种较为直接的算法是：针对四面体中的每一个面，调用函数 **ClosestPtPointTriangle()**（该函数在 5.1.5 节中定义）并计算最近点。在多个计算结果中，返回距点 P 最近的点 Q。独立于距离测试，还需要考查点 P 是否位于四面体中的面内。若点 P 位于四面体某一面内，则其即为最近点。

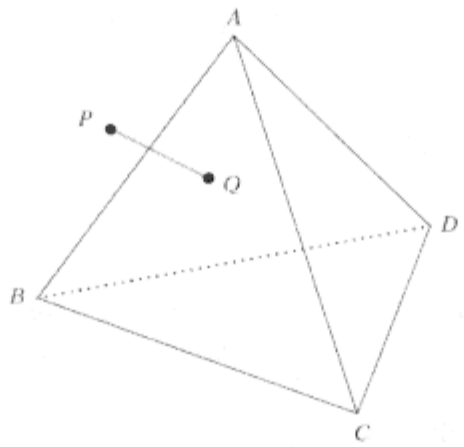


图 5.7　四面体 *ABCD* 上距点 *P* 的最近点 *Q*

现设定四面体 *ABCD*，且从体外查看时，全部面 *ABC*、*ACD*、*ADB* 和 *BDC* 均按逆时针排列，则上述算法实现如下所示：

```
Point ClosestPtPointTetrahedron(Point p, Point a, Point b, Point c, Point d)
{
    // Start out assuming point inside all halfspaces, so closest to itself
    Point closestPt = p;
    float bestSqDist = FLT_MAX;
    // If point outside face abc then compute closest point on abc
    if (PointOutsideOfPlane(p, a, b, c)) {
        Point q = ClosestPtPointTriangle(p, a, b, c);
        float sqDist = Dot(q - p, q - p);
        // Update best closest point if (squared) distance is less than current best
        if (sqDist < bestSqDist) bestSqDist = sqDist, closestPt = q;
    }
    // Repeat test for face acd
    if (PointOutsideOfPlane(p, a, c, d)) {
        Point q = ClosestPtPointTriangle(p, a, c, d);
        float sqDist = Dot(q - p, q - p);
        if (sqDist < bestSqDist) bestSqDist = sqDist, closestPt = q;
    }
    // Repeat test for face adb
    if (PointOutsideOfPlane(p, a, d, b)) {
        Point q = ClosestPtPointTriangle(p, a, d, b);
        float sqDist = Dot(q - p, q - p);
        if (sqDist < bestSqDist) bestSqDist = sqDist, closestPt = q;
    }
    // Repeat test for face bdc
    if (PointOutsideOfPlane(p, b, d, c)) {
        Point q = ClosestPtPointTriangle(p, b, d, c);
        float sqDist = Dot(q - p, q - p);
        if (sqDist < bestSqDist) bestSqDist = sqDist, closestPt = q;
    }
    return closestPt;
}
```

这里，**PointOutsideOfPlane(P, A, B, C)**的返回值对应着向量 $P-A$、$B-A$ 和 $C-A$ 的标量三重积的符号。

```
// Test if point p lies outside plane through abc
int PointOutsideOfPlane(Point p, Point a, Point b, Point c)
{
    return Dot(p - a, Cross(b - a, c - a)) >= 0.0f;        // [AP AB AC] >= 0
}
```

通常情况下，四面体顶点的环绕方向事先难以预料。例如，当从四面体外部查看时，无法保证面 *ABC* 顶点总是按逆时针旋转。此时，需要向函数 **PointOutsideOfPlane()**传递第 4 个顶点，并确定该顶点与测试顶点是否位于四面体测试面的两侧。函数实现如下所示：

```
// Test if point p and d lie on opposite sides of plane through abc
int PointOutsideOfPlane(Point p, Point a, Point b, Point c, Point d)
{
    float signp = Dot(p - a, Cross(b - a, c - a));        // [AP AB AC]
    float signd = Dot(d - a, Cross(b - a, c - a));        // [AD AB AC]
    // Points on opposite sides if expression signs are opposite
    return signp * signd < 0.0f;
}
```

上述算法运行良好且易于实现。然而，通过采用计算三角形最近点时所使用的基础算法，上述算法仍存在改进的余地。首先，假定顶点 P 的 Voronoi 特征域存在，则 Q 为 P 在该特征上的正交投影。针对四面体，需要考查总计 14 个特征域：4 个顶点域、6 个边域以及 4 个面域。如果 P 不属于任何一个特征域，则必定位于四面体内部。四面体特征域的包含测试类似于前述三角形测试。例如，如果满足下列表达式，则 P 位于顶点 A 的 Voronoi 域：

$$(P-A)\cdot(B-A)\leqslant 0$$
$$(P-A)\cdot(C-A)\leqslant 0$$
$$(P-A)\cdot(D-A)\leqslant 0$$

点 P 位于关联 AB 边的 Voronoi 域中时，应满足下列测试条件（假定面呈逆时针环绕）：

$$(P-A)\cdot(B-A)\geqslant 0$$
$$(P-B)\cdot(A-B)\geqslant 0$$
$$(P-A)\cdot((B-A)\times \mathbf{n}_{ABC})\geqslant 0，其中\ \mathbf{n}_{ABC}=(B-A)\times(C-A)$$
$$(P-A)\cdot(\mathbf{n}_{ADB}\times(B-A))\geqslant 0，其中\ \mathbf{n}_{ADB}=(D-A)\times(B-A)$$

类似的表达式集合也可用于定义其他域的包含测试。与前述测试方法相比，该算法似乎无明显的性能改观。然而，Voronoi 域间却可以共享多数计算过程而无须重复计算。当然，可以利用拉格朗日恒等式来简化 Voronoi 域测试过程中的相关表达式——类似于三角形最近点测试时的优化方法，即全部测试可以采用 10 个不同的点积值加以构造。

5.1.7　点到凸多面体的最近点

对于空间内一点 P，存在多种算法确定凸多面体 H 上距 P 的最近点。一种易于实现的、复杂度为

$O(n)$的算法是：计算多面体上每一个面中距 P 的最近点，并返回一个最近结果。同时，还要兼顾测试点 P 是否位于 H 各面的背面，即 P 位于 H 内部这一情况。为了加速测试过程，面距离测试计算规定：点 P 位于各面的正面。

对于较大的多面体，一种较为快速的预计算方法为：以层次结构的方式构建多面体的各部分。利用这种预构造的层级方案将使最近点计算的时间复杂度达到 log 级别。Dobkin-Kirkpatrick 层次结构算法便是其中一例，并在第 9 章加以描述。第 9 章还讨论了高效计算多面体最近点的其他一些算法（如 GJK 算法）。

5.1.8 两条直线间的最近点

在 2D 空间中，两条直线如果不平行则必定相交，在 3D 空间中则未必。进一步讲，在严格的实数运算过程中即使两直线相交，但由于浮点计算舍入误差等因素，结果也并非如此。因此，为了保证 3D 空间内两直线相交测试的健壮性，较好的方案是：若两直线之间彼此未达到足够接近，则设定为不相交。基于此，相交测试转化为计算两直线之间的距离，并检测该距离是否小于给定的阀值。

两直线间的最近点可按如下方式计算：给定直线 L_1 和 L_2，且由顶点 P_1、Q_1 和顶点 P_2、Q_2 定义的直线方程为：

$$L_1(s) = P_1 + s\mathbf{d}_1,\ \mathbf{d}_1 = Q_1 - P_1$$

$$L_2(t) = P_2 + t\mathbf{d}_2,\ \mathbf{d}_2 = Q_2 - P_2$$

针对 s 和 t 的几组计算结果，$L_1(s)$和 $L_2(t)$分别对应直线上的最近点，并且 $\mathbf{v}(s, t) = L_1(s) - L_2(t)$定义了二者间的一个向量（如图 5.8 所示）。当 $\mathbf{v}$ 具有最小长度值时，两顶点则为最近点。因此，核心步骤为：上述过程形成于 $\mathbf{v}$ 垂直于 L_1 和 L_2 处。为了分析该计算过程，将考查一点 P 和直线 L 之间的最近距离，即点 P 和点 Q 构成的线段长度，其中点 Q 为点 P 在直线 L 上的正交投影。

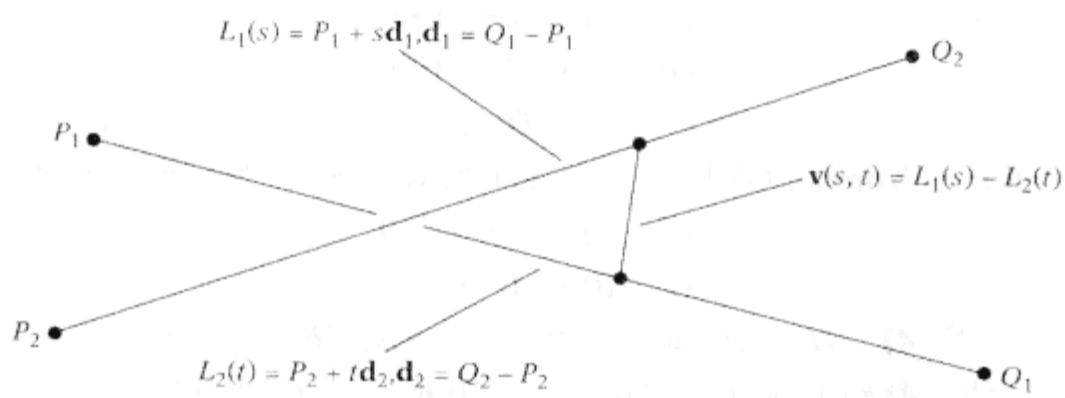


图 5.8　向量 $\mathbf{v}(s, t)$连接直线 $L_1(s)$和 $L_2(t)$上的两个最近点，且分别垂直于两条直线

其结果为：线段 PQ 垂直于直线 L。由于上述推导过程涉及与 L_2 相关的 $L_1(s)$以及与 L_1 相关的 $L_2(t)$，$\mathbf{v}$ 必定垂直于两条直线。对于非平行的直线，$\mathbf{v}$ 唯一存在。

上述问题演变为计算 s、t 值，且满足下面两个垂直约束条件：

$$\mathbf{d}_1 \cdot \mathbf{v}(s, t) = 0$$

$$\mathbf{d}_2 \cdot \mathbf{v}(s, t) = 0$$

利用参数方程替换上式中的 $\mathbf{v}(s, t)$：

$$\mathbf{d}_1 \cdot (L_1(s) - L_2(t)) = \mathbf{d}_1 \cdot ((P_1 - P_2) + s\mathbf{d}_1 - t\mathbf{d}_2) = 0$$
$$\mathbf{d}_2 \cdot (L_1(s) - L_2(t)) = \mathbf{d}_2 \cdot ((P_1 - P_2) + s\mathbf{d}_1 - t\mathbf{d}_2) = 0$$

且可以表达为 2 × 2 线性方程组，其中 $\mathbf{r} = P_1 - P_2$：

$$(\mathbf{d}_1 \cdot \mathbf{d}_1)s - (\mathbf{d}_1 \cdot \mathbf{d}_2)t = -(\mathbf{d}_1 \cdot \mathbf{r})$$
$$(\mathbf{d}_2 \cdot \mathbf{d}_1)s - (\mathbf{d}_2 \cdot \mathbf{d}_2)t = -(\mathbf{d}_2 \cdot \mathbf{r})$$

符号化为矩阵格式则记作：

$$\begin{bmatrix} a & -b \\ b & -e \end{bmatrix}\begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} -c \\ -f \end{bmatrix}$$

其中 $a = \mathbf{d}_1 \cdot \mathbf{d}_1$，$b = \mathbf{d}_1 \cdot \mathbf{d}_2$，$c = \mathbf{d}_1 \cdot \mathbf{r}$，$e = \mathbf{d}_2 \cdot \mathbf{d}_2$，$f = \mathbf{d}_2 \cdot \mathbf{r}$。该方程组可利用克莱姆法则求解：

$$s = (bf - ce)/d$$
$$t = (af - bc)/d$$

其中 $d = ae - b^2$。注意，$d \geqslant 0$ 且 $d = \|\mathbf{d}_1\|^2 \|\mathbf{d}_2\|^2 - (\|\mathbf{d}_1\| \|\mathbf{d}_2\| \cos(\theta))^2 = (\|\mathbf{d}_1\| \|\mathbf{d}_2\| \sin(\theta))^2$。当 $d=0$ 时，两直线平行且需要另行处理。此时，可以选择一条直线上的任一点；在另一条直线上，可以采用 5.1.2 节中介绍的投影算法选取 P 的最近点。

5.1.9　两线段上的最近点

与计算直线 L_1 和 L_2 的最近点相比，作为直线某一部分的线段 S_1 和 S_2，其最近点计算更加复杂。若直线 L_1 和 L_2 间的最近点恰好位于对应线段上，则可以应用相关的直线最近点算法。而对于直线 L_1、L_2 间最近点位于线段外部延长线这类情况，一种常见的错误是将外部最近点截取为距其最近的线段端点。然而，如图 5.9（b）和图 5.9（c）所示，这将是一类错误的假设方案。

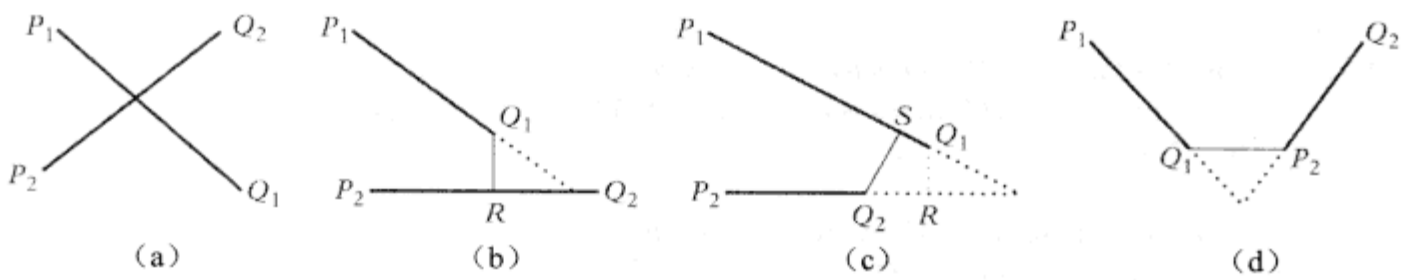


图 5.9　最近点位置。（a）最近点位于两线段内部。（b）、（c）最近点分别为线段内部一点以及另一线段的端点。（d）两线段的端点

$$S_1(s) = P_1 + s\mathbf{d}_1,\ \mathbf{d}_1 = Q_1 - P_1,\ 0 \leqslant s \leqslant 1$$
$$S_2(t) = P_2 + t\mathbf{d}_2,\ \mathbf{d}_2 = Q_2 - P_2,\ 0 \leqslant t \leqslant 1$$

不难看出，在某些情况下，若直线间某一最近点位于相关线段的外部延长线上，该点是可以截取至相应线段的最近端点处的（[Lumelsky85]）。如图 5.9（b）所示，其中 L_1 上的最近点截取为线段 S_1 的端点 Q_1。于是，计算 L_2 上距 Q_1 的最近点 R 且该点位于线段 S_2 上，则线段间的最近点为 Q_1 和 R。

若直线间的最近点皆位于各自线段的外部延长线上，则上述截取操作需要重复计算两次，如图 5.9（c）所示。L_1 上的最近点截取为线段 S_1 的端点 Q_1，计算 L_2 上距 Q_1 的最近点 R。由于 R 位于线段 S_2 的外部延长线上，将其截取为线段的最近端点 Q_2；再次计算 L_1 上距 Q_2 的最近点 S 且该点位于线段 S_1 上，则

线段间的最近点为 Q_2 和 S。

给定直线上 L_2 一点 $S_2(t) = P_2+t\mathbf{d}_2$，则 L_1 上最近点 $L_1(s)$为：

$$s = (S_2(t) - P_1) \cdot \mathbf{d}_1/\mathbf{d}_1 \cdot \mathbf{d}_1=(P_2+t\mathbf{d}_2 - P_1) \cdot \mathbf{d}_1/\mathbf{d}_1 \cdot \mathbf{d}_1$$

类似地，给定直线 S_1 上一点 $S_1(s) = P_1+s\mathbf{d}_1$，直线 L_2 上最近点 $L_2(t)$为：

$$t = (S_1(s) - P_2) \cdot \mathbf{d}_2/\mathbf{d}_2 \cdot \mathbf{d}_2= (P_1+ s\mathbf{d}_1- P_2) \cdot \mathbf{d}_2/\mathbf{d}_2 \cdot \mathbf{d}_2$$

另外，可以采用高斯消元法求解 2 × 2 线性方程组。因而，s、t 表达式可简化为：

$$s = (bt - c)/a$$

$$t = (bs + f)/e$$

其中，$a = \mathbf{d}_1 \cdot \mathbf{d}_1$，$b = \mathbf{d}_1 \cdot \mathbf{d}_2$，$c = \mathbf{d}_1 \cdot \mathbf{r}$，$e = \mathbf{d}_2 \cdot \mathbf{d}_2$，$f = \mathbf{d}_2 \cdot \mathbf{r}$。

代码实现如下所示：

```
// Clamp n to lie within the range [min, max]
float Clamp(float n, float min, float max) {
    if (n < min) return min;
    if (n > max) return max;
    return n;
}

// Computes closest points C1 and C2 of S1(s)=P1+s*(Q1-P1) and
// S2(t)=P2+t*(Q2-P2), returning s and t. Function result is squared
// distance between between S1(s) and S2(t)
float ClosestPtSegmentSegment(Point p1, Point q1, Point p2, Point q2,
                              float &s, float &t, Point &c1, Point &c2)
{
    Vector d1 = q1 - p1;        // Direction vector of segment S1
    Vector d2 = q2 - p2;        // Direction vector of segment S2
    Vector r = p1 - p2;
    float a = Dot(d1, d1);      // Squared length of segment S1, always nonnegative
    float e = Dot(d2, d2);      // Squared length of segment S2, always nonnegative
    float f = Dot(d2, r);

    // Check if either or both segments degenerate into points
    if (a <= EPSILON && e <= EPSILON) {
        // Both segments degenerate into points
        s = t = 0.0f;
        c1 = p1;
        c2 = p2;
        return Dot(c1 - c2, c1 - c2);
    }
    if (a <= EPSILON) {
        // First segment degenerates into a point
        s = 0.0f;
        t = f / e;                      // s = 0 => t = (b*s + f) / e = f / e
        t = Clamp(t, 0.0f, 1.0f);
    } else {
        float c = Dot(d1, r);
        if (e <= EPSILON) {
```

```
        // Second segment degenerates into a point
        t = 0.0f;
        s = Clamp(-c / a, 0.0f, 1.0f);        // t = 0 => s = (b*t - c) / a = -c / a
    } else {
        // The general nondegenerate case starts here
        float b = Dot(d1, d2);
        float denom = a*e-b*b;                // Always nonnegative

        // If segments not parallel, compute closest point on L1 to L2 and
        // clamp to segment S1. Else pick arbitrary s (here 0)
        if (denom != 0.0f) {
            s = Clamp((b*f - c*e) / denom, 0.0f, 1.0f);
        } else s = 0.0f;
        // Compute point on L2 closest to S1(s) using
        // t = Dot((P1 + D1*s) - P2,D2) / Dot(D2,D2) = (b*s + f) / e
        t = (b*s + f) / e;

        // If t in [0,1] done. Else clamp t, recompute s for the new value
        // of t using s = Dot((P2 + D2*t) - P1,D1) / Dot(D1,D1)= (t*b - c) / a
        // and clamp s to [0, 1]
        if (t < 0.0f) {
            t = 0.0f;
            s = Clamp(-c / a, 0.0f, 1.0f);
        } else if (t > 1.0f) {
            t = 1.0f;
            s = Clamp((b - c) / a, 0.0f, 1.0f);
        }
        }
    }

    c1 = p1 + d1 * s;
    c2 = p2 + d2 * t;
    return Dot(c1 - c2, c1 - c2);
}
```

作为一种优化方案，变量 e 的除法操作可延至已知 t 位于[0,1]时，则上述代码中较大的 else 语句可改写为：

```
...
float tnom = b*s + f;
if (tnom < 0.0f) {
    t = 0.0f;
    s = Clamp(-c / a, 0.0f, 1.0f);
} else if (tnom > e) {
    t = 1.0f;
    s = Clamp((b - c) / a, 0.0f, 1.0f);
} else {
    t = tnom / e;
}
```

一般情况下，这类延迟方案可以缓解除法操作的高昂成本。

5.1.9.1 2D 线段相交计算

测试两条线段 AB、CD 是否相交，可以首先计算二者（延长线）的交点，且查看该交点是否位于各自线段的包围盒中，如图 5.10（a）和图 5.10（b）所示。需注意的是，需要单独处理两线段平行这一特例。

计算两线段（延长线）交点时，显式定义直线 L_1 的方程 $L_1(t) = A + t(B - A)$，并隐式定义另一直线方程 $\mathbf{n} \cdot (X - C) = 0$，其中 $\mathbf{n} = (D - C)^{\perp}$ 并垂直于 CD。利用 $A + t(B - A)$带入方程 $\mathbf{n} \cdot (X - C) = 0$ 并求解 t。如下所示：

$$\mathbf{n} \cdot (A + t(B - A) - C) = 0 \Leftrightarrow \text{（利用 } A + t(B - A) \text{替换 } X\text{）}$$

$$\mathbf{n} \cdot (A - C) + t\,(\mathbf{n} \cdot (B - A)) = 0 \Leftrightarrow \text{（扩展点积；合并同类项）}$$

$$t(\mathbf{n} \cdot (B - A)) = -\mathbf{n} \cdot (A - C) \Leftrightarrow \text{（基于 } LHS \text{ 提取 } t\text{）}$$

$$t(\mathbf{n} \cdot (B - A)) = \mathbf{n} \cdot (C - A) \Leftrightarrow \text{（逆置向量）}$$

$$t = \mathbf{n} \cdot (C - A)/\mathbf{n} \cdot (B - A) \quad \text{（两边同时除以 } \mathbf{n} \cdot (B - A)\text{）}$$

则交点 $P = L_1(t)$可将 t 值代入上述显式方程得到。另外，可通过检测 $0 \leqslant t \leqslant 1$ 从而确定点 P 是否位于 AB 的包围盒内。

另一种 2D 线段相交测试方案为：两线段相交仅当两端点彼此位于线段的两侧。例如，若线段 AB 和 CD 相交，端点 A、B 定位于线段 CD 的两侧；同时，端点 C、D 也定位于线段 AB 的两侧。

测试端点 C、D 是否位于线段 AB 的两侧时，可以通过判断三角形 ABD 和 ABC 的环绕方向加以解决，如图 5.10（c）所示。而三角形的有符号面积则可以确定其环绕方向。回忆一下，若三角形顶点呈逆时针旋转，则其有符号面积为正值；若逆时针旋转，则为负值；若三角形退化为共线顶点或一点，则为 0。

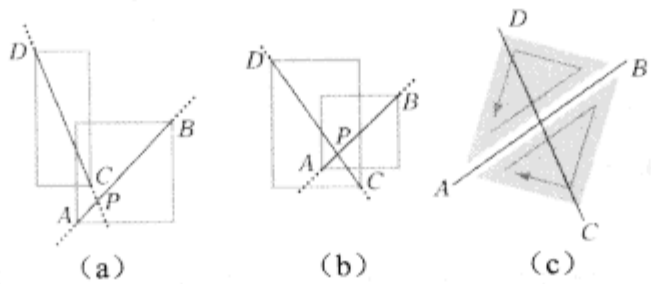


图 5.10　（a）由于线段 AB 和 CD 延长线的交点位于包围盒 CD 的外部，因而两线段不相交。（b）由于点 P 位于 AB 和 CD 的包围盒内部，因而两线段相交。（c）由于三角形 ABD 和 ABC 的环绕方向相反，因而线段 CD 与直线 AB 相交

```
// Returns 2 times the signed triangle area. The result is positive if
// abc is ccw, negative if abc is cw, zero if abc is degenerate.
float Signed2DTriArea(Point a, Point b, Point c)
{
    return (a.x - c.x) * (b.y - c.y) - (a.y - c.y) * (b.x - c.x);
}
```

当线段相交时，检测过程所用到的有符号面积也可以用于计算相交点，其计算推导过程如下所示：

```
// Test if segments ab and cd overlap. If they do, compute and return
// intersection t value along ab and intersection position p
int Test2DSegmentSegment(Point a, Point b, Point c, Point d, float &t, Point &p)
{
    // Sign of areas correspond to which side of ab points c and d are
    float a1 = Signed2DTriArea(a, b, d);        // Compute winding of abd (+ or -)
    float a2 = Signed2DTriArea(a, b, c);        // To intersect, must have sign opposite of a1

    // If c and d are on different sides of ab, areas have different signs
    if (a1 * a2 < 0.0f) {
        // Compute signs for a and b with respect to segment cd
        float a3 = Signed2DTriArea(c, d, a);        // Compute winding of cda (+ or -)
        // Since area is constant a1 - a2 = a3 - a4, or a4 = a3 + a2 - a1
        float a4 = Signed2DTriArea(c, d, b);        // Must have opposite sign of a3
        float a4 = a3 + a2 - a1;
        // Points a and b on different sides of cd if areas have different signs
        if (a3 * a4 < 0.0f) {
            // Segments intersect. Find intersection point along L(t) = a + t * (b - a).
            // Given height h1 of an over cd and height h2 of b over cd,
            // t = h1 / (h1 - h2) = (b*h1/2) / (b*h1/2 - b*h2/2) = a3 / (a3 - a4),
            // where b (the base of the triangles cda and cdb, i.e., the length
            // of cd) cancels out.
            t = a3 / (a3 - a4);
            p = a + t * (b - a);
            return 1;
        }
    }
    // Segments not intersecting (or collinear)
    return 0;
}
```

这里，**a1 * a2 < 0.0f** 语句用于判断 **a1** 和 **a2** 间的符号是否相同（**a3**、**a4** 同理）。当变量为有符号整数时，可采用异或操作 **a1^a2** < 0 而非乘法计算，从而避免潜在的上溢问题。另外，对于共线顶点问题，为了实现较好的测试过程，**a1**、**a2** 应按如下方式对 0 值加以判断：

```
if (a1 != 0.0f && a2 != 0.0f && a1*a2 < 0.0f) ...        // for floating-point variables
if ((a1 | a2) != 0 && a1^a2 < 0) ...                     // for integer variables
```

最后，需要注意的是，对于某些应用，需要在执行线段相交测试之前先处理线段 *AB* 和 *CD* 的包围盒相交测试。对于 3D 空间内成本高昂的线段相交测试，这一点尤为重要。

5.1.10　线段和三角形最近点

线段 *PQ* 和三角形 *ABC* 之间的最近点不必是唯一的。当线段平行于三角形时，则存在无限多个等距离的最近点。这里，不考虑线段与三角形平面平行与否，通常总是可以在线段的一个端点与三角形内部之间确定一个最近距离，如图 5.11（a）所示；或者在线段与三角形某一边之间确定一个最近距离，如图 5.11（b）所示。

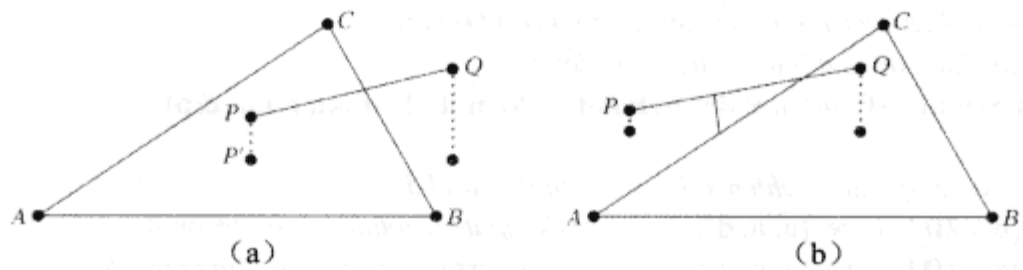


图 5.11　线段和三角形之间最近点组合的实现方法。（a）线段端点和三角形内部顶点。（b）两点分别位于线段和三角形的某一边上

仅当线段端点在三角形支撑面的投影位于三角形内部时，将产生如图 5.11（a）中所示的情况。但是，即使线段端点的投影已位于三角形内部，线段与三角形之间的最近点却有可能出现在三角形的某一边上。因此，最近点计算涉及下列对象：

- ☑ 线段 PQ 与三角形边 AB。
- ☑ 线段 PQ 与三角形边 BC。
- ☑ 线段 PQ 与三角形边 CA。
- ☑ 线段端点 P 与三角形平面（P 的投影位于三角形 ABC 内）。
- ☑ 线段端点 Q 与三角形平面（Q 的投影位于三角形 ABC 内）。

最终，上述计算返回具有唯一最小距离的最近点作为结果。

在某些情况下，可适当地调整测试的数量以减少计算量。例如，若线段的两端点投影皆位于三角形内部，则无须再执行线段-边的最近点测试，因为最近点必然产生于线段的某一投影点中；若只是线段的一个端点投影位于三角形内部，则只须执行一次相应的线段-边测试；当线段的端点投影皆位于三角形外部，也只须执行一次相应的线段-边测试。对于后两种情况，可以通过计算线段端点所处的 Voronoi 域确定相关的线段-边测试。

最后一种情况是线段与三角形相交。对于这类穿越相交状况，交点即是最近点。当然，如果线段位于三角形平面内部，则线段上与三角形相交的任一点皆为最近点。另外，[Eberly01]和[Schneider02]中分别介绍了基于向量微积分的线段-三角形之间的最近点算法。

5.1.11　两个三角形之间的最近点计算

不仅局限于线段-三角形之间的最近点测试，在两个三角形之间也存在无限多个等距离的最近点。两个三角形 T_1 和 T_2 之间的最近点计算一般可认为：相应最近点位于某一个三角形的边上。因此，其结果转化为：可对两三角形中 6 条边之间的可能组合，执行线段-三角形的最近点计算。则包含最小（平方）距离的一组点即为最近点集合。

需注意的是，线段-三角形之间的距离测试成本较高。一种较好的三角形 T_1、T_2 间的最近点集测试方案为：一组最近点产生于两个三角形的相应边上，如图 5.12（a）所示；或产生于某一顶点与另一个三角形的内部点之间，如图 5.12（b）所示。问题转化为：计算两三角形间一对边的最近距离，以及顶点与三角形之间的最近点（相应顶点的投影位于三角形内部）。综上所述，需要执行 6 次顶点-三角形测试以及 9 次边-边测试。最后，从多组最近点中，选择具有最小距离的一组顶点作为最终三角形间的最近点。

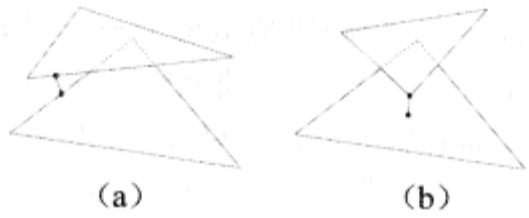


图 5.12　两三角形之间最近点组合的实现方法。（a）一组最近点产生于两个三角形的相应边上。（b）产生于某一顶点与另一个三角形的内部点之间

若已知三角形之间重合，则额外的测试过程将剔除这两个三角形间的相交计算。这是因为，若执行相应的相交测试，两个三角形之间的距离约为 0，由于存在无限多个交点，则最近点无法得到正确的定义（如三角形之间平行以及重合）。

5.2　图元测试

与距离计算相比，图元间的测试应用并不是十分广泛。一般情况下，这类测试只返回图元之间是否相交，而不包含相交位置以及方式。由于不包含额外的数据信息，因而其测试速度较快。

5.2.1　分离轴测试

针对各类相交测试，一种十分有效的工具是分离轴测试。分离轴测试源于凸体分析基本原理——分离超平面理论。该理论表明：给定两个凸体集 A 和 B，若两个集合不存在交集，则必存在一个分离超平面 P，使得 A 和 B 分别位于该超平面的两侧。

分离轴测试则是上述理论的直观体现，由于两个凸体彼此之间无法实现环绕曲合，因此，若凸体间不相交则必存在一个间隙并可以插入一个平面，从而分离两个凸体对象。若物体对象间存在凹体，一般情况下，平面不足以分离这两个非相交物体对象，且需要采用相应的曲面实现物体对象间的分离。若物体对象之间相交，则无论平面还是曲面都无法进行插入操作以实现对象间的分离。

给定一个超平面 P 且分离两个凸体集 A 和 B，则直线分离轴 L 垂直于 P。之所以称之为分离轴，是因为 A 和 B 在 L 上的正交投影生成了两个非重叠的区间（如图 5.13 所示）。由于两个区间不存在交集，其结论为：几何体间彼此独立。当且仅当超平面存在时分离轴存在，且二者都可作为相应的测试平台。然而在实际操作中，较好的方法是采用成本低廉的分离轴测试。

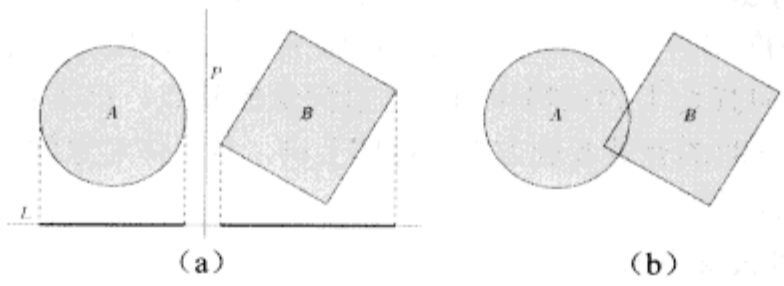


图 5.13　（a）凸体对象 A 和 B 由超平面 P（众多可选超平面中的一个）分开，也可描述为：A 和 B 在其分离轴 L（垂直于 P）投影上处于分离状态。（b）相同的对象处于相交状态，且不存在超平面使其处于分离状态

针对分离轴测试，值得注意的是，多种碰撞检测图元——线段、AABB、OBB、k-DOP 以及球体——通常为对称几何体，其包含中心 C 且为投影轴上投影区间的中心位置。给定分离轴 L，两个对称物体对象 A 和 B 之间的高效分离测试则转换为：计算其投影区间的 1/2 宽度或半径，求和并与二中心投影间的距离进行比较，若和值小于中心投影之间的距离，则物体对象间处于分离状态。相应的示例如图 5.14 所示，其中 A 和 B 分别对应圆和有向矩形（或者等价于球体和 OBB——这取决于观察角度）。

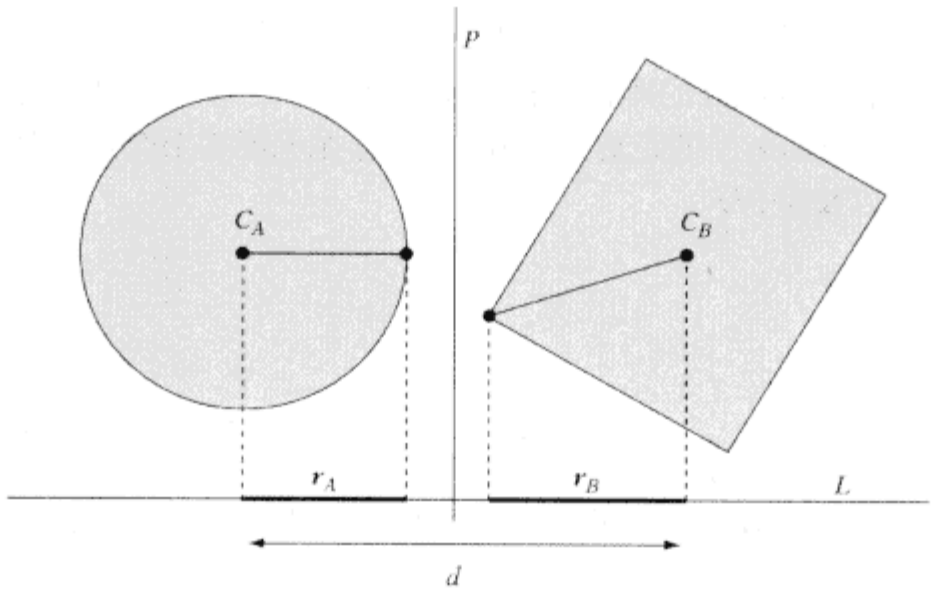


图 5.14　若球体的投影半径（或宽度值的 1/2）和小于其中心投影距离，则两球体间处于分离状态

首先，对于每一个物体对象获取沿 L 的一个支撑点，即 L 上任一方向、距离物体中心点最远的一点（由于投影的对称性，则存在两个方向上的等距点）。通过计算对象中心在 L 上的投影和最远点之间的距离，可以得到两个物体对象的半径 r_A 和 r_B，同时也可得到两物体中心间的投影距离 d。给定上述计算数据，若 $r_A + r_B < d$，则两物体间处于分离状态。

针对非对称物体对象——如任意形状的凸体——也可以应用分离轴测试方案，但测试方法需要进行相应的修改，即需要将凸体顶点投影至分离轴上以计算相应的投影区间。

通过相应的检测手段，分离轴易于计算。然而在具体实现过程中，需要将无穷多个潜在分离轴限制在少数几个可测试轴上。对于凸多面体，可以显著地降低测试轴的数量。不考虑碰撞顺序，两个多面体对象依据其特征可以产生 6 种不同的碰撞方式：面-面、面-边、面-顶点、边-边、边-顶点以及顶点-顶点。由于顶点可以视为边的一部分，则上述顶点测试组合可归类于边碰撞情形。因而，碰撞分类减少至 3 类组合：面-面、面-边以及边-边。

对于面-面测试，可以将物体的面法线作为潜在的测试分离轴；而对于边-边测试，潜在的测试分离轴对应于两条边的叉积。下面将考查物体间发生边-边碰撞时生成的数据信息，并分析叉积向量用于分离轴测试的原因。两条最近碰撞边上的点将作为边垂直向量的基点。该垂直向量为此二边的叉积向量，作为分离轴，它将是一个测试时的有效备选方案。总体上讲，多面体对象间的分离测试需要考查下列轴：

- ☑ 平行于物体 A 的面法线的轴。
- ☑ 平行于物体 B 的面法线的轴。
- ☑ 平行于物体 A、B 各边生成的叉积向量的轴。

一旦获取分离轴，测试可立即退出并返回“分离状态”这一信息。若全部轴测试完毕且不存在分

离轴，则对象间必定相交。

对于包含相同数量面（F）和边（E）的两个常规多面体，潜在分离轴的数量为 $2F+E^2$。由于分离轴的数量是边的二次函数，对于具有较高复杂度的物体对象而言，分离轴测试则在实际应用中难以实现。然而，可以通过缓存最近一次有效分离轴且在后续查询中首先对其执行测试这一方式加速分离轴的测试过程。当然，理想情况下最好提前退出测试，并在两次查询之间保持对象空间、时间上的一致性。

当两个多面体产生碰撞时，分离轴测试也可用于辅助计算碰撞信息。当测试未曾出现“提前退出”且在某一轴上检测到碰撞发生时，需要在全部轴上执行相交测试。待全部轴检测完毕后，选择具有最小重叠量的轴向作为碰撞法线，且上述重叠量还可以用于估算穿越深度。另外，辅以某些额外的计算且通过分离轴的支持，可以最终得到相应的碰撞点。[Larcombe95]中给出了更多有关包围盒分离轴测试的理论，读者可以深入了解相关内容。

5.2.1.1　分离轴测试的健壮性

如前所述，物体间相应边的叉积向量可以构造分离轴，但这也导致了与分离轴测试相关的、潜在的健壮性问题。当两条边平行时，其叉积结果为 0 向量。因而，该轴上的投影以及投影之和也将为 0。如果测试过程不谨慎地处理这类问题，0 向量将被错误地用于分离轴。由于计算过程采用了浮点运算，对于接近平行的两条边所生成的准 0 向量，也有可能产生上述问题。实际上，有关分离轴的健壮性问题在 4.4.2 节中计算 OBB 间的相交测试时就已出现。当时，也给出了特定环境下该问题的处理方法。

在条件允许的情况下，健壮性问题应遵循具体情况具体分析这一原则。一类普遍采用的方案是：检测叉积向量是否为（准）0 向量。若是，则重新生成与两个边向量垂直的新轴，或者仅忽略该轴。

下列代码片段针对边 AB 和 CD，给出了健壮的分离轴测试的实现方式：

```
// Compute a tentative separating axis for ab and cd
Vector m = Cross(ab, cd);
if (!IsZeroVector(m)) {
    // Edges ab and cd not parallel, continue with m as a potential separating axis
    ...
} else {
    // Edges ab and cd must be (near) parallel, and therefore lie in some plane P.
    // Thus, as a separating axis try an axis perpendicular to ab and lying in P
    Vector n = Cross(ab, c - a);
    m = Cross(ab, n);
    if (!IsZeroVector(m)) {
        // Continue with m as a potential separating axis
        ...
    }
    // ab and ac are parallel too, so edges must be on a line. Ignore testing
    // the axis for this combination of edges as it won't be a separating axis.
    // (Alternatively, test if edges overlap on this line, in which case the
    // objects are overlapping.)
    ...
}
```

IsZeroVector()函数测试其参数是否为准 0 向量（依据某个允许范围值，参见 11.3.1 节）。扩大这一允许范围值的区间并且将近似平行边视为平行边——这将导致相交状态被解释为准相交状态。因此，

该问题比方案的选择更加令人注目。例如，由于 0 向量分离轴上的投影，使得原本相交的两个物体被错误地报告为不相交。

另一类与健壮性相关的错误根源是叉积向量过大，从而导致计算过程中的精度损失。如果并未限定输入向量的范围，则应在叉积计算之前对向量执行范化操作以保持相应的精度。

5.2.2 球体与平面间的测试

可以采用多种方式实现球体与平面间的测试。本节将分析 3 种测试类型：测试球体是否与平面相交；测试球体是否完全位于平面的背面；测试球体是否与平面的负半空间相交。图 5.15 显示了这 3 种测试方案。

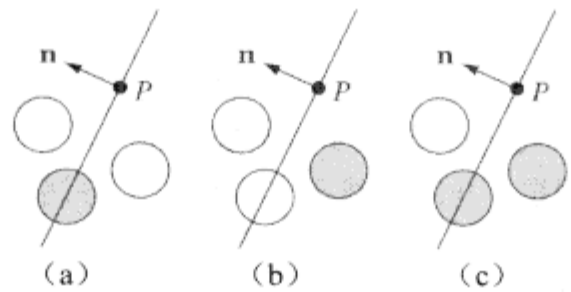


图 5.15 球体与平面之间的 3 种相交状态。（a）球体与平面相交。（b）球体完全位于平面的背面。（c）球体与平面的负半空间相交。符合条件的球体采用灰色表示

令球体 S 中心位置为 C，半径为 r，且平面 π 由方程$(\mathbf{n} \cdot X) = d$ 定义，其中 $\mathbf{n}$ 为单位向量，即$\|\mathbf{n}\| = 1$。为了计算球体是否与平面相交，可以将球心代入平面方程。由于 $\mathbf{n}$ 是单位向量，上述计算结果为球心距平面的有符号距离。若距离的绝对值小于球体半径，则平面与球体相交：

```
// Determine whether plane p intersects sphere s
int TestSpherePlane(Sphere s, Plane p)
{
    // For a normalized plane (|p.n| = 1), evaluating the plane equation
    // for a point gives the signed distance of the point to the plane
    float dist = Dot(s.c, p.n) - p.d;
    // If sphere center within +/-radius from plane, plane intersects sphere
    return Abs(dist) <= s.r;
}
```

为了确定球体是否完全位于平面 π 的背面（平面的负半空间内），测试过程可变化为：

```
// Determine whether sphere s is fully behind (inside negative halfspace of) plane p
int InsideSpherePlane(Sphere s, Plane p)
{
    float dist = Dot(s.c, p.n) - p.d;
    return dist < -s.r;
}
```

若将平面的负半空间看作是球体测试中的几何实体，则测试过程可变化为：

```
// Determine whether sphere s intersects negative halfspace of plane p
int TestSphereHalfspace(Sphere s, Plane p)
{
    float dist = Dot(s.c, p.n) - p.d;
    return dist <= s.r;
}
```

5.2.3　盒体与平面间的测试

令平面 P 由方程$(\mathbf{n} \cdot X) = d$ 定义，则盒体 B 与平面 P 的相交测试完全可采用分离轴测试方案。这里，只需测试与平面法线 $\mathbf{n}$ 平行的轴。由于平面无限扩展，因而不存在边-边组合测试。与盒体面法线对应的轴可从测试中排除，因为平面的无限扩展性表明，如果盒体的一个面不与测试平面平行，则二者总是存在相交的几率。

考查第一种情形：给定有向包围盒 B，定义其常规表达方式为：中心 C、局部坐标轴 $\mathbf{u}_0$、$\mathbf{u}_1$ 和 $\mathbf{u}_2$、3 个标量值 e_0、e_1 和 e_2（则相应面尺寸为 $2e_i$，$0 \leqslant i \leqslant 2$）。OBB 中的顶点 R 则定义为 $R = C \pm a_0\mathbf{u}_0 \pm a_1\mathbf{u}_1 \pm a_2\mathbf{u}_2$，其中$|a_i| \leqslant e_i$。类似地，OBB 中的 8 个顶点 V_i，$0 \leqslant i \leqslant 7$，定义为 $V_i = C \pm e_0\mathbf{u}_0 \pm e_1\mathbf{u}_1 \pm e_2\mathbf{u}_2$。

既然平行于 $\mathbf{n}$ 的直线都可以作为分离轴，一种较好的选择是令直线 L 穿过盒体的中心，且 L 的直线方程定义为 $L(t) = C + t\mathbf{n}$。当 $t = 0$ 时，即为盒体中心 C 在直线 L 上的投影。由于 OBB 关于其中心点对称，则其在 L 上的投影也将生成一个对称区间$[C - r\,\mathbf{n}, C + r\,\mathbf{n}]$，其中中心点为 C，半径为 r。当前，B 与 P 的相交问题转化为：计算投影区间的半径 r，并检测 B 中心点至 P 的距离是否小于 r。

由于 OBB 上距中心最大距离点一般是 OBB 的顶点，则向量 $\mathbf{n}$ 上的最大投影半径可通过这些顶点得到。因此，计算半径 r 时只需考查这一类顶点即可。相应 8 个顶点的投影半径如下所示：

$$r_i = (V_i - C) \cdot \mathbf{n} = (C \pm e_0\mathbf{u}_0 \pm e_1\mathbf{u}_1 \pm e_2\mathbf{u}_2 - C) \cdot \mathbf{n} = (\pm e_0\mathbf{u}_0 \pm e_1\mathbf{u}_1 \pm e_2\mathbf{u}_2) \cdot \mathbf{n}$$

根据点积的分配律，该表达式可记为：

$$r_i = \pm(e_0\mathbf{u}_0 \cdot \mathbf{n}) \pm (e_1\mathbf{u}_1 \cdot \mathbf{n}) \pm (e_2\mathbf{u}_2 \cdot \mathbf{n})$$

当全部计算项为正值时，可得到半径 r 的最大值，即通过计算各项绝对值并求和得到 $\mathbf{n}$ 向的正向幅度。

$$r = |e_0\mathbf{u}_0 \cdot \mathbf{n}| + |e_1\mathbf{u}_1 \cdot \mathbf{n}| + |e_2\mathbf{u}_2 \cdot \mathbf{n}|$$

由于 OBB 范围值设定为正值，r 可写为：

$$r = e_0|\mathbf{u}_0 \cdot \mathbf{n}| + e_1|\mathbf{u}_1 \cdot \mathbf{n}| + e_2|\mathbf{u}_2 \cdot \mathbf{n}|$$

若分离轴向量 $\mathbf{n}$ 不是单位向量，则 r 为：

$$r = (e_0|\mathbf{u}_0 \cdot \mathbf{n}| + e_1|\mathbf{u}_1 \cdot \mathbf{n}| + e_2|\mathbf{u}_2 \cdot \mathbf{n}|)/||\mathbf{n}||$$

将点 C 带入平面方程可得 C 距 P 的有符号距离 s，记为 $s = \mathbf{n} \cdot C - d$。$s$ 的另一种计算方法是获取 P 至 C 的距离 u，且 $s = -u$。如前所述，对平面 P 有 $\mathbf{n} \cdot X = d$，其中，对平面上的一点 Q 有 $d = Q \cdot \mathbf{n}$。由于平面 P 上的所有点在 L 上的投影结果为一点，当然也包括点 Q 在 L 上的投影。因此，距离 $u = (Q - C) \cdot \mathbf{n} = Q \cdot \mathbf{n} - C \cdot \mathbf{n} = d - C \cdot \mathbf{n}$，除去符号变化，该计算结果等同于将盒体中心点代入平面方程后的计算值。图 5.16 显示了测试过程中用到的各个量值。

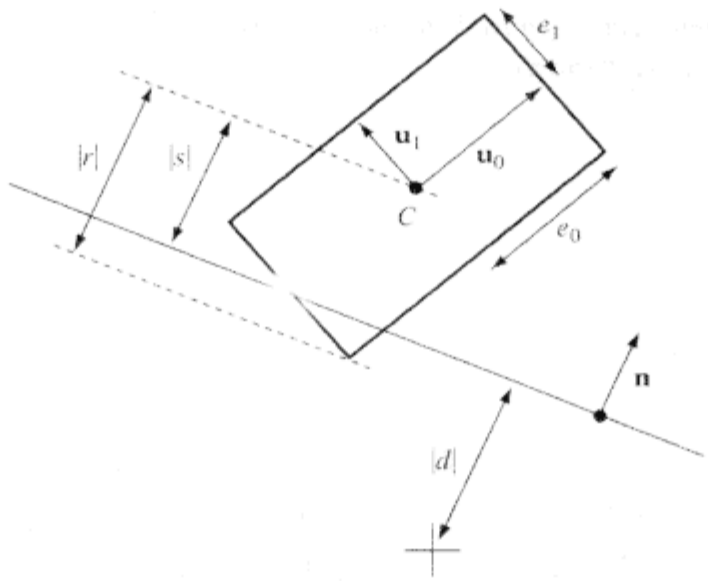


图 5.16　OBB 与平面之间的相交测试

当$-r \leqslant s \leqslant r$ 时，OBB 与平面 P 相交，即$|s| \leqslant r$。综上所述，其测试代码如下所示：

```
// Test if OBB b intersects plane p
int TestOBBPlane(OBB b, Plane p)
{
    // Compute the projection interval radius of b onto L(t) = b.c + t * p.n
    float r = b.e[0]*Abs(Dot(p.n, b.u[0])) +
              b.e[1]*Abs(Dot(p.n, b.u[1])) +
              b.e[2]*Abs(Dot(p.n, b.u[2]));
    // Compute distance of box center from plane
    float s = Dot(p.n, b.c) - p.d;
    // Intersection occurs when distance s falls within [-r,+r] interval
    return Abs(s) <= r;
}
```

测试过程中，向量 **n** 不必加以范化。若 **n** 不是单位向量，r 和 s 相当于增加了一个系数$\|\mathbf{n}\|$且对测试不会产生任何影响。

其他测试过程可以采用类似的方式加以实现。例如，若 $s \leqslant -r$，则 OBB 位于平面的负半空间；若 $r \leqslant s$，则 OBB 完全处于平面的正半空间中。给定 OBB 且定义为 $B = C + k_0\mathbf{v}_0 + k_1\mathbf{v}_1 + k_2\mathbf{v}_2$，$0 \leqslant k_0$，$k_1$，$k_2 \leqslant 1$，则半径 $r = (|\mathbf{v}_0 \cdot \mathbf{n}| + |\mathbf{v}_1 \cdot \mathbf{n}| + |\mathbf{v}_2 \cdot \mathbf{n}|)/2$；对于轴对齐包围盒 B 且已知局部坐标轴为 $\mathbf{u}_0$、$\mathbf{u}_1$、$\mathbf{u}_2$，则代码可相应地简化为：

```
// Test if AABB b intersects plane p
int TestAABBPlane(AABB b, Plane p)
{
    // These two lines not necessary with a (center, extents) AABB representation
    Point c = (b.max + b.min) * 0.5f;        // Compute AABB center
    Point e = b.max - c;                     // Compute positive extents

    // Compute the projection interval radius of b onto L(t) = b.c + t * p.n
    float r = e[0]*Abs(p.n[0]) + e[1]*Abs(p.n[1]) + e[2]*Abs(p.n[2]);
    // Compute distance of box center from plane
```

```
    float s = Dot(p.n, c) - p.d;
    // Intersection occurs when distance s falls within [-r,+r] interval
    return Abs(s) <= r;
}
```

该测试等同于沿平面法线方向计算 AABB 的最远顶点，且两对角顶点分别位于平面的两侧。

5.2.4　锥体与平面间的测试

给定平面$(\mathbf{n} \cdot X) = d$，其中对于面上一点 P，有 $d = -P \cdot \mathbf{n}$ 且 $\mathbf{n}$ 为单位向量。定义一个锥体，其中锥顶为 T，单位轴向为 d，高度为 h，底面半径为 r（如图 5.17 所示）。若锥体中存在一点且位于平面的负半空间，即对于锥体中一点 X 且有$(\mathbf{n} \cdot X) < d$，则锥体与平面的负半空间相交。针对上述条件，只需测试锥体中的两个点：

- ☑ 锥顶 T。
- ☑ 位于锥体底面上且沿$-\mathbf{n}$ 方向的最远点 Q。

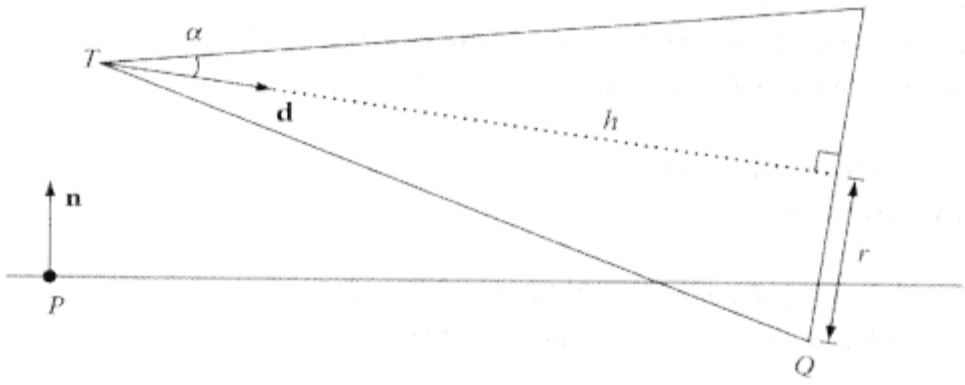


图 5.17　圆锥体与平面或半空间相交测试所涉及的相关变量

对于第二个测试，需要定位点 Q。由锥顶沿方向向量至底面且沿底面向平面方向移动，即可得到点 Q：

$$Q = T + h\,\mathbf{v} + r\,\mathbf{m}$$

其中，向量 $\mathbf{m} = (\mathbf{n} \times \mathbf{v}) \times \mathbf{v}$。

若 $\mathbf{n} \times \mathbf{v}$ 为 0 向量，则锥体轴平行于平面法线，且 $Q = T + h\,\mathbf{v}$ 为半空间测试的最终点。在这种情况下，向量 $\mathbf{m}$ 也为 0 向量且无须进行额外处理。

有时，锥体采用锥顶半角 α 表示，而非底面半径 r。此时，$\tan\alpha = r/h$，则 $r = h \tan\alpha$。但在当前测试中，定义半径 r（而非半角 α）可以使计算表达更加清晰。

如果仅对平面本身而非半空间执行相交测试，则当锥顶点位于平面背面时，Q 改为计算沿 $\mathbf{n}$ 向的最远顶点。相交测试则转换为测试点 T 和 Q 是否位于平面的两侧，即计算$(\mathbf{n} \cdot T) - d$ 和$(\mathbf{n} \cdot Q) - d$ 之间是否具有不同的符号。

5.2.5　球体与 AABB 之间的测试

测试球体与轴对齐包围盒之间是否相交，可以采取计算二者之间的中心距离（参见 5.1.3.1 节）并

与球体半径进行比较这一方法加以实现。如果该距离小于球体半径，则二者相交。为了避免昂贵的平方根运算，距离和半径皆采取平方形式且不会对测试结果产生任何影响。利用 5.1.3.1 节中的 **SqDistPointAABB()**函数，上述实现代码如下所示：

```
// Returns true if sphere s intersects AABB b, false otherwise
int TestSphereAABB(Sphere s, AABB b)
{
    // Compute squared distance between sphere center and AABB
    float sqDist = SqDistPointAABB(s.c, b);
    // Sphere and AABB intersect if the (squared) distance
    // between them is less than the (squared) sphere radius
    return sqDist <= s.r * s.r;
}
```

在碰撞处理过程中，常需要返回 AABB 上距球心最近的一点，此时，测试可利用 **ClosestPtPointAABB()**函数且代码如下所示：

```
// Returns true if sphere s intersects AABB b, false otherwise.
// The point p on the AABB closest to the sphere center is also returned
int TestSphereAABB(Sphere s, AABB b, Point &p)
{
    // Find point p on AABB closest to sphere center
    ClosestPtPointAABB(s.c, b, p);

    // Sphere and AABB intersect if the (squared) distance from sphere
    // center to point p is less than the (squared) sphere radius
    Vector v = p - s.c;
    return Dot(v, v) <= s.r * s.r;
}
```

需注意的是，计算球体是否完全位于 AABB 的面外，对于二者是否相交并不是一类充分测试。例如，球体位于 AABB 的某一边外部且不与 AABB 相交，但球体并没有完全处于该边的两个公共面之外，如图 5.18 所示。

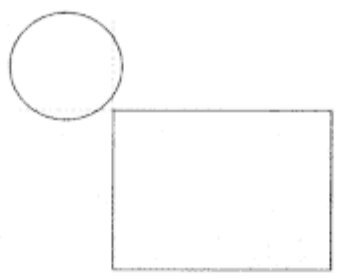


图 5.18　球体并未完全位于 AABB 数据平面的外部，却不与 AABB 相交

5.2.6　球体与 OBB 之间的测试

球体与方向包围盒之间的相交测试几乎等同于与 AABB 之间的测试，只是将 **ClosestPtPointAABB()**

函数调用替换为 **ClosestPtPointOBB()**，代码如下所示：

```
// Returns true if sphere s intersects OBB b, false otherwise.
// The point p on the OBB closest to the sphere center is also returned
int TestSphereOBB(Sphere s, OBB b, Point &p)
{
    // Find point p on OBB closest to sphere center
    ClosestPtPointOBB(s.c, b, p);

    // Sphere and OBB intersect if the (squared) distance from sphere
    // center to point p is less than the (squared) sphere radius
    Vector v = p - s.c;
    return Dot(v, v) <= s.r * s.r;
}
```

5.2.7　球体与三角形之间的测试

球体-三角形的相交测试与球体-包围盒相交测试遵循同一模式。首先计算三角形中距球心最近的一点 P，且将 P 与球心之间的距离和球体半径进行比较，进而确定其相交状态：

```
// Returns true if sphere s intersects triangle ABC, false otherwise.
// The point p on abc closest to the sphere center is also returned
int TestSphereTriangle(Sphere s, Point a, Point b, Point c, Point &p)
{
    // Find point P on triangle ABC closest to sphere center
    p = ClosestPtPointTriangle(s.c, a, b, c);
    // Sphere and triangle intersect if the (squared) distance from sphere
    // center to point p is less than the (squared) sphere radius
    Vector v = p - s.c;
    return Dot(v, v) <= s.r * s.r;
}
```

为了避免昂贵的平方根计算，代码中仍采用了平方距离。

5.2.8　球体与多边形之间的测试

可依据最近的几种测试方案对球体-多边形执行类似的测试：计算多边形上距球心的最近点，并将两点间的距离与球体半径比较。另外，给定一点 P，多边形上距 P 的最近点可按下列方式求解：对多边形执行三角形划分并计算每一个三角形距 P 的最近点，并选取最近值作为最终结果。对于大型多边形，该算法并不是一种十分高效的解决方案，对于包含 n 个顶点的多边形，需要执行 $n-2$ 次三角形测试。

另一类解决方案可以通过依次执行下列测试加以实现：

1. 测试球体是否与多边形平面相交，若不相交，则退出并返回 false。
2. 测试多边形中的每一条边并考查是否穿越球体，若穿越，则退出并返回 true。
3. 将球心投影至多边形平面上。执行点-多边形包含测试（参见 5.4.1 节）并查看该点是否位于多

边形内部。若是，则退出并返回 true，否则退出并返回 false 且球体不与多边形相交。

下列代码片段显示了上述测试的实现过程：

```
// Test whether sphere s intersects polygon p
int TestSpherePolygon(Sphere s, Polygon p)
{
    // Compute normal for the plane of the polygon
    Vector n = Normalize(Cross(p.v[1] - p.v[0], p.v[2] - p.v[0]));
    // Compute the plane equation for p
    Plane m; m.n = n; m.d = -Dot(n, p.v[0]);
    // No intersection if sphere not intersecting plane of polygon
    if (!TestSpherePlane(s, m)) return 0;
    // Test to see if any one of the polygon edges pierces the sphere
    for (int k = p.numVerts, i = 0, j = k - 1; i < k; j = i, i++) {
        float t;
        Point q;
        // Test if edge (p.v[j], p.v[i]) intersects s
        if (IntersectRaySphere(p.v[j], p.v[i] - p.v[j], s, t, q) && t <= 1.0f)
        return 1;
    }
    // Test if the orthogonal projection q of the sphere center onto m is inside p
    Point q = ClosestPtPointPlane(s.c, m);
    return PointInPolygon(q, p);
}
```

对于上述方案，作为一种优化手段可以将前两步改为：将球体和多边形投影至包含多边形最大面积的主平面上，并使问题转化为圆与多边形之间的 2D 测试，这将减少测试过程中相应的计算量。

5.2.9 AABB 与三角形之间的测试

三角形 T 与包围盒 B 之间的相交测试可通过分离轴测试这一方案得以高效实现（[Eberly01]、[Akenine-Möller01]）。其中，需要考查 13 条轴上的投影情况：

1. AABB 上的 3 条面法线。
2. 三角形上的一条面法线。
3. 二者之间边-边组合形成的 9 条叉积轴。

如前所述，一旦计算得到相应的分离轴，测试退出并返回“不相交”这一信息。当全部轴测试完毕且未发现分离轴，则包围盒与三角形之间必定相交。另外，[Akenine-Möller01]中建议，高效的测试顺序为 3—1—2。

上述 13 轴测试方案也可应用于 OBB 与 AABB 之间的相交测试。但针对 AABB（由于其局部坐标轴已知），可采用某些优化手段加速测试运行期间的计算。这里，虽然只是针对 AABB 执行测试，但为了显示与 OBB 之间的类似性并简化 AABB 的特定优化方案，AABB 采用了 OBB 中的常规构造方法，即中心 C；局部坐标轴 $\mathbf{u}_0=(1, 0, 0)$，$\mathbf{u}_1=(0, 1, 0)$，$\mathbf{u}_2=(0, 0, 1)$；范围值 e_0，e_1，e_2。测试三角形定义为 $V_0=(v_{0x},v_{0y},v_{0z})$，$V_1=(v_{1x},v_{1y},v_{1z})$，$V_2=(v_{2x},v_{2y},v_{2z})$。

首先，针对 AABB 的 3 条面法线（$\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2$）进行测试：计算三角形的 AABB 并测试与 B 的相交性。若两个 AABB 之间不相交，则 B 与 T 之间也不相交。若相应轴与三角形面法线平行，则测试转化为 AABB 是否与三角形平面相交——该测试算法可参见 5.2.3 节，在此不再赘述。最后，将对 B 与 T 中 9 条边组合形成的叉积轴进行测试。

如多数分离轴测试所示，可以将某一物体对象移至原点以简化计算过程。这里，盒体中心点被移至原点，相关变量按下列方式得以更新：$V_0 \leftarrow V_0 - C$，$V_1 \leftarrow V_1 - C$，$V_2 \leftarrow V_2 - C$，$C \leftarrow (0, 0, 0)$。三角形 3 条边定义为 $\mathbf{f}_0 = V_1 - V_0 = (f_{0x}, f_{0y}, f_{0z})$，$\mathbf{f}_1 = V_2 - V_1 = (f_{1x}, f_{1y}, f_{1z})$，$\mathbf{f}_2 = V_0 - V_2 = (f_{2x}, f_{2y}, f_{2z})$。作为分离轴加以考查的 9 个轴定义为 $\mathbf{a}_{ij} = \mathbf{u}_i \times \mathbf{f}_j$。如下所示，由于 $\mathbf{u}_0$，$\mathbf{u}_1$，$\mathbf{u}_2$ 皆为基础向量，9 条轴可简化为：

$$\mathbf{a}_{00} = \mathbf{u}_0 \times \mathbf{f}_0 = (1, 0, 0) \times \mathbf{f}_0 = (0, -f_{0z}, f_{0y})$$
$$\mathbf{a}_{01} = \mathbf{u}_0 \times \mathbf{f}_1 = (1, 0, 0) \times \mathbf{f}_1 = (0, -f_{1z}, f_{1y})$$
$$\mathbf{a}_{02} = \mathbf{u}_0 \times \mathbf{f}_2 = (1, 0, 0) \times \mathbf{f}_2 = (0, -f_{2z}, f_{2y})$$
$$\mathbf{a}_{10} = \mathbf{u}_1 \times \mathbf{f}_0 = (0, 1, 0) \times \mathbf{f}_0 = (f_{0z}, 0, -f_{0x})$$
$$\mathbf{a}_{11} = \mathbf{u}_1 \times \mathbf{f}_1 = (0, 1, 0) \times \mathbf{f}_1 = (f_{1z}, 0, -f_{1x})$$
$$\mathbf{a}_{12} = \mathbf{u}_1 \times \mathbf{f}_2 = (0, 1, 0) \times \mathbf{f}_2 = (f_{2z}, 0, -f_{2x})$$
$$\mathbf{a}_{20} = \mathbf{u}_2 \times \mathbf{f}_0 = (0, 0, 1) \times \mathbf{f}_0 = (-f_{0y}, f_{0x}, 0)$$
$$\mathbf{a}_{21} = \mathbf{u}_2 \times \mathbf{f}_1 = (0, 0, 1) \times \mathbf{f}_1 = (-f_{1y}, f_{1x}, 0)$$
$$\mathbf{a}_{22} = \mathbf{u}_2 \times \mathbf{f}_2 = (0, 0, 1) \times \mathbf{f}_2 = (-f_{2y}, f_{2x}, 0)$$

轴 $\mathbf{n}$ 上的盒体投影半径为：

$$r = e_0|\mathbf{u}_0 \cdot \mathbf{n}| + e_1|\mathbf{u}_1 \cdot \mathbf{n}| + e_2|\mathbf{u}_2 \cdot \mathbf{n}|$$

其中，$\mathbf{n} = \mathbf{a}_{00}$，则上式简化为：

$$r = e_0|\mathbf{u}_0 \cdot \mathbf{a}_{00}| + e_1|\mathbf{u}_1 \cdot \mathbf{a}_{00}| + e_2|\mathbf{u}_2 \cdot \mathbf{a}_{00}| \Leftrightarrow$$
$$r = e_0|0| + e_1|-\mathbf{a}_{00z}| + e_2|\mathbf{a}_{00y}| \Leftrightarrow$$
$$r = e_1|f_{0z}| + e_2|f_{0y}|$$

对于其他 $\mathbf{a}_{ij}$ 轴上的 AABB 的投影半径，其简化表达式也可以采取类似的方式获得。另外，盒体的投影区间皆位于$[-r, r]$。

将 T 投影至轴 $\mathbf{n}$ 上时，其投影区间范围为$[\min(p_0, p_1, p_2), \max(p_0, p_1, p_2)]$，其中 p_0，p_1，p_2 为原点至三角形顶点在 $\mathbf{n}$ 上的投影点之间的距离。由于 $\mathbf{n} = \mathbf{a}_{00}$，则有：

$$p_0 = V_0 \cdot \mathbf{a}_{00} = V_0 \cdot (0, -f_{0z}, f_{0y}) = -v_{0y}f_{0z} + v_{0z}f_{0y} = -v_{0y}(v_{1z} - v_{0z}) + v_{0z}(v_{1y} - v_{0y})$$
$$= -v_{0y}v_{1z} + v_{0z}v_{1y}$$
$$p_1 = V_1 \cdot \mathbf{a}_{00} = V_1 \cdot (0, -f_{0z}, f_{0y}) = -v_{1y}f_{0z} + v_{1z}f_{0y} = -v_{1y}(v_{1z} - v_{0z}) + v_{1z}(v_{1y} - v_{0y})$$
$$= v_{1y}v_{0z} - v_{1z}v_{0y} = p_0$$
$$p_2 = V_2 \cdot \mathbf{a}_{00} = V_2 \cdot (0, -f_{0z}, f_{0y}) = -v_{2y}f_{0z} + v_{2z}f_{0y} = -v_{2y}(v_{1z} - v_{0z}) + v_{2z}(v_{1y} - v_{0y})$$

在给定分离轴上，如果投影区间$[-r, r]$与区间$[\min(p_0, p_1, p_2), \max(p_0, p_1, p_2)]$无交集，则三角形与 AABB 不相交。

对于轴 $\mathbf{n} = \mathbf{a}_{00}$，有 $p_0 = p_1$ 且三角形的投影区间简化为$[\min(p_0, p_2), \max(p_0, p_2)]$。由于 9 条轴向量都包含 0 分量，因此在这 9 条轴上的三角形投影区间可以采用同一简化方式。这里，若 $\max(p_0, p_2) < -r$ 或 $\min(p_0, p_2) > r$，则 AABB 和三角形投影区间之间无交集。若目标系统中包含 min()和 max()浮点指令，可采用 $\max(-\max(p_0, p_2), \min(p_0, p_2)) > r$ 以消除前述（可能代价较高的）比较操作——在 SIMD 实现中，

该比较方式十分有效（关于更多的 SIMD 实现方式可参见第 13 章）。

下列代码片段给出了上述测试的实现方式：

```
int TestTriangleAABB(Point v0, Point v1, Point v2, AABB b)
{
    float p0, p1, p2, r;

    // Compute box center and extents (if not already given in that format)
    Vector c = (b.min + b.max) * 0.5f;
    float e0 = (b.max.x - b.min.x) * 0.5f;
    float e1 = (b.max.y - b.min.y) * 0.5f;
    float e2 = (b.max.z - b.min.z) * 0.5f;

    // Translate triangle as conceptually moving AABB to origin
    v0 = v0 - c;
    v1 = v1 - c;
    v2 = v2 - c;

    // Compute edge vectors for triangle
    Vector f0 = v1 - v0,    f1 = v2 - v1, f2 = v0 - v2;

    // Test axes a00..a22 (category 3)
    // Test axis a00
    p0 = v0.z*v1.y - v0.y*v1.z;
    p2 = v2.z*(v1.y - v0.y) - v2.z*(v1.z - v0.z);
    r = e1 * Abs(f0.z) + e2 * Abs(f0.y);
    if (Max(-Max(p0, p2), Min(p0, p2)) > r) return 0;        // Axis is a separating axis

    // Repeat similar tests for remaining axes a01..a22
    ...

    // Test the three axes corresponding to the face normals of AABB b (category 1).
    // Exit if…
    // … [-e0, e0] and [min(v0.x,v1.x,v2.x), max(v0.x,v1.x,v2.x)] do not overlap
    if (Max(v0.x, v1.x, v2.x) < -e0 || Min(v0.x, v1.x, v2.x) > e0) return 0;
    // … [-e1, e1] and [min(v0.y,v1.y,v2.y), max(v0.y,v1.y,v2.y)] do not overlap
    if (Max(v0.y, v1.y, v2.y) < -e1 || Min(v0.y, v1.y, v2.y) > e1) return 0;
    // … [-e2, e2] and [min(v0.z,v1.z,v2.z), max(v0.z,v1.z,v2.z)] do not overlap
    if (Max(v0.z, v1.z, v2.z) < -e2 || Min(v0.z, v1.z, v2.z) > e2) return 0;

    // Test separating axis corresponding to triangle face normal (category 2)
    Plane p;
    p.n = Cross(f0, f1);
    p.d = Dot(p.n, v0);
    return TestAABBPlane(b, p);
}
```

需注意的是，在前述投影分类中，第 2 类（计算退化三角形或超大尺寸三角形的面法线）和第 3 类（两条平行边的叉积结果将产生 0 向量）情形都涉及相应的健壮性问题。关于健壮性问题，较全面

的解决方案可参见 5.2.1.1 节。

[Voorhies92]中也给出了三角形-AABB 相交测试的相关探讨。关于任意多边形与 AABB 之间的相交测试，可参见[Green95]。

5.2.10　三角形之间的测试

存在多种算法计算三角形 *ABC* 和 *DEF* 之间的相交性问题。一类较为直观的测试方法是：某一个三角形的两条边穿越另一个三角形内部区域；或三角形之间出现单边彼此内部穿越（如图 5.19 所示）。

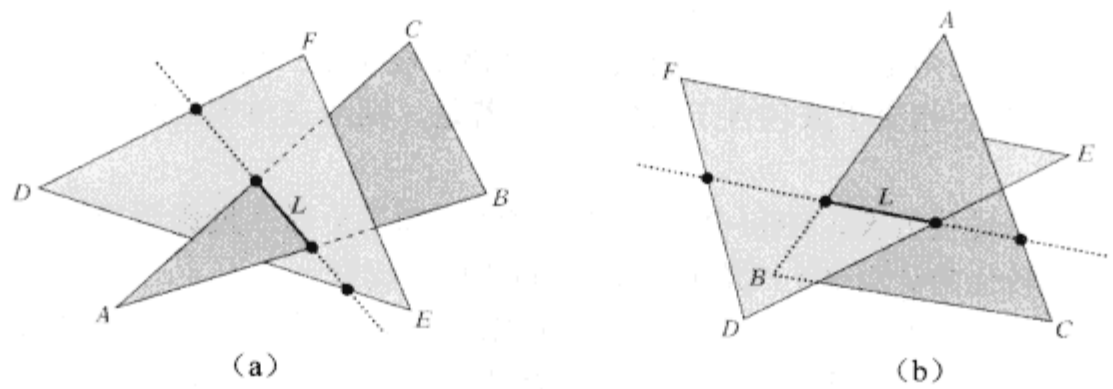


图 5.19　两个三角形之间的相交状态。（a）三角形的两条边穿越另一个三角形的内部区域。（b）单边内部穿越

上述测试均无法正确处理共面三角形以及退化三角形（退化为直线或点）这一类问题。当然，一般情况下不存在相关的三角形相交测试算法能够默认地处理这一类问题：相关问题均视为特例加以处理。

第 2 种解决方案是采用分离轴测试。在三角形之间的测试过程中，需要考查 11 个分离轴：平行于两个三角形面法线的两个轴；两个三角形之间的 9 种边-边组合。针对每一个轴，三角形将投影至该轴上并计算用于相交测试的投影区间。若任一轴上不存在投影区间的交集，则三角形之间不相交且退出测试；若投影区间在全部 11 个轴上均相交，则两个三角形之间必定相交。

[Möller97b]中给出了一种类似于分离轴的测试方案——区间相交算法。与前两种方法相比，该算法成本低廉。算法首先查看两条面法线是否为分离轴。这可以在两个三角形之间查看三角形顶点是否位于另一个三角形的一侧加以实现：若是，则三角形之间不相交，否则三角形平面必相交于一直线 L 且 $L(t) = P + t\,\mathbf{d}$，其中 $\mathbf{d} = \mathbf{n}_1 \times \mathbf{n}_2$ 为两三角形面法线 $\mathbf{n}_1$、$\mathbf{n}_2$ 的叉积向量。另外，该直线也交于这两个三角形。现在则可以计算三角形与 L 之间的相交区间（图 5.19 所示标记为黑点的交点）。仅当上述区间存在交集，则三角形相交。一种优化方案是：不采用 L 直接计算三角形的相交区间，而是在最接近于平行 L 的主坐标轴上计算区间并判断区间的交集。综上所述，区间相交算法需要执行下列步骤：

1．计算三角形 1 的平面方程。若三角形 2 的顶点位于该平面一侧，测试退出并返回“不相交”这一信息。

2．计算三角形 2 的平面方程。若三角形 1 的顶点位于该平面一侧，测试退出并返回“不相交”这一信息。

3．计算两平面的交线 L。

4．确定最接近平行于 L 的主坐标轴。

5．计算已投影至相应主坐标轴上的各三角形中交线 L 的相交区间。

6．若相交区间存在交集，则三角形相交，否则三角形之间不相交。

该测试的 C 实现代码可以通过网络下载。

另一个与区间相交测试具有类似复杂度但其概念更易理解的算法是穿越（penetration）算法，该算法出现于 ERIT 相交测试包中（[Held97]）。该算法将测试三角形平面之间是否出现彼此穿越这一现象，若不出现，测试退出并返回“不相交”这一信息，否则将计算相交线段，并测试该线段与第一个三角形的相交状态以及包含状态，从而确定两三角形之间的最终相交结果。综上所述，穿越算法步骤如下：

1．计算三角形 1 的平面方程。若三角形 2 的全部顶点位于该平面的一侧，则测试退出并返回“不相交”这一信息。

2．计算三角形 2 与三角形 1 平面的相交线段，且该线段位于三角形 1 的平面中。

3．测试相交线段是否相交于或包含于三角形 2。若是，则两三角形之间相交，否则两三角形之间不相交。

作为步骤 3 的一种优化方案，Held 建议：可以将线段和三角形 2 投影至最接近于平行三角形的主平面之上，且将问题转化为计算 2D 三角形-线段之间的相交测试。

最后一种测试算法也类似于区间相交算法，并在[Devillers02]中给出。在该算法中，前两个步骤仍然相同。Devillers 算法将计算两个三角形的“规范格式”，从而简化两个三角形平面交线 L 上的区间相交测试。该“规范格式”将依次交换三角形的顶点排列，即限定位于另一个三角形平面一侧的顶点为当前三角形的首顶点（即顶点 A 和顶点 D），交换其余顶点并确保顶点 A 和顶点 D 位于另一相应三角形平面之上（按逆时针方向观察另一相应三角形）。

待获取了三角形的“规范格式”后，顶点 A 和顶点 B 的关联边（即边 AB、AC 和边 DE、DF）将与两平面交线 L 相交。此时，L 上的区间相交计算可以采用相应顶点上的两个标量三重积测试加以确定，而无须显式计算该区间。

总体而言，与这里所讨论的其他算法相比，Devillers 测试算法逻辑较为复杂。但 Devillers 提出，若算法加以正确实现，其结果代码的执行效果将呈现一定的快速性，且能够满足相应的健壮性要求。

5.3 直线、光线和有向线段的相交测试

直线、光线和线段的相交计算常用于各类测试中，如模拟飞行的子弹或瞄准线测试。直线相交测试常用来替代复杂的查询计算。例如，玩家的手、脚与周围环境的接触状态、车轮与地面的接触状态都可以采用简单的直线测试实现高效的建模方案。下面将针对常用的几类碰撞图元探讨高效的直线、光线和线段的相交测试方案。

5.3.1 线段与平面的相交测试

给定平面 P 且有直线方程$(\mathbf{n} \cdot X) = d$，线段的参数化方程为 $S(t) = A + t(B - A)$，其中 $0 \leqslant t \leqslant 1$（如图 5.20 所示）。线段与平面相交时的 t 值可以通过替换平面参数方程中的 X 并求解 t 而得到：

$$(\mathbf{n} \cdot (A + t(B - A))) = d \text{（利用 } S(t) = A + t(B - A) \text{替换} (\mathbf{n} \cdot X) = d \text{ 中的 } X\text{）}$$

$$\mathbf{n} \cdot A + t\,\mathbf{n} \cdot (B - A) = d \text{（点积扩展）}$$

$$t\,\mathbf{n} \cdot (B - A) = d - \mathbf{n} \cdot A \text{（将标量移至 } RHS\text{）}$$

$$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A)) \quad \text{（两边同除以 } \mathbf{n} \cdot (B - A) \text{并求解 } t\text{）}$$

最后，将 t 值代入至线段的参数化方程，并计算得到交点 Q：

$$Q = A + [(d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A))](B - A)$$

其实现过程如下所示：

```
int IntersectSegmentPlane(Point a, Point b, Plane p, float &t, Point &q)
{
    // Compute the t value for the directed line ab intersecting the plane
    Vector ab = b - a;
    t = (p.d - Dot(p.n, a)) / Dot(p.n, ab);

    // If t in [0..1] compute and return intersection point
    if (t >= 0.0f && t <= 1.0f) {
        q = a + t * ab;
        return 1;
    }
    // Else no intersection
    return 0;
}
```

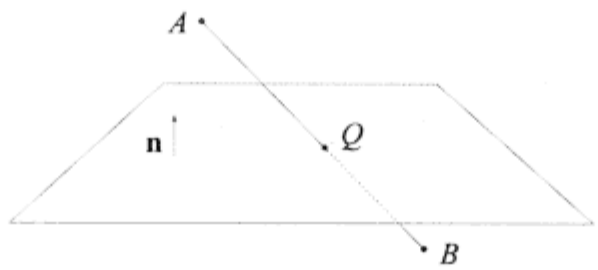


图 5.20　线段 AB 与平面的相交状态

需要注意的是，代码中并没有显示处理“除 0”这一情况。这里，假定遵循 IEEE-754 二进制浮点数算术标准，该标准给出了在除 0 时的正确的处理结果。可以参考 11.2.2 节中关于结果无穷大时的相关内容，以了解如何隐式、正确地处理除 0 错误。

如果平面未显式地加以定义，且只给出了面上的 3 个（非共线）顶点，其测试过程如下所示：

```
// Intersect segment ab against plane of triangle def. If intersecting,
// return t value and position q of intersection
int IntersectSegmentPlane(Point a, Point b, Point d, Point e, Point f,
                          float &t, Point &q)
{
    Plane p;
    p.n = Cross(e - d, f - d);
    p.d = Dot(p.n, d);
    return IntersectSegmentPlane(a, b, p, t, q);
}
```

5.3.2 光线或线段与球体的相交测试

令光线定义为 $R(t) = P + t\mathbf{d}$ 且 $t \geqslant 0$，其中 P 为光源，$\mathbf{d}$ 为标准化的方向向量且$\|\mathbf{d}\| = 1$。另外，若 $R(t)$表示为一条线段而非一束光线，则限定 $0 \leqslant t \leqslant t_{max}$。下面令球体边界定义为$(X - C) \cdot (X - C) = r^2$，其中 C 为球心，r 为球体半径。当计算光线与球体表面相交时的 t 值时，可用 $R(t)$替换 X：

$$(P + t\mathbf{d} - C) \cdot (P + t\mathbf{d} - C) = r^2$$

令 $\mathbf{m} = P - C$，则有：

$$(\mathbf{m} + t\,\mathbf{d}) \cdot (\mathbf{m} + t\,\mathbf{d}) = r^2 \Leftrightarrow \text{（替换 } \mathbf{m} = P - C\text{）}$$

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2(\mathbf{m} \cdot \mathbf{d})t + (\mathbf{m} \cdot \mathbf{m}) = r^2 \Leftrightarrow \text{（点积扩展）}$$

$$t^2 + 2(\mathbf{m} \cdot \mathbf{d})t + (\mathbf{m} \cdot \mathbf{m}) - r^2 = 0 \text{（利用 } \mathbf{d} \cdot \mathbf{d}=1 \text{ 进行简化操作；获取二次等式的标准格式）}$$

上述结果为 t 的二次方程。对于二次方程 $t^2 + 2bt + c = 0$，解为 $t = -b \pm \sqrt{b^2 - c}$。这里，$b = \mathbf{m} \cdot \mathbf{d}$，$c = (\mathbf{m} \cdot \mathbf{m}) - r^2$。

二次方程的解包含 3 个结果，这取决于判别式 $d = b^2 - c$。若 $d < 0$，则方程无实根且光线不与球体相交；若 $d = 0$，则方程存在一个实根（double 类型），且光线与球体相切于一点；若 $d > 0$，方程存在两个实根，相应光线与球体相交两次：一次进入球体边界，一次离开球体边界。其中，较小的相交 t 值为$-b - \sqrt{b^2 - c}$。然而，值得注意的是一类无效相交状态，即光线始于球体外部且背向球体，则 $t < 0$。相应示例如图 5.21 所示，其中给出了光线-球体相交测试的各种状态。

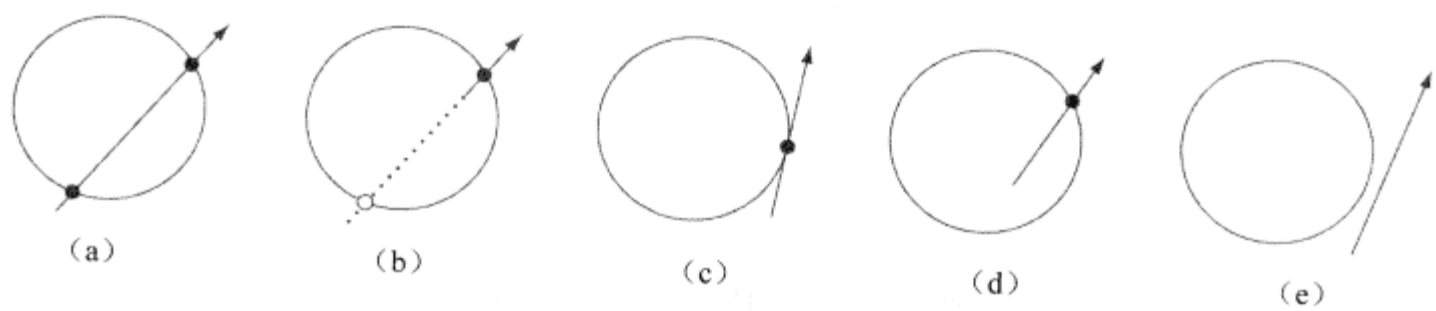


图 5.21 光线与球体间的相交状态。（a）$t > 0$，光线与球体相交（两次）。（b）$t < 0$，无效相交状态。（c）光线与球体相切。（d）光线始于球体内部。（e）光线与球体不相交

下列代码实现了相应的光线-球体相交测试。

```
// Intersects ray r = p + td, |d| = 1, with sphere s and, if intersecting,
// returns t value of intersection and intersection point q
int IntersectRaySphere(Point p, Vector d, Sphere s, float &t, Point &q)
{
    Vector m = p - s.c;
    float b = Dot(m, d);
    float c = Dot(m, m) - s.r * s.r;
    // Exit if r's origin outside s (c > 0) and r pointing away from s (b > 0)
    if (c > 0.0f && b > 0.0f) return 0;
    float discr = b*b - c;
    // A negative discriminant corresponds to ray missing sphere
    if (discr < 0.0f) return 0;
```

```
    // Ray now found to intersect sphere, compute smallest t value of intersection
    t = -b - Sqrt(discr);
    // If t is negative, ray started inside sphere so clamp t to zero
    if (t < 0.0f) t = 0.0f;
    q = p + t * d;
    return 1;
}
```

对于有向线段 AB 与球体间的测试，可复用同一代码且令 $P = A$，$\mathbf{d} = (B - A)/\|B - A\|$。一旦出现相交，则需要确认是否 $t \leqslant \|B - A\|$，从而保证交点位于线段之内。

如果只是测试光线-球体之间是否相交（不涉及碰撞时刻或碰撞位置），则代码可进行优化从而避免昂贵的平方根计算。同时，尽可能地使测试提前退出，代码如下所示：

```
// Test if ray r = p + td intersects sphere s
int TestRaySphere(Point p, Vector d, Sphere s)
{
    Vector m = p - s.c;
    float c = Dot(m, m) - s.r * s.r;
    // If there is definitely at least one real root, there must be an intersection
    if (c <= 0.0f) return 1;
    float b = Dot(m, d);
    // Early exit if ray origin outside sphere and ray pointing away from sphere
    if (b > 0.0f) return 0;
    float disc = b*b - c;
    // A negative discriminant corresponds to ray missing sphere
    if (disc < 0.0f) return 0;
    // Now ray must hit sphere
    return 1;
}
```

若向量 $\mathbf{d}$ 未曾实现范化，二次方程的求解过程则变为$(\mathbf{d} \cdot \mathbf{d})t^2+2(\mathbf{m} \cdot \mathbf{d})t+(\mathbf{m} \cdot \mathbf{m}) - r^2=0$。对于这一类形如 $at^2+ 2bt + c = 0$ 的二次方程，解为：

$$t = \frac{-b + \sqrt{b^2 - ac}}{a}$$

其中，相应的判别式为 $d = b^2 - ac$。

5.3.3　光线或线段与盒体的相交测试

回忆一下，两个平行平面构成了一个平行平面空间（slab），矩形盒体可以看作是 3 个彼此垂直的平行平面空间的交集。当且仅当一点同时位于 3 个平面空间内，该点位于盒体内；当且仅当线段与平面空间的相交结果之间也存在交集的情况下，线段与盒体相交。若线段与平面空间的相交结果之间无交集，则线段必不同时位于 3 个平面空间中，因而也不相交于 3 个平面空间交集构成的盒体。同一原理也可应用于光线或直线与盒体的相交测试中。针对由 x-slab 和 y-slab 构成的 2D 盒体，光线与其相交状态如图 5.22 所示。

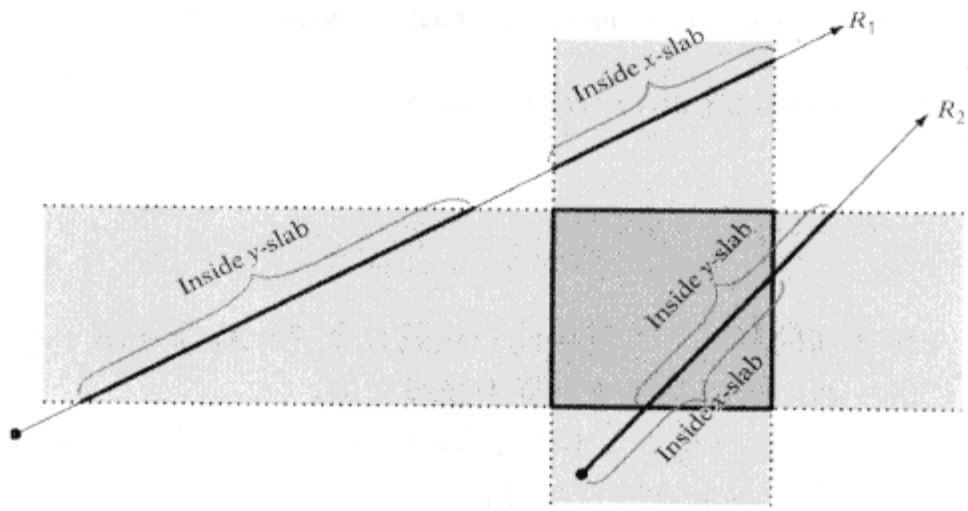


图 5.22　由于光线 R_1 与 x-slab、y-slab 之间的相交结果不存在交集，则该光线不与盒体相交；另外，由于光线 R_2 的 slab 相交结果间存在交集，因而该光线与盒体相交

因此，光线与盒体的相交测试只需计算光线与 slab 平面的相交区间，并进行某些简单的比较操作，从而呈现光线方向上的全部相交区间的正确的交集。这里，需要进行的工作是记录光线进入 slab 时各入口的最大值，以及离开 slab 时各出口的最小值。如果最大入口值大于最小出口值，则光线不与 slab 集合体相交且测试退出（返回“不相交”这一信息）。

光线与 slab 的相交区间可以通过将光线的参数化方程 $R(t) = P + t\,\mathbf{d}$ 带入到 slab 平面方程 $X \cdot \mathbf{n}_i = d_i$ 中求解，最终得到 $t = (d - P \cdot \mathbf{n}_i)/(\mathbf{d} \cdot \mathbf{n}_i)$。对于 AABB，各法线向量中均有两个分量为 0，因此，给定 $P = (p_x, p_y, p_z)$，$\mathbf{d} = (d_x, d_y, d_z)$，则针对垂直于 x 轴的平面，上述表达式可简化为 $t = (d - p_x)/d_x$，其中 d 为 x 轴上平面的相应位置。另外，为了避免光线平行于 slab 平面所产生的除 0 错误，需要对光源点单独进行 slab 包含测试。下列代码实现了光线与 AABB 之间的相交测试：

```
// Intersect ray R(t) = p + t*d against AABB a. When intersecting,
// return intersection distance tmin and point q of intersection
int IntersectRayAABB(Point p, Vector d, AABB a, float &tmin, Point &q)
{
    tmin = 0.0f;                // set to -FLT_MAX to get first hit on    line
    float tmax = FLT_MAX;       // set to max distance ray can travel (for segment)
    // For all three slabs
    for (int i = 0; i < 3; i++) {
        if (Abs(d[i]) < EPSILON) {
            // Ray is parallel to slab. No hit if origin not within slab
            if (p[i] < a.min[i] || p[i] > a.max[i]) return 0;
    } else {
        // Compute intersection t value of ray with near and far plane of slab
        float ood = 1.0f / d[i];
        float t1 = (a.min[i] - p[i]) * ood;
        float t2 = (a.max[i] - p[i]) * ood;
        // Make t1 be intersection with near plane, t2 with far plane
        if (t1 > t2) Swap(t1, t2);
        // Compute the intersection of slab intersection intervals
        if (t1 > tmin) tmin = t1;
```

```
            if (t2 > tmax) tmax = t2;
            // Exit with no collision as soon as slab intersection becomes empty
            if (tmin > tmax) return 0;
        }
    }
    // Ray intersects all 3 slabs. Return point (q) and intersection t value (tmin)
    q = p + d * tmin;
    return 1;
}
```

如果该光线与多个盒体进行相交测试，则可以一次性地计算出光线与每 3 个 slab 的划分状态，然后利用该结果分别加以判断。这里的计算过程可以采取进一步优化：通过判断光线方向向量中各分量的相应符号值，对语句 **if (t1 > t2) Swap(t1, t2)**实现预计算。因此，可提前确定光线方向向量 8 种（对于 2D 测试则为 4 种）组合中的一种，从而替换上述 if 语句，即 if 语句的判断结果视具体情况而定。

需要注意的是，上述光线-盒体相交测试过程只是 Kay-Kajiya slab 体相交测试的一个特例，参见 4.6.1 节。其中还讲到，增加 slab 的数量可以实现对象凸体的良好匹配。另外，Kay-Kajiya 测试又是 Cyrus-Beck 剪裁算法的一个特例（[Cyrus78]）。在 5.3.8 节中计算光线或线段与凸多面体相交时，便使用了一种类似于 Cyrus-Beck 剪裁算法的解决方案。

如果问题只归结为计算线段与盒体之间的相交测试且不返回交点，则该计算过程可采用分离轴算法。不失一般性，坐标系统可以选择为：盒体中心定位在其原点，方向定义为 3 个坐标轴。对于 AABB，可将线段连同 AABB 一起移至原点以完成中心定位的计算；对于 OBB，可将线段的两个端点从世界坐标系统转换至 OBB 空间内。而后利用 OBB 坐标原点实现位移变换。这里，设定 OBB 中心顶点为 C；范围向量 $\mathbf{e} = (e_0, e_1, e_2)$；局部坐标轴 $\mathbf{u}_0$、$\mathbf{u}_1$、$\mathbf{u}_2$，则针对世界坐标系统中一点 P，其 OBB 空间内的表达为(x, y, z)，其中 $x = (P - C) \cdot \mathbf{u}_0$，$y = (P - C) \cdot \mathbf{u}_1$，$z = (P - C) \cdot \mathbf{u}_2$。

令线段的中心点 $M = (m_x, m_y, m_z)$，两个端点为 $M - \mathbf{d}$ 和 $M + \mathbf{d}$，其中 $\mathbf{d} = (d_x, d_y, d_z)$为线段的方向向量，且$\|\mathbf{d}\|$为向量长度值的一半。同时，该线段投影至经过原点的一分离轴上将产生一个投影区间，其中心点位于（$\mathbf{v}$ 向上）距原点 $d_s = (M \cdot \mathbf{v})/\|\mathbf{v}\|$有符号距离处，且半径 $r_s = |\mathbf{d} \cdot \mathbf{v}|/\|\mathbf{v}\|$。除此之外，令 r_b 为盒体在向量 $\mathbf{v}$ 上投影区间的半径，当且仅当$|d_s| > r_b + r_s$ 时（如图 5.23 所示），向量 $\mathbf{v}$ 即为分离轴。

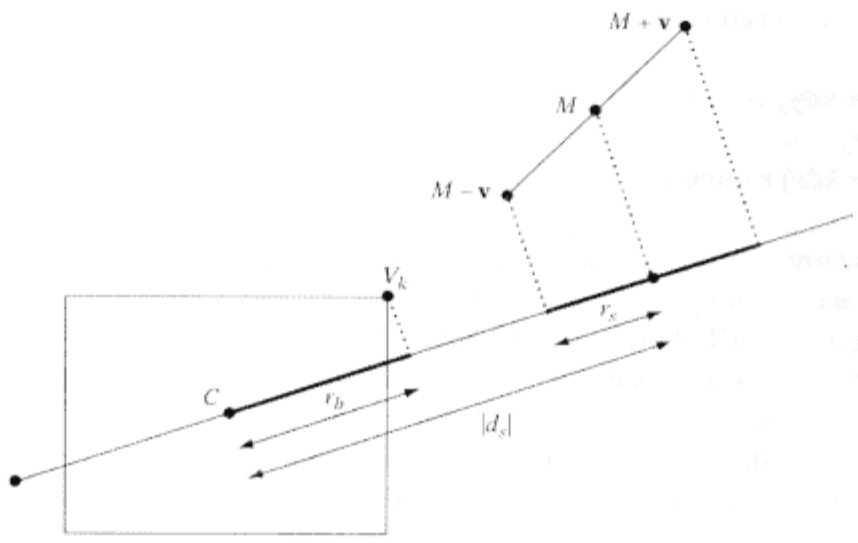


图 5.23　采用分离轴方案实现线段与 AABB 之间的相交测试

下面将考查由 3 个正交单位向量且半长值为 e_0、e_1、e_2 构成的 OBB，其投影区间半径如下所示：

$$r_b=(e_0|\mathbf{u}_0 \cdot \mathbf{v}| + e_1|\mathbf{u}_1 \cdot \mathbf{v}| + e_2|\mathbf{u}_2 \cdot \mathbf{v}|)/\|\mathbf{v}\|$$

将 $\mathbf{u}_0=(1, 0, 0)$、$\mathbf{u}_1=(0, 1, 0)$、$\mathbf{u}_2=(0, 0, 1)$代入上式，可得到 AABB 版本的表达式：

$$r_b=(e_0|v_x| + e_1|v_y|+ e_2|v_z|)/\|\mathbf{v}\|$$

其中，需要针对 6 个轴执行分离轴测试：3 条 AABB 上的面法线（$\mathbf{v}_0=(1, 0, 0)$，$\mathbf{v}_1=(0, 1, 0)$，$\mathbf{v}_2=(0, 0, 1)$）；线段方向向量与面法线 3 个叉积向量（$\mathbf{v}_3=\mathbf{d}\times(1, 0, 0)$，$\mathbf{v}_4=\mathbf{d}\times(0, 1, 0)$，$\mathbf{v}_5=\mathbf{d}\times(0, 0, 1)$）。对于上述 d_s、r_b、r_s 且$|d_s| > r_b+ r_s$，表 5.1 显示了相应的结果表达式。

表 5.1　线段和 AABB 间的 6 个分离轴相交测试

v_i	d_s	r_b	r_s	$\|d_s\|>r_b+r_s$
(1,0,0)	m_x	e_0	$\|d_x\|$	$\|m_x\|>e_0+\|d_x\|$
(0,1,0)	m_y	e_1	$\|d_y\|$	$\|m_y\|>e_1+\|d_y\|$
(0,0,1)	m_z	e_2	$\|d_z\|$	$\|m_z\|>e_2+\|d_z\|$
$\mathbf{d}\times(1,0,0)$	$(m_y d_z - m_z d_y)/\|\mathbf{d}\|$	$(e_1\|d_z\|+e_2\|d_y\|)/\|\mathbf{d}\|$	0	$\|m_y d_z - m_z d_y\|>e_1\|d_z\|+e_2\|d_y\|$
$\mathbf{d}\times(0,1,0)$	$(m_z d_x - m_x d_z)/\|\mathbf{d}\|$	$(e_0\|d_z\|+e_2\|d_x\|)/\|\mathbf{d}\|$	0	$\|m_z d_x - m_x d_z\|>e_0\|d_z\|+e_2\|d_x\|$
$\mathbf{d}\times(0,0,1)$	$(m_x d_y - m_y d_x)/\|\mathbf{d}\|$	$(e_0\|d_y\|+e_1\|d_x\|)/\|\mathbf{d}\|$	0	$\|m_x d_y - m_y d_x\|>e_0\|d_y\|+e_1\|d_x\|$

参照表 5.1 给出的分离轴表达式，可生成下列高效的实现代码：

```
// Test if segment specified by points p0 and p1 intersects AABB b
int TestSegmentAABB(Point p0, Point p1, AABB b)
{
    Point c = (b.min + b.max) * 0.5f;        // Box center-point
    Vector e = b.max - c;                    // Box halflength extents
    Point m = (p0 + p1) * 0.5f;              // Segment midpoint
    Vector d = p1 - m;                       // Segment halflength vector
    m = m - c;                               // Translate box and segment to origin
    // Try world coordinate axes as separating axes
    float adx = Abs(d.x);
    if (Abs(m.x) > e.x + adx) return 0;
    float ady = Abs(d.y);
    if (Abs(m.y) > e.y + ady) return 0;
    float adz = Abs(d.z);
    if (Abs(m.z) > e.z + adz) return 0;

    // Add in an epsilon term to counteract arithmetic errors when segment is
    // (near) parallel to a coordinate axis (see text for detail)
    adx += EPSILON; ady += EPSILON; adz += EPSILON;
    // Try cross products of segment direction vector with coordinate axes
    if (Abs(m.y * d.z - m.z * d.y) > e.y * adz + e.z * ady) return 0;
    if (Abs(m.z * d.x - m.x * d.z) > e.x * adz + e.z * adx) return 0;
    if (Abs(m.x * d.y - m.y * d.x) > e.x * ady + e.y * adx) return 0;
    // No separating axis found; segment must be overlapping AABB
    return 1;
}
```

代码中，可以去除 **e**、**d**、**m** 中的系数 0.5，则前 5 行变量初始化代码简化为：

```
Vector e = b.max - b.min;
Vector d = p1 - p0;
Point m = p0 + p1 - b.min - b.max;
```

最后要探讨的问题是代码的健壮性问题，尤其是当线段的方向向量平行于某一坐标轴时，将会导致上述 3 个叉积向量的计算结果为 0 向量。在代码中，如果线段不与 AABB 相交，则前 3 个 if 语句将会检测到这一状态；如果线段与 AABB 相交，则最后 3 个 if 语句将出现“0 > 0”这一测试状态。另外，为了避免舍入误差导致的对分离轴的错误识别，代码中分别将 **adx**、**ady**、**adz** 项增加了一个较小的 EPSILON 值并使相应的比较操作产生某种偏差，该方案类似于第 4 章中 OBB-OBB 测试时所采用的方法。

5.3.4　直线与三角形之间的相交测试

直线（以及光线和线段）与三角形之间的测试是一类十分常见的测试类型。因此，在本节以及后续两节中，将对该测试问题进行详细的讨论。下面，首先进行直线与三角形之间的相交测试。给定三角形 *ABC* 以及经过两点 *P*、*Q* 的直线，如果直线与 *ABC* 平面的交点 *R* 位于三角形 *ABC* 的内部，则直线与三角形相交（如图 5.24 所示）。所以，解决这类相交问题的一种算法即是计算交点 *R*，并执行点-三角形包含测试。值得注意的是，当三角形 *ABC* 顶点依据某种给定视见方向呈逆时针排列时，若点 R 始终位于三角形各边 *AB*、*BC* 和 *CA*（这里，各边皆视为有向线段）的左边，则点 *R* 位于三角形内部。类似地，若三角形各顶点呈顺时针排列且点 *R* 位于各边的右边，则点 *R* 位于三角形内部。

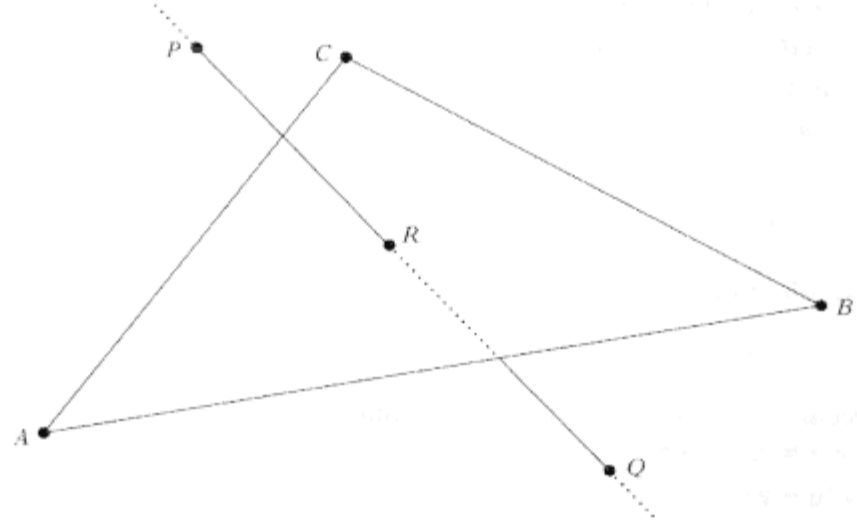


图 5.24　直线 PQ 与三角形 ABC 之间的相交测试

一般情况下，在计算过程中并不显式地计算 *R* 的具体位置，而是直接将直线 *PQ* 与三角形各边进行测试。考虑下列标量三重积：

$$u = [PQ\ PC\ PB]$$
$$v = [PQ\ PA\ PC]$$
$$w = [PQ\ PB\ PA]$$

若三角形 *ABC* 呈逆时针环绕方向，当 *PQ* 位于边 *BC*、*CA* 和 *AB* 的左侧时，则 $u \geqslant 0$，$v \geqslant 0$，$w \geqslant 0$

分别返回真值。类似地，若三角形 ABC 呈顺时针环绕方向，当 PQ 位于上述边的右侧时，标量三重积将返回非负值。另外，考虑到观察角度的不同，双面三角形将分别呈顺时针和逆时针环绕方向。这里，若标量三重积具备相同的符号（此处忽略 0 值），则 PQ 将穿越三角形内部。

在计算三角形 ABC 的交点时，u、v 和 w 将分别正比于 u^*，v^*和 w^*，如下所示：

$$u^* = ku = [PR\ PC\ PB]$$
$$v^* = kv = [PR\ PA\ PC]$$
$$w^* = kw = [PR\ PB\ PA]$$

其中，$k = \|PR\|/\|PQ\|$。

这里，u^*，v^*和 w^*正比于四面体 $RBCP$、$RCAP$ 和 $RABP$ 的体积。由于上述四面体具有相同的高度值（可参见 3.4 节中关于质心坐标的讨论），因而体积值分别正比于底面三角形 RBC、RCA 和 RAB。不难看出，u^*，v^*和 w^*（更为重要的是 u、v 和 w）可直接用于计算 R 的质心坐标。基于此，下列代码将针对直线 PQ 和逆时针三角形 ABC 进行测试，并计算交点的质心坐标：

```
//Given line pq and ccw triangle abc, return whether line pierces triangle. If
//so, also return the barycentric coordinates (u,v,w) of the intersection point
int IntersectLineTriangle(Point p, Point q, Point a, Point b, Point c,
                          float &u, float &v, float &w)
{
    Vector pq = q - p;
    Vector pa = a - p;
    Vector pb = b - p;
    Vector pc = c - p;
    // Test if pq is inside the edges bc, ca and ab. Done by testing
    // that the signed tetrahedral volumes, computed using scalar triple
    // products, are all positive
    u = ScalarTriple(pq, pc, pb);
    if (u < 0.0f) return 0;
    v = ScalarTriple(pq, pa, pc);
    if (v < 0.0f) return 0;
    w = ScalarTriple(pq, pb, pa);
    if (w < 0.0f) return 0;

    // Compute the barycentric coordinates (u, v, w) determining the
    // intersection point r, r = u*a + v*b + w*c
    float denom = 1.0f / (u + v + w);
    u *= denom;
    v *= denom;
    w *= denom;     // w = 1.0f - u - v;
    return 1;
}
```

考虑到代码的健壮性，需单独处理直线 PQ 与三角形 ABC 平面重合这一情形。其中，当 $u = v = w = 0$ 时，直线 PQ 位于三角形 ABC 的平面内部，若不加以处理，将会导致除 0 错误。

另外，通过重组标量三重积中的相关项并共享同一叉积结果，上述代码可得到进一步的优化，如下所示：

```
Vector m = Cross(pq, pc);
u = Dot(pb, m);          // ScalarTriple(pq, pc, pb);
if (u < 0.0f) return 0;

v = -Dot(pa, m);         // ScalarTriple(pq, pa, pc);
if (v < 0.0f) return 0;
w = ScalarTriple(pq, pb, pa);
if (w < 0.0f) return 0;
```

针对双面三角形，代码可改写为：

```
Vector m = Cross(pq, pc);
u = Dot(pb, m);          // ScalarTriple(pq, pc, pb);
v = -Dot(pa, m);         // ScalarTriple(pq, pa, pc);
if (!SameSign(u, v)) return 0;
w = ScalarTriple(pq, pb, pa);
if (!SameSign(u, w)) return 0;
```

测试中的相关运算还可改写为：

```
Vector m = Cross(pq, p);
u = Dot(pq, Cross(c, b)) + Dot(m, c - b);
v = Dot(pq, Cross(a, c)) + Dot(m, a - c);
w = Dot(pq, Cross(b, a)) + Dot(m, b - a);
```

或等价于下列内容：

```
Vector m = Cross(pq, q);
float s = Dot(m, c - b);
float t = Dot(m, a - c);
u = Dot(pq, Cross(c, b)) + s;
v = Dot(pq, Cross(a, c)) + t;
w = Dot(pq, Cross(b, a)) - s - t;
```

对于一个既定的三角形，如同边向量 $\mathbf{e}_0 = C - B$，$\mathbf{e}_1 = A - C$，相应的叉积 $C \times B$、$A \times C$、$B \times A$ 也可视为常量。如果这一类数据与三角形同时保存，则运行时只需要一次叉积计算和 5 次点积计算。由于叉积结果值将取决于直线的相关数据，但其只需计算一次，即使该直线与多个三角形之间进行相交测试。另外，最后一种表达方式等同于：直线 PQ 与视为 Plücker 坐标直线的三角形各边之间的相交测试（[Amanatides97]）。关于 Plücker 坐标，[Erickson97]和[Shoemake98]中进行了简单扼要的介绍。虽然 Plücker 坐标方法能够适当地减少浮点数操作（假设不存在特定的点积或叉积指令），但实际操作过程中，与标量三重积方法相比，其额外的内存访问很可能导致运行速度的降低。同时，Plücker 坐标算法造成的较大的内存开销也限制了其应用范围。最后需要注意的是，三角形的面法线 $\mathbf{n} = (B - A) \times (C - A)$可以通过上述已保存的 3 个“常量”叉积向量获得，且记为 $\mathbf{n} = -(C \times B) - (A \times C) - (B \times A)$。

这里需要补充说明的是，当直线 PQ 与三角 ABC 相交时，可以通过显式计算直线 $L(t) = P + t(Q - P)$ 与 ABC 平面的相交 t 值，从而将上述测试类型扩展至线段的相交测试，并且丢弃 $0 \leqslant t \leqslant 1$ 之外的相交状态。5.3.6 节将给出另一种线段-三角形测试算法。

另外，通过计算直线与三角形各边之间的位置关系，可以方便地确定 3D 环境下直线或线段的交点是否位于三角形的内部——这是在进行三角形相交测试时较为健壮的一类测试方案。关于这一类内在健壮性问题，11.3.3 节将给出详细的讨论。

在上述相关内容中，曾利用了标量三重积计算交点与三角形各边的位置关系，该算法也可应用于任意凸多边形测试中，但基于效率方面的考虑，对直线-平面交点执行多边形内部包含测试将是一类较好的测试方法。标量三重积方法较好地解决了四边形相交问题，可参见后续小节内容。

5.3.5 直线与四边形之间的相交测试

5.3.4 节描述的标量三重积算法可以直接应用于直线与四边形 *ABCD* 之间的交点 *R* 测试中。这里，假设 *ABCD* 为凸四边形且顶点按逆时针排列。若一点位于 *ABCD* 内部，则必定位于三角形 *ABC* 或三角形 *DAC* 内部。

由于边 *CA* 为两个三角形的公共边，对该边首先进行测试可以有效地确定交点 *R* 所处的三角形区域。例如，如果交点 *R* 位于边 *CA* 的左方，则 *R* 必定处于三角形 *DAC* 的外部，因此，只需针对三角形 *ABC* 加以测试并计算其中的两条剩余边。如果交点 *R* 位于边 *CA* 的右方，则一定位于三角形 *ABC* 的外部且只需测试三角形 *DAC* 及其另外两条边。这里，交点 *R* 与边 *CA* 的位置依赖关系实现了对任意三角形的指派。无论交点 *R* 位于边 *CA* 的哪一侧，都只需测试 3 条边。依据浮点操作的相关特征，直线与四边形的测试消耗此处将等价于直线与三角形之间的测试。

但此相交测试方法无法直接返回交点的质心坐标，亦即应确定质心坐标所关联的某一个三角形。换句话说，坐标数据需要依据三角形 *ABC* 计算得到，即使交点位于三角形 *DAC* 内。在下列代码实现中，函数将计算并返回交点信息而非质心坐标：

```
// Given line pq and ccw quadrilateral abcd, return whether the line
// pierces the triangle. If so, also return the point r of intersection
int IntersectLineQuad(Point p, Point q, Point a, Point b, Point c, Point d, Point &r)
{
    Vector pq = q - p;
    Vector pa = a - p;
    Vector pb = b - p;
    Vector pc = c - p;
    // Determine which triangle to test against by testing against diagonal first
    Vector m = Cross(pc, pq);
    float v = Dot(pa, m);          // ScalarTriple(pq, pa, pc);
    if (v >= 0.0f) {
        // Test intersection against triangle abc
        float u = -Dot(pb, m);         // ScalarTriple(pq, pc, pb);
        if (u < 0.0f) return 0;
        float w = ScalarTriple(pq, pb, pa);
        if (w < 0.0f) return 0;
        // Compute r, r = u*a + v*b + w*c, from barycentric coordinates (u, v, w)
        float denom = 1.0f / (u + v + w);
        u *= denom;
        v *= denom;
```

```
        w *= denom;                    // w = 1.0f - u - v;
        r = u*a + v*b + w*c;
    } else {
        // Test intersection against triangle dac
        Vector pd = d - p;
        float u = Dot(pd, m);          // ScalarTriple(pq, pd, pc);
        if (u < 0.0f) return 0;
        float w = ScalarTriple(pq, pa, pd);
        if (w < 0.0f) return 0;
        v = -v;
        // Compute r, r = u*a + v*d + w*c, from barycentric coordinates (u, v, w)
        float denom = 1.0f / (u + v + w);
        u *= denom;
        v *= denom;
        w *= denom;                    // w = 1.0f - u - v;
        r = u*a + v*d + w*c;
    }
    return 1;
}
```

该特定相交测试算法也可应用于凹四边形测试，且假定对角线完全位于四边形内部。一种有效的解决方法是：预计算并输出对角线某一相关端点作为对角线的首顶点，从而得到内部对角线 CA。然而，该算法无法适用于自相交四边形。

5.3.6　光线或线段与三角形之间的相交测试

对比 5.3.5 节描述的算法，[Möller97a]中给出了一种类似的方法用于处理光线或线段与三角形的相交问题。回忆一下，$T(u, v, w) = uA + vB + wC$ 定义了三角形 ABC 平面上的点，其中(u, v, w)为该点的质心坐标且有 $u + v + w = 1$。当且仅当该点的质心坐标满足 $0 \leqslant u$、v、$w \leqslant 1$，则点 T 位于三角形 ABC 内部。上述内容也可记作 $T(v, w) = A + v(B - A) + w(C - A)$，且 $u = 1 - v - w$。若 $v \geqslant 0$，$w \geqslant 0$，$v + w \leqslant 1$，则点 T 位于三角形 ABC 内。

令两端点 P、Q 构成的有向线段定义为 $R(t) = P + t(Q - P)$，$0 \leqslant t \leqslant 1$。通过设定 $T(v, w) = R(t)$可以求解 t、v、w，且随后确定这些变量是否位于相交状态所需的范围内。

$$T(v, w) = R(t) \Leftrightarrow \text{（初始表达式）}$$

$$A + v(B - A) + w(C - A) = P + t(Q - P) \Leftrightarrow \text{（替换参数表达式）}$$

$$(P - Q)t + (B - A)v + (C - A)w = P - A \quad \text{（重组相关项）}$$

这一推导过程构造出了一个 3 × 3 线性方程组，因此其矩阵形式可记为：

$$\begin{bmatrix} (P-Q) & (B-A) & (C-A) \end{bmatrix} \begin{bmatrix} t \\ v \\ w \end{bmatrix} = \begin{bmatrix} (P-A) \end{bmatrix}$$

其中相关向量作为列向量给出，则可利用克莱姆法则对 t、v、w 进行求解：

$$t = \det\begin{bmatrix} (P-A) & (B-A) & (C-A) \end{bmatrix} / \det\begin{bmatrix} (P-Q) & (B-A) & (C-A) \end{bmatrix}$$

$$v = \det[(P-Q) \quad (P-A) \quad (C-A)] / \det[(P-Q) \quad (B-A) \quad (C-A)]$$
$$w = \det[(P-Q) \quad (B-A) \quad (P-A)] / \det[(P-Q) \quad (B-A) \quad (C-A)]$$

由于 det[**a b c**] = **a** • (**b** × **c**)，则上式可简化为：

$$t = (P-A) \cdot \mathbf{n}/d$$
$$v = (C-A) \cdot \mathbf{e}/d$$
$$w = -(B-A) \cdot \mathbf{e}/d$$

其中：

$$\mathbf{n} = (B-A) \times (C-A)$$
$$d = (P-Q) \cdot \mathbf{n}$$
$$\mathbf{e} = (P-Q) \times (P-A)$$

需注意的是，若 $d < 0$，则线段背离指向该三角形；若 $d = 0$，则线段将平行于该三角形平面。下列代码显示了这一以不同方式实现的线段-三角形测试方案：

```
// Given segment pq and triangle abc, returns whether segment intersects
// triangle and if so, also returns the barycentric coordinates (u,v,w)
// of the intersection point
int IntersectSegmentTriangle(Point p, Point q, Point a, Point b, Point c,
                             float &u, float &v, float &w, float &t)
{
    Vector ab = b - a;
    Vector ac = c - a;
    Vector qp = p - q;

    // Compute triangle normal. Can be precalculated or cached if
    // intersecting multiple segments against the same triangle
    Vector n = Cross(ab, ac);

    // Compute denominator d. If d <= 0, segment is parallel to or points
    // away from triangle, so exit early
    float d = Dot(qp, n);
    if (d <= 0.0f) return 0;

    // Compute intersection t value of pq with plane of triangle. A ray
    // intersects iff 0 <= t. Segment intersects iff 0 <= t <= 1. Delay
    // dividing by d until intersection has been found to pierce triangle
    Vector ap = p - a;
    t = Dot(ap, n);
    if (t < 0.0f) return 0;
    if (t > d) return 0;          // For segment; exclude this code line for a ray test

    // Compute barycentric coordinate components and test if within bounds
    Vector e = Cross(qp, ap);
    v = Dot(ac, e);
    if (v < 0.0f || v > d) return 0;
    w = -Dot(ab, e);
```

```
    if (w < 0.0f || v + w > d) return 0;

    // Segment/ray intersects triangle. Perform delayed division and
    // compute the last barycentric coordinate component
    float ood = 1.0f / d;
    t *= ood;
    v *= ood;
    w *= ood;
    u = 1.0f - v - w;
    return 1;
}
```

可以看出，上述代码与[Möller97a]给出的相应方法略有不同，其中计算了三角形 *ABC* 的法线 **n**——法线在实现过程中通常是一类很有用的数据。

也可以从另一个角度考查该算法：首先仍然计算线段与三角形平面之间的交点 *S*，并通过求解其三角形质心坐标的方式执行点-三角形的包含测试（参见 3.4 节）。当计算交点 *S* 及其质心坐标时，可以复用某些计算过程，这其中也包括三角形面法线的计算。这一类结果可通过预计算的方式得到，并与三角形数据一同保存。

当采用预计算时，该算法可以得到进一步优化：即计算并保存三角形平面方程以及三角形各边的边面方程（经过三角形各边且垂直于三角形平面的 3 个平面，如图 5.25 所示）。通过将边面方程除以一个系数，该系数可以返回对面顶点（非边面上的顶点）距该面的距离，因而针对 *S* 的边面计算可直接得到 *S* 的质心坐标。需要注意的是，由于第 3 个顶点的质心坐标可以通过其他两个顶点获取，因而只须保存 3 个边面中的两个。综上所述，线段-三角形之间的相交测试可归结为 4 个平面方程的计算（以及一些辅助操作），如下列代码所示：

```
struct Triangle {
    Plane p;              // Plane equation for triangle plane
    Plane edgePlaneBC;    // When evaluated gives barycentric weight u (for vertex A)
    Plane edgePlaneCA;    // When evaluated gives barycentric weight v (for vertex B)
};

// Given segment pq and precomputed triangle tri, returns whether segment intersects
// triangle. If so, also returns the barycentric coordinates (u,v,w) of the
// intersection point s, and the parameterized intersection t value
int IntersectSegmentTriangle(Point p, Point q, Triangle tri,
                             float &u, float &v, float &w, float &t, Point &s)
{
    // Compute distance of p to triangle plane. Exit if p lies behind plane
    float distp = Dot(p, tri.p.n) – tri.p.d;
    if (distp < 0.0f) return 0;

    // Compute distance of q to triangle plane. Exit if q lies in front of plane
    float distq = Dot(q, tri.p.n) – tri.p.d;
    if (distq >= 0.0f) return 0;

    // Compute t value and point s of intersection with triangle plane
```

```
    float denom = distp – distq;
    t = distp / denom;
    s = p + t * (q – p);

    // Compute the barycentric coordinate u; exit if outside 0..1 range
    u = Dot(s, tri.edgePlaneBC.n) – tri.edgePlaneBC.d;
    if (u < 0.0f || u > 1.0f) return 0;
    // Compute the barycentric coordinate v; exit if negative
    v = Dot(s, tri.edgePlaneCA.n) – tri.edgePlaneCA.d;
    if (v < 0.0f) return 0;
    // Compute the barycentric coordinate w; exit if negative
    w = 1.0f – u – v;
    if (w < 0.0f) return 0;

    // Segment intersects tri at distance t in position s (s = u*A + v*B + w*C)
    return 1;
}
```

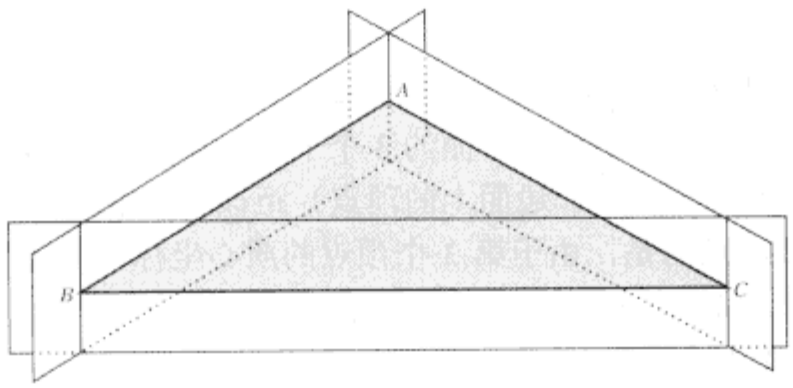


图 5.25　垂直于 *ABC* 平面并穿越三角形 *ABC* 各边的平面

代码中，只在线段与三角形真正相交时，才执行 **denom** 除法操作，从而推迟了（较耗时的）除法计算。

针对三角形 *ABC*，其相应的数据结构变量 **tri** 可按下列方式加以初始化：

```
Triangle tri;
Vector n = Cross(b – a, c – a);
tri.p = Plane(n, a);
tri.edgePlaneBC = Plane(Cross(n, c – b), b);
tri.edgePlaneCA = Plane(Cross(n, a – c), c);
```

若利用边面计算顶点的质心坐标，边面方程须除以一个系数，且该系数返回对面顶点距该面的距离（非边面顶点）：

```
tri.edgePlaneBC *= 1.0f / (Dot(a, tri.edgePlaneBC.n) – tri.edgePlaneBC.d);
tri.edgePlaneCA *= 1.0f / (Dot(b, tri.edgePlaneCA.n) – tri.edgePlaneCA.d);
```

至此，三角形初始化完毕并可以对相关数据加以保存。需注意的是，该数据结构使得相交测试不再针对三角形顶点。与包含 3 个顶点、9 个浮点数的三角形存储格式相比，当前三角形数据结构则需要 12 个浮点分量。

5.3.7　光线或线段与圆柱体之间的相交测试

任意方向上的、无限延伸的圆柱体可以按照下列方式加以定义：由顶点 P、Q 定义的一条直线以及半径 r。若 X 为圆柱体表面上一点，则需要满足下列隐式方程：

$$(\mathbf{v}-\mathbf{w})\cdot(\mathbf{v}-\mathbf{w})-r^2=0，其中，\mathbf{v}=X-P，\mathbf{d}=Q-P，\mathbf{w}=\frac{\mathbf{v}\cdot\mathbf{d}}{\mathbf{d}\cdot\mathbf{d}}\mathbf{d}$$

上述方程表明，若从 P 至顶点 X 表示的向量 $\mathbf{v}$ 中减去 $\mathbf{w}$（$\mathbf{v}$ 中平行于 $\mathbf{d}$ 的分量），则针对 X 的结果向量，其长度必定等于 r 且位于圆柱体面上。方程中使用了平方值而非根计算，从而简化了后续的处理过程。

对于由顶点 A、B 定义的直线 $L(t)=A+t(B-A)$（如图 5.26 所示），其与圆柱体的交点可将 $L(t)$ 代入上式中的 X 项并求解 t 计算得到。同时，将 $\mathbf{v}=L(t)-P=(A-P)+t(B-A)$ 记为 $\mathbf{v}=\mathbf{m}+t\,\mathbf{n}$，即 $\mathbf{m}=A-P$，$\mathbf{n}=B-A$。经过相关计算，上述方程转化为：

$$\left(\mathbf{n}\cdot\mathbf{n}-\frac{(\mathbf{n}\cdot\mathbf{d})^2}{\mathbf{d}\cdot\mathbf{d}}\right)t^2+2\left(\mathbf{m}\cdot\mathbf{n}-\frac{(\mathbf{n}\cdot\mathbf{d})(\mathbf{m}\cdot\mathbf{d})}{\mathbf{d}\cdot\mathbf{d}}\right)t+\mathbf{m}\cdot\mathbf{m}-\frac{(\mathbf{m}\cdot\mathbf{d})^2}{\mathbf{d}\cdot\mathbf{d}}-r^2=0$$

其中，可以将方程的两边乘以 $\mathbf{d}\cdot\mathbf{d}$，从而消除多个 $\mathbf{d}\cdot\mathbf{d}$ 分母项：

$$((\mathbf{d}\cdot\mathbf{d})(\mathbf{n}\cdot\mathbf{n})-(\mathbf{n}\cdot\mathbf{d})^2)t^2+2((\mathbf{d}\cdot\mathbf{d})(\mathbf{m}\cdot\mathbf{n})-(\mathbf{n}\cdot\mathbf{d})(\mathbf{m}\cdot\mathbf{d}))t+(\mathbf{d}\cdot\mathbf{d})((\mathbf{m}\cdot\mathbf{m})-r^2)-(\mathbf{m}\cdot\mathbf{d})^2=0$$

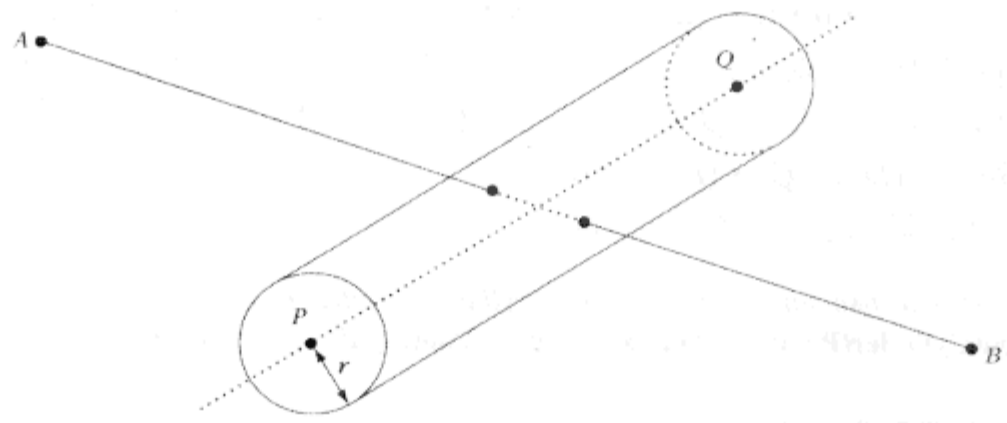


图 5.26　由顶点 A 和 B 定义的直线或线段与圆柱体之间的相交测试，且圆柱体由顶点 P 和 Q 以及半径 r 加以定义

需要注意的是，虽然将二次方程的两边乘以 $\mathbf{d}\cdot\mathbf{d}$ 可以避免成本高昂的除法操作，但考虑到算法自身的健壮性问题，这仍然会产生某些异常情况（基于中间项的计算精度）。另外，在理想情况下，圆柱体的方向向量 $\mathbf{d}$ 最好以单位向量的形式给出，从而消去 $\mathbf{d}\cdot\mathbf{d}$ 项（确切地讲，如果 $\mathbf{d}$ 为单位向量，圆柱体的面方程可简单地表示为 $(\mathbf{v}\times\mathbf{d})\cdot(\mathbf{v}\times\mathbf{d})=r^2$）。

无论采用哪一类方式，关于 t 的二次方程都可以表达为 $at^2+2bt+c=0$。其中：

$$a=(\mathbf{d}\cdot\mathbf{d})(\mathbf{n}\cdot\mathbf{n})-(\mathbf{n}\cdot\mathbf{d})^2$$
$$b=(\mathbf{d}\cdot\mathbf{d})(\mathbf{m}\cdot\mathbf{n})-(\mathbf{n}\cdot\mathbf{d})(\mathbf{m}\cdot\mathbf{d})$$
$$c=(\mathbf{d}\cdot\mathbf{d})((\mathbf{m}\cdot\mathbf{m})-r^2)-(\mathbf{m}\cdot\mathbf{d})^2$$

基于拉格朗日恒等式，上述项可写为：

$$a=(\mathbf{d}\times\mathbf{n})\cdot(\mathbf{d}\times\mathbf{n})$$
$$b=(\mathbf{d}\times\mathbf{m})\cdot(\mathbf{d}\times\mathbf{n})$$

$$c = (\mathbf{d} \times \mathbf{m}) \cdot (\mathbf{d} \times \mathbf{m}) - r^2(\mathbf{d} \cdot \mathbf{d})$$

上式表明：若 $a = 0$，则 **d** 与 **n** 平行且 a 通常为正值（浮点数运算可能会产生少量误差，从而导致 a 为负值）。而且，c 的符号常用于指出顶点 A 位于圆柱体表面的内部（$c < 0$）还是外部（$c > 0$）。这里求解 t 的方式为：

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a}$$

其中，判别式 b^2-ac 决定了二次方程中实根的数量。若判别式为负值，则方程无实根，即直线不与圆柱体相交；若判别式为正值，则方程有两个实根，且较小根对应直线进入圆柱面时的 t 值，较大根对应直线离开圆柱面时的 t 值；若判别式为 0，则方程只有一个根，即直线在 t 值处与圆柱面相切于一点。

与无限延伸的圆柱体相比较而言，当确定常规圆柱体的相交状态时，首先，不考虑底面（分别位于 P、Q 处且垂直于方向 **d**）范围之外的相交测试。若（方向）直线并非背离指向底面平面，则需要进行该平面的相交计算，进而执行圆柱体的面相交测试。这里给定一个有效的 t 值，若$(L(t) - P) \cdot \mathbf{d} < 0$ 或$(\mathbf{m} \cdot \mathbf{d}) + t(\mathbf{n} \cdot \mathbf{d}) < 0$，则 $L(t)$位于底面 P 的外部。当 $L(t)$位于圆柱体的外部时，若 $\mathbf{n} \cdot \mathbf{d} \leqslant 0$，则 L 背离或平行于底面 P。仅当 $\mathbf{n} \cdot \mathbf{d} > 0$ 时才需执行底面 P 的相交测试。令底面 P 的平面方程为$(X - P) \cdot \mathbf{d} = 0$，将 $L(t)$代入 X 并计算平面交点处的 t 值为：$t = -(\mathbf{m} \cdot \mathbf{d})/(\mathbf{n} \cdot \mathbf{d})$。若$(L(t) - P) \cdot (L(t) - P) \leqslant r^2$，则交点位于底面内部。

类似地，当$(L(t) - P) \cdot \mathbf{d} > \mathbf{d} \cdot \mathbf{d}$ 或$(\mathbf{m} \cdot \mathbf{d}) + t(\mathbf{n} \cdot \mathbf{d}) > \mathbf{d} \cdot \mathbf{d}$ 时，$L(t)$位于底面 Q 平面的外部。由于 $L(t)$位于圆柱体底面 Q 的外部，则仅当 $\mathbf{n} \cdot \mathbf{d} < 0$ 时才进行底面 Q 的相交测试；当 $\mathbf{n} \cdot \mathbf{d} \geqslant 0$ 时，直线与圆柱体之间不存在相交。此处，底面 Q 的平面方程为$(X - Q) \cdot \mathbf{d} = 0$，且交点处的 t 值为：$t = ((\mathbf{d} \cdot \mathbf{d}) - (\mathbf{m} \cdot \mathbf{d}))/(\mathbf{n} \cdot \mathbf{d})$。另外，当$(L(t) - Q) \cdot (L(t) - Q) \leqslant r^2$ 时，交点位于底面内部。针对线段与常规圆柱体之间的相交测试，下列代码显示了上述分析过程：

```
// Intersect segment S(t)=sa+t(sb-sa), 0<=t<=1 against cylinder specified by p, q and r
int IntersectSegmentCylinder(Point sa, Point sb, Point p, Point q, float r, float &t)
{
    Vector d = q – p, m = sa – p, n = sb – sa;
    float md = Dot(m, d);
    float nd = Dot(n, d);
    float dd = Dot(d, d);
    // Test if segment fully outside either endcap of cylinder
    if (md < 0.0f && md + nd < 0.0f) return 0;  // Segment outside 'p' side of cylinder
    if (md > dd && md + nd > dd) return 0;      // Segment outside 'q' side of cylinder
    float nn = Dot(n, n);
    float mn = Dot(m, n);
    float a = dd * nn – nd * nd;
    float k = Dot(m, m) – r * r;
    float c = dd * k – md * md;
    if (Abs(a) < EPSILON) {
        // Segment runs parallel to cylinder axis
        if (c > 0.0f) return 0;                 // 'a' and thus the segment lie outside cylinder
        // Now known that segment intersects cylinder; figure out how it intersects
        if (md < 0.0f) t = -mn / nn;            // Intersect segment against 'p' endcap
```

```
        else if (md > dd) t = (nd - mn) / nn;      // Intersect segment against 'q' endcap
        else t = 0.0f;                             // 'a' lies inside cylinder
        return 1;
    }
    float b = dd * mn – nd * md;
    float discr = b * b – a * c;
    if (discr < 0.0f) return 0;                    // No real roots; no intersection
    t = (-b – Sqrt(discr)) / a;
    if (t < 0.0f || t > 1.0f) return 0;            // Intersection lies outside segment
    if (md + t * nd < 0.0f) {
        // Intersection outside cylinder on 'p' side
        if (nd <= 0.0f) return 0;                  // Segment pointing away from endcap
        t = -md / nd;
        // Keep intersection if Dot(S(t) - p, S(t) - p) <= r∧2
        return k + 2 * t * (mn + t * nn) <= 0.0f;
    } else if (md + t * nd > dd) {
        // Intersection outside cylinder on 'q' side
        if (nd >= 0.0f) return 0;                  // Segment pointing away from endcap
        t = (dd – md) / nd;
        // Keep intersection if Dot(S(t) - q, S(t) - q) <= r∧2
        return k + dd – 2 * md + t * (2 * (mn – nd) + t * nn) <= 0.0f;
    }
    // Segment intersects cylinder between the endcaps; t is correct
    return 1;
}
```

对上述算法稍加调整即可用于光线（非线段）与常规圆柱体之间的相交测试。同时，针对胶囊体的相交测试也可以利用类似的方法，其中底面相交测试变为底半球相交测试。[Shene94]、[Cychosz94]和[Held97]分别对直线、光线和线段与圆柱体之间的相交测试进行了相关的阐述。

5.3.8　光线或线段与凸多面体之间的相交测试

当在线段 $S(t) = A + t(B - A)$（其中 $0 \leqslant t \leqslant 1$）与凸多面体之间执行相交测试时，关于凸多面体，一种较为简洁的表达方式是将其看作半空间集合的交集（[Haines91b]）。其中，若存在 t 值，$0 \leqslant t \leqslant 1$ 且 $S(t)$位于半空间集合内部，则二者相交。基于此，一种高效的方法是针对每一个半空间对线段实施剪裁处理，即去除半空间外线段上的某一部分。若经过这一剪裁处理后，剩余线段长度为 0，则线段不与多面体相交且测试退出并返回“不相交”这一信息。否则，剪裁后的剩余线段即为二者的交集。

剪裁处理将考查区间为 $t_{first} \leqslant t \leqslant t_{last}$ 的某一线段，其中 t_{first}、t_{last} 分别设定为 0 和 1。当线段与半空间平面相交时，则针对 t 值进行相应的调整（此处，光线 t_{last} 初始化为$+\infty$，直线 t_{first} 初始化为$-\infty$）。

回忆一下，对于线段 $S(t) = A + t(B - A)$（其中 $0 \leqslant t \leqslant 1$）以及平面 $\mathbf{n} \cdot X = d$，二者相交时的 t 值为：$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A))$。因此，若判别式 $\mathbf{n} \cdot (B - A)$为负值，则平面面向线段；若判别式为正值，则平面背向线段；若判别式为 0，则二者平行。当平面面向线段时，参数值 t 对应线段进入半空间时交点处的 t 值。其结果为：若 t 大于 t_{first}，则 t_{first} 为当前 t 值且线段被剪裁。当平面背离线段时，t 值对应线段离开表空间时交点处的 t 值。因此，若 t 小于 t_{last}，则 t_{last} 为当前 t 值。最后，若 $t_{last} < t_{first}$，则不存在

线段并且也不存在交点。

这里，需要对与平面平行的线段单独加以处理。如果线段位于半空间的外部且平行于相应平面，则不与多面体相交，测试即刻退出并返回“不相交”这一信息。若该线段位于半空间内部，则可以简单地忽略当前测试平面，并直接处理下一个平面。其中，针对线段与多面体之间的内部/外部包含测试，可将线段的两端点分别代入平面方程，并比较二者的符号值以判断端点相对于平面的位置关系。

图 5.27 显示了 2D 环境下线段与多个半空间之间的剪裁过程，以及经半空间剪裁后光线的逻辑交集所体现的相交状态。

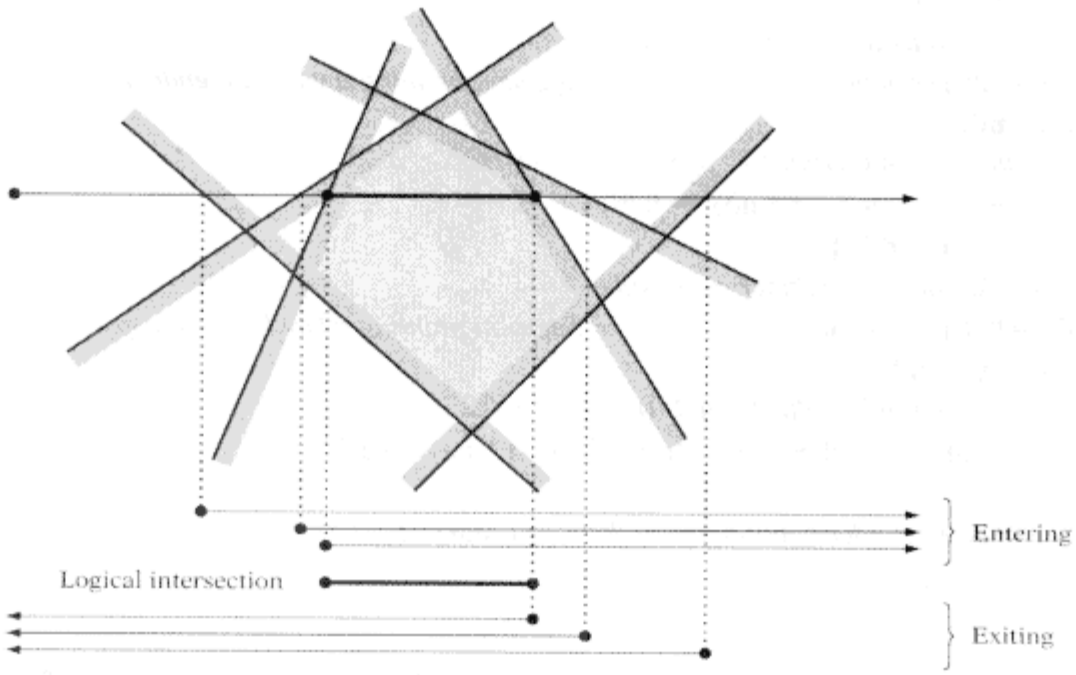


图 5.27　光线（或线段）与凸多面体（定义为一组半空间交集）之间的相交测试可视为：经全部半空间剪裁后的光线的逻辑交集

下列代码实现了上述测试过程：

```
// Intersect segment S(t)=A+t(B-A), 0<=t<=1 against convex polyhedron specified
// by the n halfspaces defined by the planes p[]. On exit tfirst and tlast
// define the intersection, if any
int IntersectSegmentPolyhedron(Point a, Point b, Plane p[], int n,
                               float &tfirst, float &tlast)
{
    // Compute direction vector for the segment
    Vector d = b – a;
    // Set initial interval to being the whole segment. For a ray, tlast should be
    // set to +FLT_MAX. For a line, additionally tfirst should be set to –FLT_MAX
    tfirst = 0.0f;
    tlast = 1.0f;
    // Intersect segment against each plane
    for (int i = 0; i<n; i++) {
        float denom = Dot(p[i].n, d);
        float dist = p[i].d - Dot(p[i].n, a);
        // Test if segment runs parallel to the plane
        if (denom == 0.0f) {
```

```
                // If so, return "no intersection" if segment lies outside plane
                if (dist > 0.0f) return 0;
            } else {
                // Compute parameterized t value for intersection with current plane
                float t = dist / denom;
                if (denom < 0.0f) {
                    // When entering halfspace, update tfirst if t is larger
                    if (t > tfirst) tfirst = t;
                } else {
                    // When exiting halfspace, update tlast if t is smaller
                    if (t<tlast) tlast = t;
                }
                // Exit with "no intersection" if intersection becomes empty
                if (tfirst>tlast) return 0;
            }
        }
        // A nonzero logical intersection, so the segment intersects the polyhedron
        return 1;
}
```

综上所述，该问题类似于线段与 3 组 slab 定义的盒体间的相交测试（参见 5.3.3 节）——显然，盒体是凸多面体的一个特例。因此，上述代码也可应用于线段-盒体测试。

5.4　其他类型的测试

虽然本节讨论的测试计算并不属于前述章节所划分的相交测试范畴，但这一类问题经常会出现，并且对其进行分析将有助于对相关问题的处理。

5.4.1　点与多边形之间的测试

点与多边形之间的包含测试存在多种算法，其中，[Haines94]中提供了一个较好的点-多边形包含测试算法；另外，[Weiler94]中也提出了一个基于象限测试的算法；同时，在[Walker99]中讨论了一个基于 CSG 多边形表示的算法；最后，[Huang97]中也针对相关算法进行了较为全面的分析。

对于多数实时碰撞检测中的点-多边形测试，其测试范围一般只限于凸多边形。对于 n 边凹多边形的点包含测试，其时间消耗接近 $O(n)$；而通过采用区间半分算法，凸多边形的特定结构可使得点包含测试的时间消耗达到 $O(\log n)$级别。

这里，假设 n 边凸多边形顶点 $V_0,V_1,\cdots,V_{n-1}$ 呈逆时针排列。通过测试顶点 P 与方向直线 V_0、V_k（$k=\lfloor n/2 \rfloor$）的位置关系，可将多边形的测试范围缩至一半。该过程重复执行并相应地调整 k 值，直至发现顶点 P 位于多边形的外部或位于有向直线 $V_0V_k \sim V_0V_{k+1}$ 之间。对于后者，若顶点 P 位于有向直线 V_kV_{k+1} 的左侧，则其位于多边形内部。

这里，作为一个实例，将考查一个 8 条边的凸多边形，如图 5.28 所示。首先，顶点 P 针对直线 V_0V_4 进行测试，并将多边形顶点范围降至一半。由于顶点 P 位于 A 的左侧，则丢弃多边形的右半部分，并

继续对左半部分执行相同处理。下一步，在顶点 P 与直线 B 之间执行测试，最后对直线 C 进行测试。此时，顶点 P 位于邻接顶点构成的两条直线之间。最后，顶点 P 与该邻接顶点构成的直线进行测试。此处，测试结果为：顶点 P 位于多边形内部。

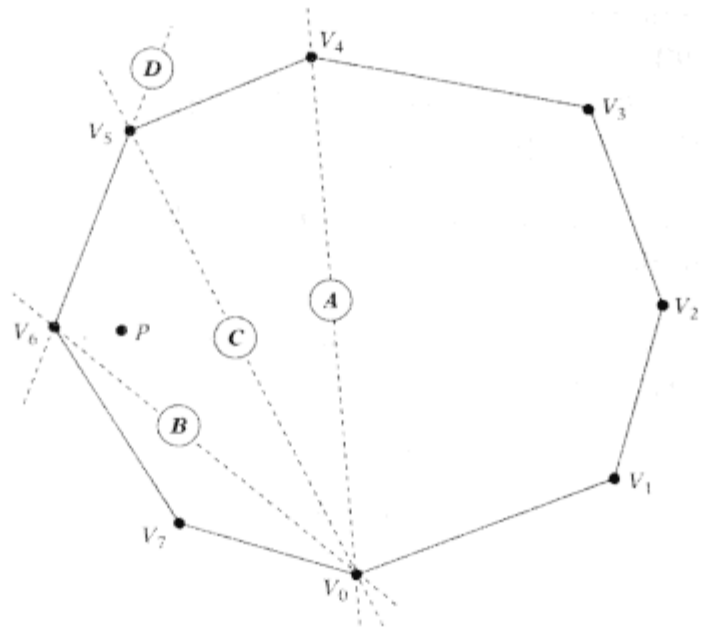


图 5.28　凸多边形顶点的二分搜索使得点 P 的包含测试的时间复杂度为 $O(\log n)$（这里测试了 4 条边 A～D）

另外，预计算函数 **TriangleIsCCW()**将计算三角形顶点是否以逆时针方向排列。该测试的高效实现方式如下列代码所示：

```
// Test if point p lies inside ccw-specified convex n-gon given by vertices v[]
int PointInConvexPolygon(Point p, int n, Point v[])
{
    // Do binary search over polygon vertices to find the fan triangle
    // (v[0], v[low], v[high]) the point p lies within the near sides of
    int low = 0, high = n;
    do {
        int mid = (low + high) / 2;
        if (TriangleIsCCW(v[0], v[mid], p))
            low = mid;
        else
            high = mid;
    } while (low + 1 < high);

    // If point outside last (or first) edge, then it is not inside the n-gon
    if (low == 0 || high == n) return 0;

    // p is inside the polygon if it is left of
    // the directed edge from v[low] to v[high]
    return TriangleIsCCW(v[low], v[high], p);
}
```

该点包含测试方法类似于[Preparata85]中所描述的相关算法结构，其中 n 边多边形内部也存在一点与此处的顶点 V_0 对应。

针对凹多边形亦存在一种 $O(n)$级别的测试方法，该算法将某一顶点作为源点并沿某一方向（通常为 x 正轴）投射光线，并计算该光线穿越多边形边界的次数。如果该光线穿越多边形边界一次，则顶点必定位于多边形的内部；如果穿越两次，则顶点必定位于多边形的外部，即光线先进入多边形而后离开。一般来讲，当穿越次数为奇数时，顶点位于多边形内部；穿越次数为偶数时，顶点位于多边形外部。该方法一般称作“穿越测试（crossings test）”，如图 5.29 所示。

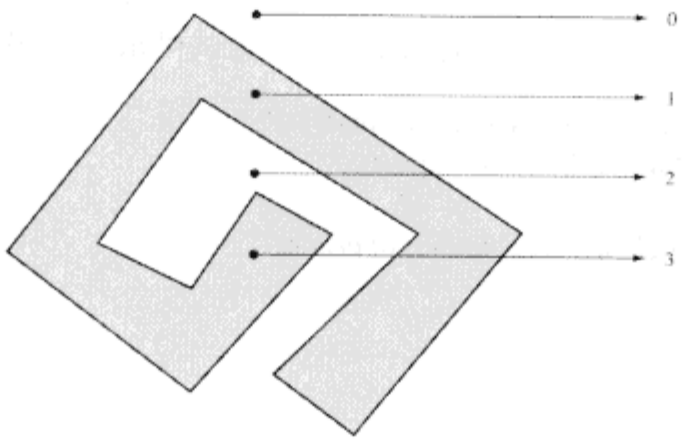


图 5.29　分别从 4 个不同的查询点投射光线。若光线穿越对象边界奇数次，则查询点位于多边形内部

另外，应正确处理下列情形：光线穿越至多边形顶点或光线与多边形某边重合。此时，若不加以正确处理，将会导致错误的穿越计数值。在实际应用中，可以将光线上的顶点移动一个极小量，从而改进上述问题。[Haines94]中列举了区别于该“穿越测试”算法的某些不同之处。

5.4.2　点与三角形之间的测试

点与三角形的包含测试是一类常见问题，因而有必要单独对其加以研究。当前，已存在两种有效的点-三角形包含测试算法：一种是 3.4 节中所讨论的未经处理的三角形（计算质心坐标），另一种是 5.3.6 小节中描述的经过预计算的三角形。

本节将讨论第 3 种算法，即针对三角形 ABC 及其平面内一点 P 进行相应的平移操作，使得顶点 P 位于原点。因此，测试转化为：测试原点是否位于平移后的三角形内部。

当且仅当三角形 PAB、PBC、PCA 皆呈现顺时针排列或逆时针排列时，顶点 P 位于三角形 ABC 内部。当前，由于视顶点 P 为原点，因此上述方法等价于计算 $\mathbf{u} = B \times C$，$\mathbf{v} = C \times A$，$\mathbf{w} = A \times B$ 是否指向同一方向，即测试是否 $\mathbf{u} \cdot \mathbf{v} \geqslant 0$ 且 $\mathbf{u} \cdot \mathbf{w} \geqslant 0$。测试的实现代码如下所示：

```
// Test if point P lies inside the counterclockwise triangle ABC
int PointInTriangle(Point p, Point a, Point b, Point c)
{
    // Translate point and triangle so that point lies at origin
    a -= p; b -= p; c -= p;
    // Compute normal vectors for triangles pab and pbc
    Vector u = Cross(b, c);
    Vector v = Cross(c, a);
    // Make sure they are both pointing in the same direction
    if (Dot(u, v) < 0.0f) return 0;
```

```
    // Compute normal vector for triangle pca
    Vector w = Cross(a, b);
    // Make sure it points in the same direction as the first two
    if (Dot(u, w) < 0.0f) return 0;
    // Otherwise P must be in (or on) the triangle
    return 1;
}
```

与点积计算相比，当叉积操作成本较为高昂时，该测试可依据拉格朗日恒等式记为 5 点积形式：

```
// Test if point P lies inside the counterclockwise 3D triangle ABC
int PointInTriangle(Point p, Point a, Point b, Point c)
{
    // Translate point and triangle so that point lies at origin
    a -= p; b -= p; c -= p;
    float ab = Dot(a, b);
    float ac = Dot(a, c);
    float bc = Dot(b, c);
    float cc = Dot(c, c);
    // Make sure plane normals for pab and pbc point in the same direction
    if (bc * ac – cc * ab < 0.0f) return 0;
    // Make sure plane normals for pab and pca point in the same direction
    float bb = Dot(b, b);
    if (ab * bc – ac * bb < 0.0f) return 0;
    // Otherwise P must be in (or on) the triangle
    return 1;
}
```

这里需留意该函数与 3.4 节中 **Barycentric()**函数之间的类似性。

在 2D 环境下，该测试更加简单。不失一般性，假设三角形 ABC 呈逆时针排列。当且仅当顶点 P 分别位于有向线段 AB、BC、CA 的左侧时，P 位于三角形内部。该测试可以采用第 3 章定义的 2D 叉积计算加以实现。即给定下列函数：

```
// Compute the 2D pseudo cross product Dot(Perp(u), v)
float Cross2D(Vector2D u, Vector2D v)
{
    return u.y * v.x – u.x * v.y;
}
```

顶点 P 相对于线段 AB 的左侧位置判断则转化为对该函数的调用，且输入参数为 $P-A$ 和 $B-A$，并随后查看该叉积结果是否为正值。全部测试过程如下所示：

```
// Test if 2D point P lies inside the counterclockwise 2D triangle ABC
int PointInTriangle(Point2D p, Point2D a, Point2D b, Point2D c)
{
    // If P to the right of AB then outside triangle
    if (Cross2D(p – a, b – a) < 0.0f) return 0;
    // If P to the right of BC then outside triangle
    if (Cross2D(p – b, c – b) < 0.0f) return 0;
```

```
    // If P to the right of CA then outside triangle
    if (Cross2D(p – c, a – c) < 0.0f) return 0;
    // Otherwise P must be in (or on) the triangle
    return 1;
}
```

如果三角形的顺时针或逆时针排列方向未知，仍然可以使用同一算法思想。亦即可以通过计算顶点 *P* 是否位于三角形各边的同一侧，从而实现顶点 *P* 与三角形 *ABC* 的包含测试。因此，存在两种情况：顶点 *P* 位于各边的左侧或位于各边的右侧。否则，顶点 *P* 将位于三角形的外部。在具体实现过程中，代码需要进行适当的调整，并测试 2D 叉积结果是否具有相同的符号值。

```
// Test if 2D point P lies inside 2D triangle ABC
int PointInTriangle2D(Point2D p, Point2D a, Point2D b, Point2D c)
{
    float pab = Cross2D(p – a, b – a);
    float pbc = Cross2D(p – b, c – b);
    // If P left of one of AB and BC and right of the other, not inside triangle
    if (!SameSign(pab, pbc)) return 0;
    float pca = Cross2D(p – c, a – c);
    // If P left of one of AB and CA and right of the other, not inside triangle
    if (!SameSign(pab, pca)) return 0;
    // P left or right of all edges, so must be in (or on) the triangle
    return 1;
}
```

由于 **Cross2D(p – a, b – a)**的返回值为三角形 *PAB* 有符号面积的 2 倍（若 *PAB* 呈逆时针排列，则为正值，否则为负值），因而该测试的另一个版本将计算三角形 *PAB*、*PBC*、*PCA* 的面积，并查看面积和是否大于三角形 *ABC* 的面积：若是，则顶点 *P* 位于三角形 *ABC* 的外部。为了节省计算量，*ABC* 的面积可以事先保存于三角形的相应字段中。在 3D 环境下，该测试可按如下方式加以优化：存储范化后的边面方程且计算测试点的质心坐标。

5.4.3　点与多面体之间的测试

同样，目前存在多种算法可以计算点与多面体之间的包含关系。但具体情况下的具体应用将取决于多面体的定义方式及其凹凸性特征。本节将对几类不同的算法加以概述，某些细节将在第 8 章、第 9 章中进行深入的讨论。

一种简单的测试方案是：定义多面体为凸体且视其为多个半空间的交集。此时，若顶点分别位于每一个半空间内，则其位于多面体内部。

```
// Test if point p inside polyhedron given as the intersection volume of n halfspaces
int TestPointPolyhedron(Point p, Plane *h, int n)
{
    for (int i = 0; i < n; i++) {
        // Exit with 'no containment' if p ever found outside a halfspace
        if (DistPointPlane(p, h[i]) > 0.0f) return 0;
    }
```

```
        // p inside all halfspaces, so p must be inside intersection volume
            return 1;
}
```

若多面体以顶点格式给出且未显示任何彼此连接的信息，则可采用第 9 章描述的 GJK 算法作为一种有效的点包含测试方法。同时，也只有凸多面体可以按照此格式给出。

如果多面体以封闭网格的形式给出，一种算法是针对该网格首先建立一棵实体叶节点 BSP 树（参见第 8 章）。基于该 BSP 树，如果测试顶点结束于一个实体叶节点，则该顶点位于多面体内部（否则位于外部）。该算法可工作于凸多面体以及凹多面体中。

对于网格多面体或半空间交集多面体，一类有效的算法是以测试点为源点、沿任意方向（一般为主轴，如+x 轴）投射一条光线，并计算相交面的数量。如果相交的面数量为奇数，则该顶点位于多面体的内部，否则顶点位于多面体的外部。另外，由于不同浮点数类型间的误差，当光线击中边或顶点（附近）时，需要考虑算法的健壮性问题。为了消除这类问题所产生的退化现象，一种有效的方法是重新投射光线，但需注意不要再次击中边或顶点。同时，该算法也可应用于凸多面体或凹多面体中。

使用预构造的数据结构，例如 Dobkin-Kirkpatrick 层次结构（参见 9.3.1 节），点包含测试的时间消耗能够达到 $O(\log n)$级别。

5.4.4 两个平面间的相交测试

令平面 π_1 和 π_2 的平面方程分别定义为 $\mathbf{n}_1 \cdot X = d_1$，$\mathbf{n}_2 \cdot X = d_2$。当两平面间相互不平行时，必相交于一条直线 L 且有 $L = P + t\mathbf{d}$。由于直线 L 为两个平面的公共边，则必定垂直于各自平面的面法线。因此，直线 L 的方向向量可通过两平面面法线叉积 $\mathbf{d} = \mathbf{n}_1 \times \mathbf{n}_2$ 获得（如图 5.30 所示）。若 $\mathbf{d}$ 为 0 向量，则平面间彼此平行（且分离）或重合，否则两平面相交。

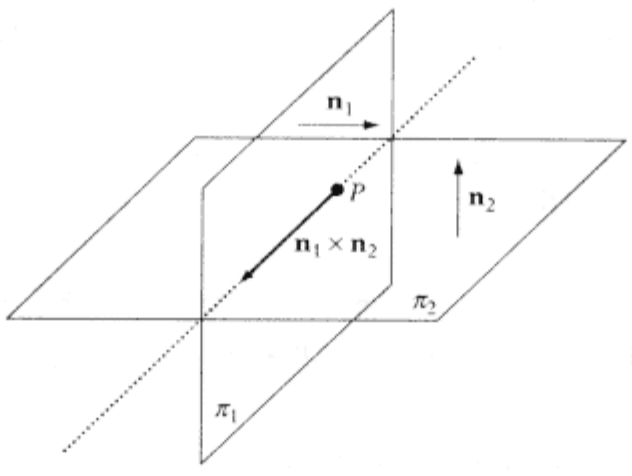


图 5.30　两个平面间的相交状态

若直线 L 计算完毕，则可求得线上一点 P。一种获取线上一点的方法是：在垂直于 $\mathbf{d}$ 的平面上表示该点。由此可知，该平面将跨越 $\mathbf{n}_1$ 和 $\mathbf{n}_2$ 且根据相关 k_1、k_2 值有 $P = k_1\mathbf{n}_1 + k_2\mathbf{n}_2$。进而可知，点 P 定位于平面 π_1 和 π_2 上且满足两平面的平面方程。这里，在平面方程中利用顶点 P 的表达式替换 X 将构造下列 2 × 2 线性方程组：

$$\mathbf{n}_1 \cdot (k_1\mathbf{n}_1 + k_2\mathbf{n}_2) = d_1$$

$$\mathbf{n}_2 \cdot (k_1\mathbf{n}_1 + k_2\mathbf{n}_2) = d_2$$

根据点积的线性特征，上述方程组等价于：

$$k_1(\mathbf{n}_1 \cdot \mathbf{n}_1) + k_2(\mathbf{n}_1 \cdot \mathbf{n}_2) = d_1$$
$$k_1(\mathbf{n}_1 \cdot \mathbf{n}_2) + k_2(\mathbf{n}_2 \cdot \mathbf{n}_2) = d_2$$

并利用下列等式求解 k_1、k_2（如通过克莱姆法则）：

$$k_1 = (d_1(\mathbf{n}_2 \cdot \mathbf{n}_2) - d_2(\mathbf{n}_1 \cdot \mathbf{n}_2))/denom$$
$$k_2 = (d_2(\mathbf{n}_1 \cdot \mathbf{n}_1) - d_1(\mathbf{n}_1 \cdot \mathbf{n}_2))/denom$$

其中，$denom = (\mathbf{n}_1 \cdot \mathbf{n}_1)(\mathbf{n}_2 \cdot \mathbf{n}_2) - (\mathbf{n}_1 \cdot \mathbf{n}_2)^2$。

下列代码给出了上述内容的实现方式：

```
// Given planes p1 and p2, compute line L = p+t*d of their intersection.
// Return 0 if no such line exists
int IntersectPlanes(Plane p1, Plane p2, Point &p, Vector &d)
{
    // Compute direction of intersection line
    d = Cross(p1.n, p2.n);

    // If d is zero, the planes are parallel (and separated)
    // or coincident, so they're not considered intersecting
    if (Dot(d, d) < EPSILON) return 0;

    float d11 = Dot(p1.n, p1.n);
    float d12 = Dot(p1.n, p2.n);
    float d22 = Dot(p2.n, p2.n);

    float denom = d11*d22 - d12*d12;
    float k1 = (p1.d*d22 - p2.d*d12) / denom;
    float k2 = (p2.d*d11 - p1.d*d12) / denom;
    p = k1*p1.n + k2*p2.n;
    return 1;
}
```

由于分母表达式为$(\mathbf{n}_1 \times \mathbf{n}_2) \cdot (\mathbf{n}_1 \times \mathbf{n}_2)$采用拉格朗日恒等式后的计算结果，因而代码可以得到进一步优化。这里，分母等价于：

$$denom = \mathbf{d} \cdot \mathbf{d}$$

利用下列恒等式：

$$\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \cdot \mathbf{w})\mathbf{v} - (\mathbf{v} \cdot \mathbf{w})\mathbf{u}$$

$P = k_1\mathbf{n}_1 + k_2\mathbf{n}_2$ 可进一步简化为：

$$P = k_1\mathbf{n}_1 + k_2\mathbf{n}_2 \Leftrightarrow \text{（初始表达式）}$$
$$P = [(d_1(\mathbf{n}_2 \cdot \mathbf{n}_2) - d_2(\mathbf{n}_1 \cdot \mathbf{n}_2))\mathbf{n}_1 + (d_2(\mathbf{n}_1 \cdot \mathbf{n}_1) - d_1(\mathbf{n}_1 \cdot \mathbf{n}_2))\mathbf{n}_2]/denom \Leftrightarrow \text{（替换 } k_1 \text{ 和 } k_2\text{）}$$
$$Pdenom = (d_1(\mathbf{n}_2 \cdot \mathbf{n}_2) - d_2(\mathbf{n}_1 \cdot \mathbf{n}_2))\mathbf{n}_1 + (d_2(\mathbf{n}_1 \cdot \mathbf{n}_1) - d_1(\mathbf{n}_1 \cdot \mathbf{n}_2))\mathbf{n}_2 \Leftrightarrow \text{（两边乘以 } denom\text{）}$$
$$Pdenom = (d_1(\mathbf{n}_2 \cdot \mathbf{n}_2)\mathbf{n}_1 - d_2(\mathbf{n}_1 \cdot \mathbf{n}_2)\mathbf{n}_1) + (d_2(\mathbf{n}_1 \cdot \mathbf{n}_1)\mathbf{n}_2 - d_1(\mathbf{n}_1 \cdot \mathbf{n}_2)\mathbf{n}_2) \Leftrightarrow \text{（标量乘法分配律）}$$
$$Pdenom = d_1((\mathbf{n}_2 \cdot \mathbf{n}_2)\mathbf{n}_1 - (\mathbf{n}_1 \cdot \mathbf{n}_2)\mathbf{n}_2) + d_2((\mathbf{n}_1 \cdot \mathbf{n}_1)\mathbf{n}_2 - (\mathbf{n}_1 \cdot \mathbf{n}_2)\mathbf{n}_1) \Leftrightarrow \text{（合并同类项）}$$
$$Pdenom = d_1(\mathbf{n}_2 \times (\mathbf{n}_1 \times \mathbf{n}_2)) + d_2(\mathbf{n}_1 \times (\mathbf{n}_2 \times \mathbf{n}_1)) \Leftrightarrow \text{（利用向量恒等变换重写上述等式）}$$

$$Pdenom = d_1(\mathbf{n}_2\times(\mathbf{n}_1\times\mathbf{n}_2)) - d_2(\mathbf{n}_1\times(\mathbf{n}_1\times\mathbf{n}_2)) \Leftrightarrow \text{（改变符号以使叉积格式相同）}$$

$$Pdenom = (d_1\mathbf{n}_2 - d_2\mathbf{n}_1)\times(\mathbf{n}_1\times\mathbf{n}_2) \Leftrightarrow \text{（合并同类项）}$$

$$P = (d_1\mathbf{n}_2 - d_2\mathbf{n}_1)\times\mathbf{d}/denom \text{（利用 } \mathbf{d} \text{ 替换 } \mathbf{n}_1\times\mathbf{n}_2\text{；两边同除以 } denom\text{）}$$

$L = P + t\mathbf{d}$ 中各项的最终结果为：

$$P = (d_1\mathbf{n}_2 - d_2\mathbf{n}_1)\times\mathbf{d}/(\mathbf{d}\cdot\mathbf{d})$$

$$\mathbf{d} = \mathbf{n}_1\times\mathbf{n}_2$$

相关实现代码为：

```
// Given planes p1 and p2, compute line L = p+t*d of their intersection.
// Return 0 if no such line exists
int IntersectPlanes(Plane p1, Plane p2, Point &p, Vector &d)
{
    // Compute direction of intersection line
    d = Cross(p1.n, p2.n);

    // If d is (near) zero, the planes are parallel (and separated)
    // or coincident, so they're not considered intersecting
    float denom = Dot(d, d);
    if (denom < EPSILON) return 0;

    // Compute point on intersection line
    p = Cross(p1.d*p2.n - p2.d*p1.n, d) / denom;
    return 1;
}
```

需注意的是，若已知平面法线为单位向量（即$\|\mathbf{n}_1\| = \|\mathbf{n}_2\| = 1$），则无须再执行分母项的除法操作。

5.4.5 3 个平面间的相交测试

给定平面 π_1、π_2 和 π_3，存在 5 种不同的相交可能性，如图 5.31 所示。

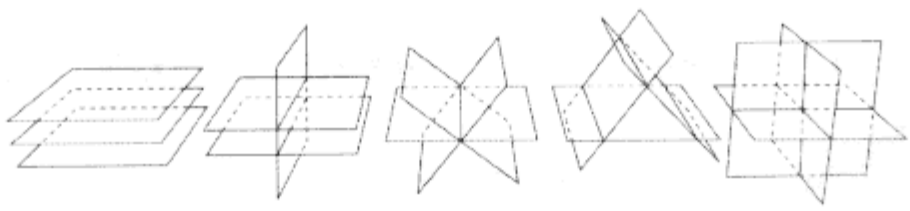


图 5.31　3 个平面形成的 5 种相交状态

1. 3 个平面间彼此平行（也可能共面）。
2. 一个平面贯穿两个平面平行，从而形成两条相互平行的交线。
3. 3 个平面相交于一条直线。
4. 3 个平面之间彼此两两相交于一条直线。
5. 3 个平面相交于一点。

下面定义 3 个平面。π_1:$\mathbf{n}_1\cdot X = d_1$，$\pi_2$:$\mathbf{n}_2\cdot X = d_2$，$\pi_3$:$\mathbf{n}_3\cdot X = d_3$。对于上述相交状态“5”，则有 $\mathbf{n}_1\cdot(\mathbf{n}_2\times\mathbf{n}_3) = 0$，其他状态也可以类似方法加以定义。其中，当平面相交于一点 $X = (x_1, x_2, x_3)$时，可将

平面方程视为一个 3 × 3 线性方程组，并求解该交点：

$$\mathbf{n}_1 \cdot X = d_1$$
$$\mathbf{n}_2 \cdot X = d_2$$
$$\mathbf{n}_3 \cdot X = d_3$$

可利用高斯消元法或根据行列式和克莱姆法则求解该方程组。令 $\mathbf{m}_1 = [\mathbf{n}_1^x \quad \mathbf{n}_2^x \quad \mathbf{n}_3^x]^T$，$\mathbf{m}_2 = [\mathbf{n}_1^y \quad \mathbf{n}_2^y \quad \mathbf{n}_3^y]^T$，$\mathbf{m}_3 = [\mathbf{n}_1^z \quad \mathbf{n}_2^z \quad \mathbf{n}_3^z]^T$，根据克莱姆法则，解为：

$$x_1 = |\mathbf{d} \quad \mathbf{m}_2 \quad \mathbf{m}_3| / |\mathbf{m}_1 \quad \mathbf{m}_2 \quad \mathbf{m}_3|$$
$$x_2 = |\mathbf{m}_1 \quad \mathbf{d} \quad \mathbf{m}_3| / |\mathbf{m}_1 \quad \mathbf{m}_2 \quad \mathbf{m}_3|$$
$$x_3 = |\mathbf{m}_1 \quad \mathbf{m}_2 \quad \mathbf{d}| / |\mathbf{m}_1 \quad \mathbf{m}_2 \quad \mathbf{m}_3|$$

其中，$\mathbf{d} = [d_1 \quad d_2 \quad d_3]^T$。

选择最为适宜的计算表达式——以尽可能复用某些计算式——标量三重积可简化为：

$$x_1 = \mathbf{d} \cdot \mathbf{u}/denom$$
$$x_2 = \mathbf{m}_3 \cdot \mathbf{v}/denom$$
$$x_3 = -\mathbf{m}_2 \cdot \mathbf{v}/denom$$

其中，$\mathbf{u} = \mathbf{m}_2 \times \mathbf{m}_3$，$\mathbf{v} = \mathbf{m}_1 \times \mathbf{d}$，$denom = \mathbf{m}_1 \cdot \mathbf{u}$。

下列代码实现了 3 平面相交算法：

```
// Compute the point p at which the three planes p1, p2 and p3 intersect (if at all)
int IntersectPlanes(Plane p1, Plane p2, Plane p3, Point &p)
{
    Vector m1 = Vector(p1.n.x, p2.n.x, p3.n.x);
    Vector m2 = Vector(p1.n.y, p2.n.y, p3.n.y);
    Vector m3 = Vector(p1.n.z, p2.n.z, p3.n.z);

    Vector u = Cross(m2, m3);
    float denom = Dot(m1, u);
    if (Abs(denom) < EPSILON) return 0;        // Planes do not intersect in a point
    Vector d(p1.d, p2.d, p3.d);
    Vector v = Cross(m1, d);
    float ood = 1.0f / denom;
    p.x = Dot(d, u) * ood;
    p.y = Dot(m3, v) * ood;
    p.z = -Dot(m2, v) * ood;
    return 1;
}
```

[Goldman90]中指出，也可利用下列公式求解 X：

$$X = \frac{d_1(\mathbf{n}_2 \times \mathbf{n}_3) + d_2(\mathbf{n}_3 \times \mathbf{n}_1) + d_3(\mathbf{n}_1 \times \mathbf{n}_2)}{\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)}$$

此处，该公式可认为是：针对某值 a、b、c，将 X 视为平面交线方向的线性组合：

$$X = a(\mathbf{n}_2 \times \mathbf{n}_3) + b(\mathbf{n}_3 \times \mathbf{n}_1) + c(\mathbf{n}_1 \times \mathbf{n}_2)$$

将交点代入相应的平面方程，有：

$$\mathbf{n}_1 \cdot (a(\mathbf{n}_2 \times \mathbf{n}_3) + b(\mathbf{n}_3 \times \mathbf{n}_1) + c(\mathbf{n}_1 \times \mathbf{n}_2)) = d_1$$

$$\mathbf{n}_2 \cdot (a(\mathbf{n}_2 \times \mathbf{n}_3) + b(\mathbf{n}_3 \times \mathbf{n}_1) + c(\mathbf{n}_1 \times \mathbf{n}_2)) = d_2$$
$$\mathbf{n}_3 \cdot (a(\mathbf{n}_2 \times \mathbf{n}_3) + b(\mathbf{n}_3 \times \mathbf{n}_1) + c(\mathbf{n}_1 \times \mathbf{n}_2)) = d_3$$

并可以简化为：

$$\mathbf{n}_1 \cdot a(\mathbf{n}_2 \times \mathbf{n}_3) = d_1$$
$$\mathbf{n}_2 \cdot b(\mathbf{n}_3 \times \mathbf{n}_1) = d_2$$
$$\mathbf{n}_3 \cdot c(\mathbf{n}_1 \times \mathbf{n}_2) = d_3$$

其中，不难获取 a、b、c 的各项值。将 a、b、c 的各项值代入即可得到最初公式。同时，稍加处理可使得公式得到进一步简化：

$$X = \frac{d_1(\mathbf{n}_2 \times \mathbf{n}_3) + \mathbf{n}_1 \times (d_3\mathbf{n}_2 - d_2\mathbf{n}_3)}{\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)}$$

其代码实现如下所示：

```
// Compute the point p at which the three planes p1, p2 and p3 intersect (if at all)
int IntersectPlanes(Plane p1, Plane p2, Plane p3, Point &p)
{
    Vector u = Cross(p2.n, p3.n);
    float denom = Dot(p1.n, u);
    if (Abs(denom) < EPSILON) return 0;          // Planes do not intersect in a point
    p = (p1.d * u + Cross(p1.n, p3.d * p2.n – p2.d * p3.n)) / denom;
    return 1;
}
```

这里，两种算法都可以对 3 × 3 线性方程组进行求解，其范围并不仅限于计算 3 个非共面交点问题。

5.5 动态相交测试

目前所涉及的相关测试算法只是针对静态物体对象：即某一时刻、某一位置点的相交测试。但是，静态测试将会导致一种“隧道效应（tunneling）”：物体在某一时刻从一个位置位移至下一个位置时，可能会错过碰撞检测对象。图 5.32（a）说明了这一现象，其中球体与矩形障碍物之间即产生了“隧道效应”。

一类有效的解决方法是：令物体按照运动路径连续移动，并以时间为参数定义该位移，最后在碰撞时刻——两物体间距离为 0 时——计算该点。图 5.32（b）显示了这种连续扫掠测试方案。然而，虽然计算这类简单物体、简单位移的碰撞时刻是切实可行的，但在处理运动方式不确定、具有复杂形状的物体时，该算法实现起来并非易事。准确地讲，由于成本高昂，该算法无法在实时应用中加以实现。

然而，仍然存在一种折中方案，即对物体对象的运动路径实施采样（sample），并在其位移过程中执行多次静态物体测试。图 5.32（c）显示了对球体的运动路径进行多步细分的方式，并在每一个采样点上采取物体间的静态测试。

通过在采样点间获取运动物体自身的相交状态，可以部分地消除隧道效应。然而，如图 5.32（c）中的黑色三角形所示，仍然存在某些空间使得细小、狭长的物体产生隧道效应。为了全面地考查这一问题，物体对象需要在尺寸上进行相应的扩展，使得全部采样能够充分覆盖连续扫掠对象形成的体空间。

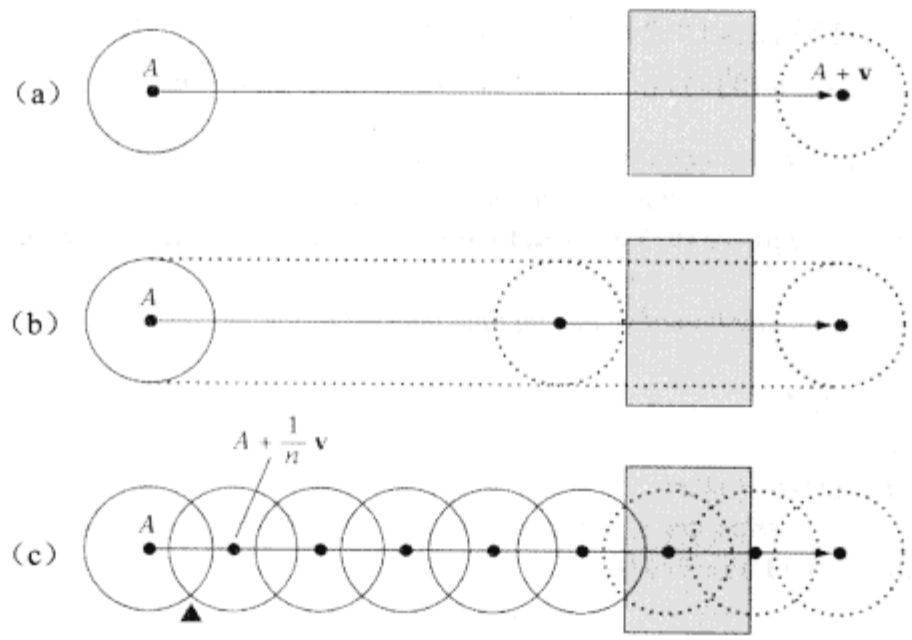


图 5.32　动态碰撞测试。（a）仅在对象位移的起始点、结束点进行测试将产生隧道效应。（b）采用扫略测试计算最终碰撞点，但该算法应用范围有限。（c）路径采样需要较大的计算量且仍会产生隧道效应，相关区域采用黑色三角形标识

在最坏情况下，即使未发生碰撞，也需要在运动路径上测试全部 n 个采样点，即时间复杂度为 $O(n)$。需要注意的是，细分以及采样计算只是针对物体的运动路径，而非运动时间。另外，细分物体对象的运动路径并记录其相交状态也是一类重要的问题。

在许多例子中，物体对象可假定为分段线性运动，即仅在两点间实现位移，并在初始点或终点即刻进行旋转操作。因此，对于两个物体 A 和 B，可从 A 的运动中“减去”物体 B 的运动，则物体 B 相对静止，且相交测试可视作运动的物体对象（A）与静止的物体（B）之间的计算，即两物体之间的彼此相对运动。最后，如果交点已计算完毕，需要将 B 的运动状态最终作用于交点之上，从而获取正确的世界坐标而非物体的相对坐标。下面将首先探讨两个动态测试的通用算法，而后通过多个特定算法来阐述常见相交测试过程中出现的问题。

5.5.1　运动物体的区间半分法相交测试

在采样算法和连续扫描测试之间的某一位置，可采用基于递归的二分搜索算法计算碰撞时刻（若存在）。再次考查下列问题：球体 S 高速掠过一个静止物体，S 的初始位置为 A 且运动方向以向量 $\mathbf{v}$ 表示。首先，构造一个包围球且完全包含位置 A 及位置 $A+\mathbf{v}$ 的球体，并利用该包围球与其他物体执行碰撞测试。若不存在碰撞，则 S 直接到达终点。然而，若其中检测到碰撞，则二分计算位移量并以同一方式递归地测试前半部分。如果其中未检测到碰撞，则对后半部分执行相同的递归测试。若受测位移量在趋于某个最小预定值时碰撞检测过程仍未结束，则可假定该碰撞存在且递归结束。

图 5.33（a）显示了区间二分法的测试过程，其中，根据总位移量构造了一个球体，并在某一点处检测到与另一物体发生碰撞。这里需要计算位移的中点，而后分别对各半部分执行递归调用。在首次递归调用过程中，由起始点至中点形成的球体 B_1 中并未发现任何碰撞，且该递归调用退出并返回“未碰撞”信息；该过程继续执行并开始第二次递归调用，同时，检测到包围后半程的球体 B_2 与物体发生碰撞。因此，在当前二分区间内生成新的中点并针对该二分区间再次执行递归计算，如图 5.33（b）所

示。同理，球体 B_3 中发生碰撞且执行递归调用。在该例子中，由于产生了碰撞相交，因而递归调用最终退出。关于运动-静止球体之间的区间二分算法，其代码实现如下所示：

```
// Intersect sphere s0 moving in direction d over time interval t0 <= t <= t1, against
// a stationary sphere s1. If found intersecting, return time t of collision
int TestMovingSphereSphere(Sphere s0, Vector d, float t0, float t1, Sphere s1, float &t)
{
    // Compute sphere bounding motion of s0 during time interval from t0 to t1
    Sphere b;
    float mid = (t0 + t1) * 0.5f;
    b.c = s0.c + d * mid;
    b.r = (mid – t0) * Length(d) + s0.r;
    // If bounding sphere not overlapping s1, then no collision in this interval
    if (!TestSphereSphere(b, s1)) return 0;

    // Cannot rule collision out: recurse for more accurate testing. To terminate the
    // recursion, collision is assumed when time interval becomes sufficiently small
    if (t1 - t0 < INTERVAL_EPSILON) {
        t = t0;
        return 1;
    }

    // Recursively test first half of interval; return collision if detected
    if (TestMovingSphereSphere(s0, d, t0, mid, s1, t)) return 1;

    // Recursively test second half of interval
    return TestMovingSphereSphere(s0, d, mid, t1, s1, t);
}
```

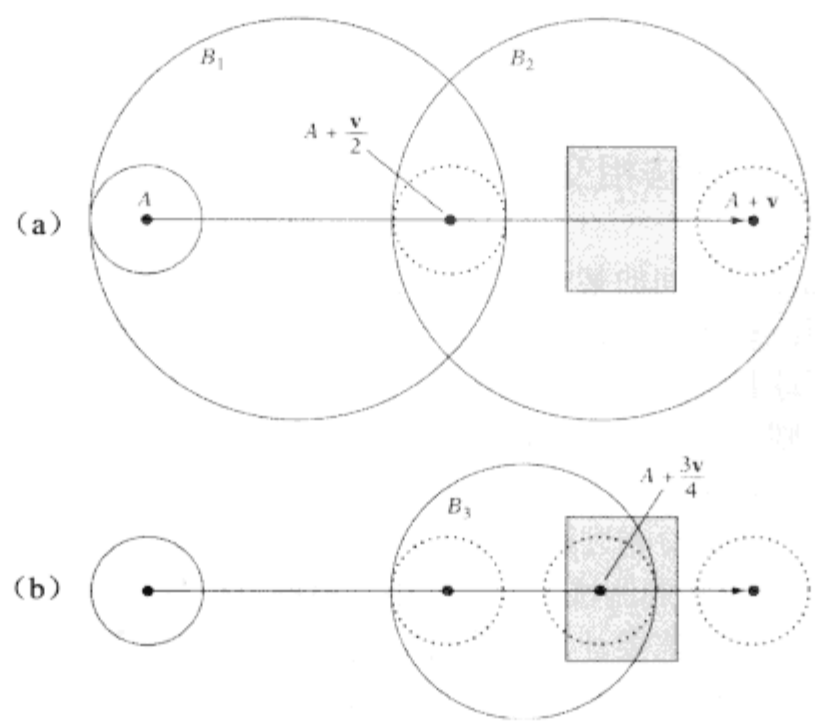


图 5.33　采用区间二分法计算运动球体与静止球体碰撞测试的步骤

通常情况下，该算法的时间复杂度为 $O(\log n)$。然而，在某些情况下也会出现异常。对于该算法，其最坏情况是球体以稍大于结束递归调用时的预设距离平行于表面运动。此时，算法的每一步都将对两个子区间执行递归调用，从而导致球体位移被细分为多个小步，且在每一步中都将执行碰撞检测。对此，较为紧凑的包围体类型（如 OBB）将会有较好的表现。但需要注意的是，OBB 的计算代价相对高昂。在某些特定的结构场景中，AABB 将是一类较好的选择，并可以与地面、墙壁实现较好的对齐——彼此接近于垂直对齐并且与世界坐标系统实现对齐。

区间二分算法一般适用于处理以任意方向运动的复杂物体。此处，可以定义一个函数 **MaximumObjectMovementOverTime()**，并以物体对象和时间间隔作为参数。函数将返回对应时间间隔内、位移后的物体表面顶点间的最大路径长度。另外，定义函数 **MinimumObjectDistanceAtTime()**，并以两个物体以及时间值作为参数。函数将返回在给定时间值内、物体表面间的最小距离。当定义了上述两个函数之后，区间二分算法的代码实现如下所示：

```
// Test collision between objects a and b moving over the time interval
// [startTime, endTime]. When colliding, time of collision is returned in hitTime
int IntervalCollision(Object a, Object b, float startTime, float endTime, float &hitTime)
{
    // Compute the maximum distance objects a and b move over the time interval
    float maxMoveA = MaximumObjectMovementOverTime(a, startTime, endTime);
    float maxMoveB = MaximumObjectMovementOverTime(b, startTime, endTime);
    float maxMoveDistSum = maxMoveA + maxMoveB;
    // Exit if distance between a and b at start larger than sum of max movements
    float minDistStart = MinimumObjectDistanceAtTime(a, b, startTime);
    if (minDistStart > maxMoveDistSum) return 0;
    // Exit if distance between a and b at end larger than sum of max movements
    float minDistEnd = MinimumObjectDistanceAtTime(a, b, endTime);
    if (minDistEnd > maxMoveDistSum) return 0;

    // Cannot rule collision out: recurse for more accurate testing. To terminate the
    // recursion, collision is assumed when time interval becomes sufficiently small
    if (endTime – startTime < INTERVAL_EPSILON) {
        hitTime = startTime;
        return 1;
    }
    // Recursively test first half of interval; return collision if detected
    float midTime = (startTime + endTime) * 0.5f;
    if (IntervalCollision(a, b, startTime, midTime, hitTime)) return 1;
    // Recursively test second half of interval
    return IntervalCollision(a, b, midTime, endTime, hitTime);
}
```

对于凸多边形物体对象，当获取最近碰撞时间后，随后的分离轴测试将会提供相应的碰撞法线，并作为碰撞返回信息（即对应的分离轴）。同时，分离轴测试也可直接用于运动凸体对象的碰撞计算，参见 5.5.2 节。

5.5.2 运动凸体对象的分离轴测试

对于以恒速运动的非旋转凸多边形对象或凸体对象，对分离轴测试算法（参见 5.2.1 节）稍加修正即可用于计算物体间的相交时间（若存在）。首先需要注意的是，对于运动的物体 A 和 B，可将 B 的速度减去 A 的速度，从而使问题有效地转化为运动物体 B 与静止物体 A 之间的相交测试。

如果两物体在初始状态下即相交，则碰撞时间为 $t = 0$，否则两物体间为分离状态。因此，考查 A 和 B 在任意直线 L 上的投影区间。很明显，仅当 B 的投影区间移向 A 的投影区间且产生交迭时，才存在碰撞。

需要注意的是，对于 A 和 B 投影区间接触的初始时刻之前，二者必定在某一轴向上处于分离状态；对于 A 和 B 投影区间接触的结束时刻之后，二者必定在某一轴向上再次处于分离状态。因此，可利用静态分离轴测试中同一分离轴上的投影，来计算碰撞的初始时刻和结束时刻。

结果为：针对每一个潜在的分离轴，可通过计算投影区间相交时的初始和结束时刻，从而确定 B 与 A 的相交状态。这里，可以采用简单的线性操作计算初始和结束时刻，并分别对应于物体间的初始、最终碰撞状态。如果物体间在各分离轴上同时相交，则它们必定相交。基于此，下列计算结果将最终确定物体间的碰撞结果：碰撞初始时刻和结束时刻所确定的时间区间的交集非空。这里，令 t_{first} 记录分离轴上全部初始碰撞时刻的最大值，t_{last} 记录分离轴上全部碰撞结束时刻的最小值。待全部轴测试完毕后，当且仅当 $t_{first} \leqslant t_{last}$，物体间相交。如果测试过程中的某一时刻出现了 $t_{first} > t_{last}$，则物体间不会出现相交状态。此时，可即刻返回“不相交”这一信息。

[Gomez99]中针对 AABB 的运动物体分离轴测试技术给出了相关的描述；针对一般情形，其具体内容可参见[Levine00]。关于运动 AABB 的相交测试，其算法实例将在 5.5.8 节中加以演示。

5.5.3 运动球体与平面间的相交测试

令平面 π 的平面方程定义为 $\mathbf{n} \cdot X = d$，其中 $\mathbf{n}$ 为单位向量。同时，令球心为 C 且半径为 r，其位移方向向量以 $\mathbf{v}$ 表示，因此球心位移可以 $C(t) = C + t\,\mathbf{v}$ 定义，其中 $0 \leqslant t \leqslant 1$。另外，一点 R 距平面的有符号距离为$(\mathbf{n} \cdot R) - d$，因此，若$|(\mathbf{n} \cdot C) - d| \leqslant r$，则球体与平面相交。

球体相对于平面的运动方式可按下列状态加以分类：考查 $\mathbf{n} \cdot \mathbf{v}$ 的符号以及球体相对于平面的位置。若$(\mathbf{n} \cdot \mathbf{v}) > 0$，假设球体位于平面的正面，则球体背离平面运动；当球体位于平面的背面，则球体朝向平面运动。对于$(\mathbf{n} \cdot \mathbf{v}) < 0$，情况则恰好相反。这里，$(\mathbf{n} \cdot C) - d$ 的符号表明了球体与平面的位置关系，即位于哪一个面空间中，具体情况为：当$(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot C - d) > 0$ 时，球体背向平面运动；当$(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot C - d) < 0$ 时，球体面向平面运动（即考查两乘式的符号统一性）；当$(\mathbf{n} \cdot \mathbf{v}) = 0$ 时，球体以平行于平面的方式运动。

此处，假定球体朝向平面运动。在计算二者碰撞的时间值 t 时，一种较为直接的方法是将平面向球心方向移动 r 距离，则问题转化为计算线段 $S(t) = C + t\,\mathbf{v}$（$0 \leqslant t \leqslant 1$），与置换平面之间的相交状态（如图 5.34 所示）。求解 t 值，若 t 位于线段区间内，则该值为球体与平面之间的碰撞时间值。

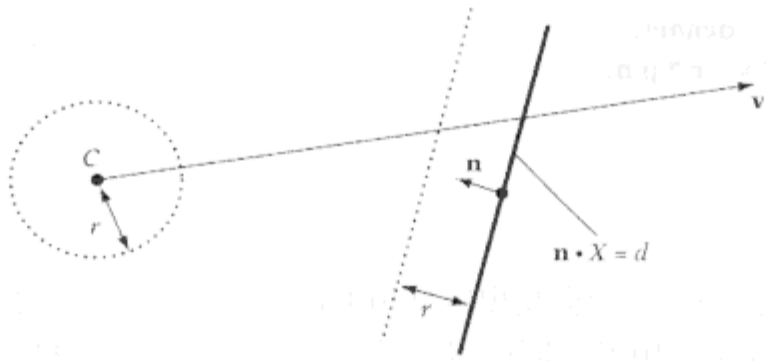


图 5.34　球心位置为 C、半径为 r、位移向量为 $\mathbf{v}$ 的运动球体与平面 $\mathbf{n} \cdot X = d$ 之间的相交测试等价于：线段 $S(t) = C + t\mathbf{v}$ 与沿 $\mathbf{n}$ 向偏移 r 的平面间的测试（这里，r 为正值且 C 位于平面的正面）

再次强调计算过程中需要检查$(\mathbf{n} \cdot C) - d$ 的符号值，从而确定平面的置换方式。若该项为正值，置换平面的方程定义为$(\mathbf{n} \cdot Q) = d + r$，则二者相交且在时刻 t 时，球体上与平面的碰撞点可表示为 $Q = C(t) - r\mathbf{n}$；若该项为负值，置换平面的方程定义为$(\mathbf{n} \cdot Q) = d - r$，且交点为 $Q = C(t) + r\mathbf{n}$。

综上所述，t 的计算过程如下所示：

$$(\mathbf{n} \cdot X) = d \pm r \Leftrightarrow \text{（位移后的平面方程）}$$

$$\mathbf{n} \cdot (C + t\mathbf{v}) = d \pm r \Leftrightarrow \text{（利用 } S(t) = C + t\mathbf{v} \text{ 替换 } X\text{）}$$

$$(\mathbf{n} \cdot C) + t(\mathbf{n} \cdot \mathbf{v}) = d \pm r \Leftrightarrow \text{（点积展开）}$$

$$t = (\pm r - ((\mathbf{n} \cdot C) - d))/(\mathbf{n} \cdot \mathbf{v}) \text{（求解 } t\text{）}$$

当 $0 \leqslant t \leqslant 1$ 时，球体在 t 时刻与平面相交，且交点为 Q。

需要注意的是，在确定球体与平面之间的运动方式时，需要计算$(\mathbf{n} \cdot C) - d$ 和 $\mathbf{n} \cdot \mathbf{v}$ 项，且该项在求解 t 值的相关表达式中曾多次出现，但在具体实现过程中则无须对其重复计算。上述相交测试的实现代码如下所示：

```
// Intersect sphere s with movement vector v with plane p. If intersecting
// return time t of collision and point q at which sphere hits plane
int IntersectMovingSpherePlane(Sphere s, Vector v, Plane p, float &t, Point &q)
{
    // Compute distance of sphere center to plane
    float dist = Dot(p.n, s.c) - p.d;
    if (Abs(dist) <= s.r) {
        // The sphere is already overlapping the plane. Set time of
        // intersection to zero and q to sphere center
        t = 0.0f;
        q = s.c;
        return 1;
    } else {
        float denom = Dot(p.n, v);
        if (denom * dist >= 0.0f) {
            // No intersection as sphere moving parallel to or away from plane
            return 0;
        } else {
            // Sphere is moving towards the plane

            // Use +r in computations if sphere in front of plane, else -r
            float r = dist > 0.0f ? s.r : -s.r;
```

```
            t = (r - dist) / denom;
            q = s.c + t * v – r * p.n;
            return 1;
        }
    }
}
```

这里，相交后的函数返回值 t 可能会超出区间[0,1]的范围。此时，t 对应着某一未来时间段，其后球体将会与平面相交。对于 t 值与[0,1]区间的包含关系，可留与调用函数加以判断。

若仅测试运动球体与平面之间是否相交，即并不计算碰撞后的具体位置，则上述计算过程可以得到极大的简化。如果球体的起点和终点分别位于平面的两侧，或球体的起点位置、终点位置已与平面之间直接处于交叠状态，则二者相交。测试代码如下所示：

```
// Test if sphere with radius r moving from a to b intersects with plane p
int TestMovingSpherePlane(Point a, Point b, float r, Plane p)
{
    // Get the distance for both a and b from plane p
    float adist = Dot(a, p.n) - p.d;
    float bdist = Dot(b, p.n) - p.d;
    // Intersects if on different sides of plane (distances have different signs)
    if (adist * bdist < 0.0f) return 1;
    // Intersects if start or end position within radius from plane
    if (Abs(adist) <= r || Abs(bdist) <= r) return 1;
    // No intersection
    return 0;
}
```

上述测试代码仍假设平面法线为单位向量，即$||\mathbf{n}|| = 1$。

5.5.4 运动 AABB 与平面间的相交测试

令平面 π 的方程定义为$(\mathbf{n} \cdot \mathrm{X}) = d$，其中 $\mathbf{n}$ 为单位向量。令轴对齐包围盒 B 的中心点为 C，局部坐标轴为 $\mathbf{u}_0 = (1, 0, 0)$、$\mathbf{u}_1 = (0, 1, 0)$、$\mathbf{u}_2 = (0, 0, 1)$，尺寸为 e_0、e_1、e_2。这里，B 的方向向量利用 $\mathbf{v}$ 加以表示，即针对盒体中心的位移，有 $C(t) = C + t\,\mathbf{v}$，其中 $0 \leqslant t \leqslant 1$。

可以将平面法线视为分离轴，则 B 在 $\mathbf{n}$ 轴上的投影半径为：

$$r = e_0|\mathbf{u}_0 \cdot \mathbf{n}| + e_1|\mathbf{u}_1 \cdot \mathbf{n}| + e_2|\mathbf{u}_2 \cdot \mathbf{n}|$$

由于 $\mathbf{u}_0$、$\mathbf{u}_1$、$\mathbf{u}_2$ 皆为恒定值，所以上式可简化为：

$$r = e_0|\mathbf{n}_x| + e_1|\mathbf{n}_y| + e_2|\mathbf{n}_z|$$

需要注意的是，投影半径并不随 B 的移动而改变。当前，测试过程等价于运动球体与平面之间的测试。同样，点 R 距离平面的有符号距离为$(\mathbf{n} \cdot R) - d$。因此，若$|(\mathbf{n} \cdot C) - d| \leqslant r$，则 AABB 在初始状态下即与平面相交。另外，若$(\mathbf{n} \cdot \mathbf{v}) > 0$，则 AABB 背离平面运动；若$(\mathbf{n} \cdot \mathbf{v}) = 0$，则 AABB 平行于平面运动。

如前所述，将平面朝向 AABB 的方向移动 r 后，该平面的方程变为$(\mathbf{n} \cdot X) = d + r$，且 AABB 上的碰撞点记为 $Q = C(t) - r\mathbf{n}$，其中 $C(t)$记录了与平面碰撞时的当前 AABB 位置。此处，交点 Q 位于平面

上的判断条件为$(\mathbf{n} \cdot Q) = d$。下面，求解 t 且有 $t = (r + d - (\mathbf{n} \cdot C))/(\mathbf{n} \cdot \mathbf{v})$。如果 $0 \leqslant t \leqslant 1$，则 AABB 在时刻 t 与平面发生碰撞。与运动球体与平面间的相交测试不同，AABB 测试中的最终结果不一定是唯一的碰撞点，因为碰撞区域可能会是 AABB 的一条边或一个面。最后，根据上述 AABB 的定义方式，除了 r 的处理方法不同以外，OBB 的测试方案基本相同，因为此时将涉及 OBB 的方向轴而不再是世界坐标轴。

5.5.5　运动球体与球体之间的相交测试

设定球体 S_0、S_1 的半径分别为 r_0 和 r_1，其位移由两个方向向量 $\mathbf{v}_0$、$\mathbf{v}_1$ 加以定义。因此，球心的参数化位移方程可表示为 $P_0(t) = C_0 + t\,\mathbf{v}_0$，$P_1(t) = C_1 + t\,\mathbf{v}_1$，其中对于时间值区间有 $0 \leqslant t \leqslant 1$。在 t 时刻两球心间的向量 $\mathbf{d}$ 可表示为：

$$\mathbf{d}(t) = (C_0 + t\mathbf{v}_0) - (C_1 + t\,\mathbf{v}_1) = (C_0 - C_1) + t(\mathbf{v}_0 - \mathbf{v}_1)$$

这里，假定球体间在初始时刻不相交，且当向量 $\mathbf{d}$ 的长度等于球体半径之和时，二者相交：$(\mathbf{d}(t) \cdot \mathbf{d}(t))^{1/2} = r_0 + r_1$。可对等式两边进行平方以避免开方计算，即 $\mathbf{d}(t) \cdot \mathbf{d}(t) = (r_0 + r_1)^2$。求解 t 时，可令 $\mathbf{s} = C_0 - C_1$，$\mathbf{v} = \mathbf{v}_0 - \mathbf{v}_1$，$r = r_0 + r_1$，则有：

$$\mathbf{d}(t) \cdot \mathbf{d}(t) = (r_0 + r_1)^2 \Leftrightarrow \text{（初始表达式）}$$

$$(\mathbf{s} + t\,\mathbf{v}) \cdot (\mathbf{s} + t\,\mathbf{v}) = r^2 \Leftrightarrow \text{（替换 } \mathbf{d}(t) = \mathbf{s} + t\,\mathbf{v}\text{）}$$

$$(\mathbf{s} \cdot \mathbf{s}) + 2(\mathbf{v} \cdot \mathbf{s})t + (\mathbf{v} \cdot \mathbf{v})t^2 = r^2 \Leftrightarrow \text{（点积展开）}$$

$$(\mathbf{v} \cdot \mathbf{v})t^2 + 2(\mathbf{v} \cdot \mathbf{s})t + (\mathbf{s} \cdot \mathbf{s} - r^2) = 0 \quad \text{（获取二次方程的标准格式）}$$

如上所示，该方程的最终形式为关于 t 的二次方程，可记为 $at^2 + 2bt + c = 0$，其中 $a = \mathbf{v} \cdot \mathbf{v}$，$b = \mathbf{v} \cdot \mathbf{s}$，$c = \mathbf{s} \cdot \mathbf{s} - r^2$，且求解 t 为：

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a}$$

此处，利用判别式 $d = b^2 - ac$ 可以确定方程实根的数量。若 $d < 0$，则方程无实根，意味着球体间不相交；若 $d = 0$，则方程有一个实根，即两球体在 t 时刻相切；若 $d > 0$，方程包含两个实根，即较小的 t 值对应首次碰撞时刻，而后相互贯穿并在较大 t 值时刻停止穿越。首次碰撞时刻如下所示：

$$t = \frac{-b - \sqrt{b^2 - ac}}{a}$$

在测试的具体实现过程中，需要确认在运动初始位置处两球体之间是否相交。否则，将无法正确处理“较大球体完全包含较小球体”这一状况，因为上述球体间首次碰撞时的相应判断条件无法满足这一特例。因此，一种测试实现方案为：

```
int TestMovingSphereSphere(Sphere s0, Sphere s1, Vector v0, Vector v1, float &t)
{
    Vector s = s1.c - s0.c;          // Vector between sphere centers
    Vector v = v1 - v0;              // Relative motion of s1 with respect to stationary s0
    float r = s1.r + s0.r;           // Sum of sphere radii
    float c = Dot(s, s) – r * r;
    if (c < 0.0f) {
        // Spheres initially overlapping so exit directly
```

```
        t = 0.0f;
        return 1;
    }
    float a = Dot(v, v);
    if (a < EPSILON) return 0;        // Spheres not moving relative each other
    float b = Dot(v, s);
    if (b >= 0.0f) return 0;          // Spheres not moving towards each other
    float d = b * b – a * c;
    if (d < 0.0f) return 0;           // No real-valued root, spheres do not intersect

    t = (-b – Sqrt(d)) / a;
    return 1;
}
```

[Gomez99]中也给出了该特定算法的类似描述。

针对该问题，另外一种算法则是利用前述相关计算结果加以实现。首先，当前问题（如图 5.35（a）所示）可视为运动球体与静止球体之间的碰撞测试，即两球体皆减去 $\mathbf{v}_1$（如图 5.35（b）所示）。由于 $\mathbf{d}(t) = r_0 + r_1$ 时二者首次碰撞，因此，两球体间分别同时缩放同一半径值，将不会影响最终的 t 值计算结果。运动球体将缩为一点，且另一静止球体的半径将合成为 r_0+r_1（如图 5.35（c）所示）。则当前问题转变为光线与静止球体之间的相交计算，其测试函数可参见 5.3.2 节。

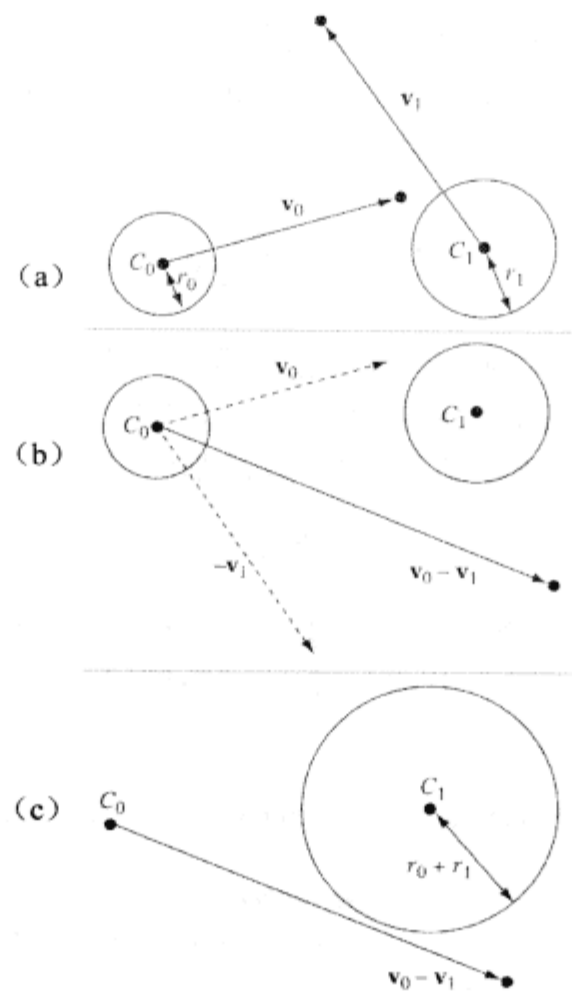


图 5.35　将运动球体-球体间的测试改写为光线相交测试。（a）运动球体-球体之间的相交测试。（b）将问题转换为运动球体与静止球体之间的测试。（c）光线与具有较大半径的球体之间的测试

此处，该问题的实现代码可直接表示为：

```
int TestMovingSphereSphere(Sphere s0, Sphere s1, Vector v0, Vector v1, float &t)
{
    // Expand sphere s1 by the radius of s0
    s1.r += s0.r;
    // Subtract movement of s1 from both s0 and s1, making s1 stationary
    Vector v = v0 – v1;
    // Can now test directed segment s = s0.c + tv, v = (v0-v1)/||v0-v1|| against
    // the expanded sphere for intersection
    Point q;
    float vlen = Length(v);
    if (IntersectRaySphere(s0.c, v / vlen, s1, t, q)) {
        return t <= vlen;
    }
    return 0;
}
```

5.5.6　运动球体与三角形以及多边形之间的相交测试

将球体 S 定义为球心 C 和半径 r，$\mathbf{v}$ 为球体 S 的方向向量，则球心的位移方程为 $C(t) = C + t\,\mathbf{v}$，其中 $0 \leqslant t \leqslant 1$。定义三角形 T 且其单位法线向量为 $\mathbf{n}$。

[Nettle00]提出了球体与三角形相交测试的概念及其测试方法。为了不失一般性，这里假定球体位于三角形 T 的正面，并在其背面结束运动。因此，球体上与平面的首次碰撞点可记为 $D = C - r\,\mathbf{n}$。当前，需要计算相对于平面的有向线段 $D + t\,\mathbf{v}$，$0 \leqslant t \leqslant 1$；且令 P 为交点。如果 P 位于三角形内部，则 t 即为相交时的相应时间值，P 为有效交点（如图 5.36（a）所示）。

针对点 P 位于三角形外部这一情况，令 Q 为三角形上距离 P 的最近点。若运动球体确实可击中三角形，则 Q 为首先击中的一点。

为了计算球体是否击中点 Q，这里可以从点 Q 出发、面向球体反向投射一条光线 $Q - t\,\mathbf{v}$。若该光线未击中球体，或相关 t 值未处于[0,1]区间范围内，则球体未击中点 Q 且不存在相交。若光线在点 R 处击中球体且 $0 \leqslant t \leqslant 1$，则球体上的点 R 即为相对于点 Q 的、时刻 t 的首次碰撞点（如图 5.36（b）所示）。

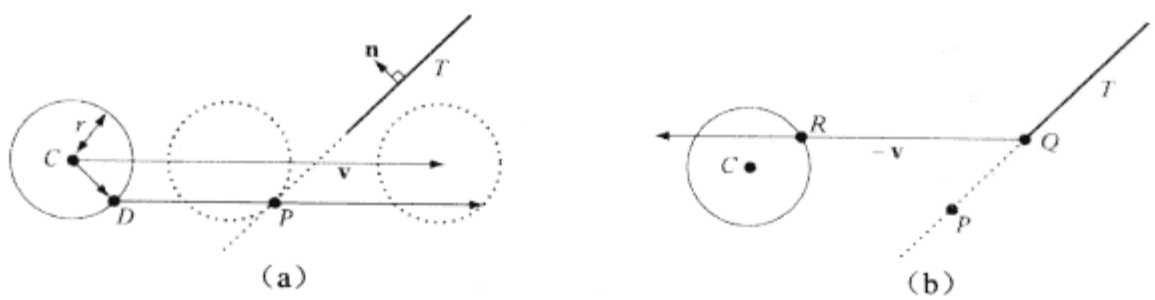


图 5.36　运动球体与三角形相交测试的 Nettle 算法

但是，该算法仍存在健壮性问题。例如，当球体以平行于三角形平面的方式移动时，位移向量与平面之间并不存在相交，从而导致算法崩溃。

然而，该问题在某些情况下也并非无法解决。例如，当多个三角形作为网格的某一部分时，它们

彼此相交并具有相应的公共边。往往在上述问题出现之前，运动球体已经与某一邻接三角形发生碰撞。当然，与三角形平面内移动相关的碰撞反馈也应针对当前情况进行相应的调整。

一种较为健壮同时消耗也较大的算法是：首先执行球体与平面之间的测试（如刚刚讲到的相关算法，或参考 5.5.3 节）。如果二者相交，则首次碰撞顶点 P 将针对 T 执行包含测试，如 Nettle 算法所示。若 P 位于 T 内部，则碰撞有效且测试退出。否则，运动球体将针对三角形的每一条边进行测试。这等价于有向线段 $C+t\mathbf{v}$，$0\leqslant t\leqslant 1$，与转换为半径 r 的圆柱体的各边之间的测试（参见 5.3.7 节）。如果在圆柱体曲面上检测到一次或多次碰撞，则最近碰撞即为首次碰撞并返回相应结果。

假设针对当前顶点并未检测到相交状态，则光线将与半径为 r、球心为三角形顶点的球体进行相交测试。再次，若存在一次或多次碰撞，则返回最近碰撞。如果不存在球体相交，则原球体不与该三角形相交。需要注意的是，光线-圆柱体之间无须进行底面测试：若球体与三角形之间相交，光线将在击中底面之前首先与球体相交于某点。如 5.3.7 节，最后一步测试因此可以省略。

综上所述，从概念上讲，该算法最好视为光线与实体空间 V 之间的测试——其中 V 由 S 针对 T 实施扫掠得到，从而形成了 T 和 S 的 Minkowski 和：$T\oplus S$（参见 3.11 节）——而非上述 3 种通过光线测试或球体测试（依据具体情况）加以实现的独立图元间的测试。5.5.7 节将探讨如何利用 Minkowski 和算法求解运动球体与 AABB 之间的相交问题。

上述第二种健壮型算法可参见[Schroeder01]，其修正版本可参见[Akenine-Möller02]。本节所讨论的这两种算法，都以一种较为直观的方式，归纳了可作用于任意多边形之上的相交测试算法。

5.5.7 运动球体与 AABB 之间的相交测试

令球体 S 定义为球心 C 和半径 r，$\mathbf{d}$ 为 S 的方向向量，可得到球心的位移方程 $C(t)=C+t\mathbf{d}$，其中 $0\leqslant t\leqslant 1$。另外，定义 AABB 为 B。这里，运动球体 S 与盒体 B 之间的相交测试等价于线段 $L(t)=C+t\mathbf{d}$（$0\leqslant t\leqslant 1$）与空间体 V 之间的相交测试，其中 V 由 S 针对 B 实现扫掠得到（即 B 和 S 的 Minkowski 和：$V=B\oplus S$），如图 5.37 所示。

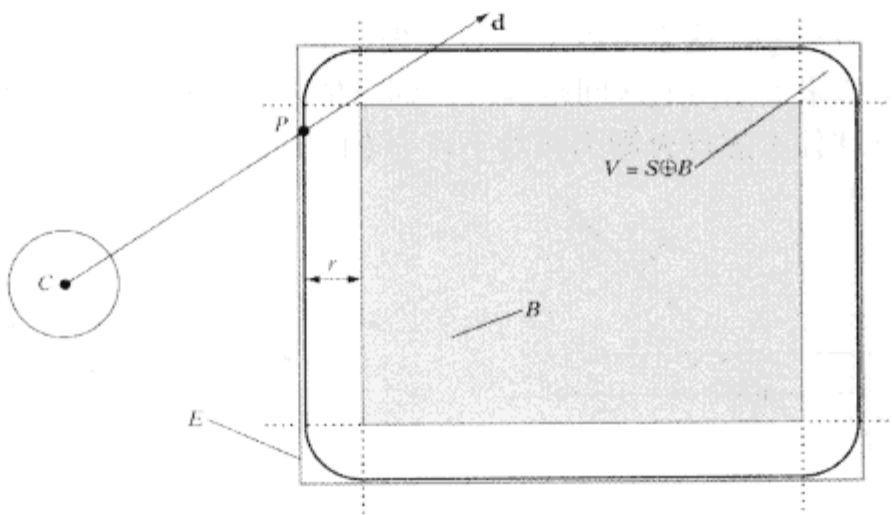


图 5.37　利用球体扫掠 AABB 后（从而构成球体与 AABB 的 Minkowski 和），运动球体与 AABB 之间的测试转换为线段与体空间测试

此处，可以将 B 中各面向外扩展一个球体半径距离 r，并定义该 AABB 为 E。这将构造出包围 V

的、最紧凑的 AABB——与 V 之间的唯一区别在于缺少圆角边。下面，将在 L 与 E 之间执行相交测试。如果 L 不与 E 相交，显然，S 也不与 B 相交。否则，令 $P = L(t)$ 为 t 时刻时 L 与 E 间的交点。

若 P 位于 B 中某一个面的 Voronoi 域，则有：S 在 t 时刻与 B 相交且球心位于点 P。然而，如果 P 位于某一边或顶点的 Voronoi 域，则需要进一步测试 L 是否穿越 V 中的圆角区域（未在 E 中表示）；或测试在某一将来时刻 t'（$t < t' \leqslant 1$）L 与 V 是否相交于某一圆角边或顶点。

若 P 位于 B 中某一边的 Voronoi 域，L 必定还与相应边定义的、半径为 r 的胶囊体（capsule）相交。当且仅当 L 与胶囊体相交时 S 与 B 相交。这里，L 与胶囊体的相交结果对应着 S 与 B 之间的真实相交状态。

最后，考查 P 位于顶点的 Voronoi 域这一情况。此时，L 必定还与对应顶点的、相应边定义的、半径为 r 的 3 个胶囊体相交。如果 L 不与某一胶囊体相交，则 S 与 B 之间也不相交。否则，S 与 B 在历经一个较小时间值 t 后相交，此时，L 与某一边胶囊体相交。作为一种加速方法，L 可与半径为 r、球心位于顶点的球体 K 进行测试。若 L 与 K 相交且交点位于顶点的 Voronoi 域，则该交点即为相应的结果而无须再针对胶囊体执行测试。然而，若 L 与 K 间的交点位于顶点的 Voronoi 域之外（或 L 与 K 不相交），则上述球体测试过程是无效的，且须进行三圆柱体测试。下列代码片段显示了 P 与 Voronoi 域之间的高效包含算法，以及针对边、顶点 Voronoi 域的边测试方案：

```
int IntersectMovingSphereAABB(Sphere s, Vector d, AABB b, float &t)
{
    // Compute the AABB resulting from expanding b by sphere radius r
    AABB e = b;
    e.min.x -= s.r; e.min.y -= s.r; e.min.z -= s.r;
    e.max.x += s.r; e.max.y += s.r; e.max.z += s.r;

    // Intersect ray against expanded AABB e. Exit with no intersection if ray
    // misses e, else get intersection point p and time t as result
    Point p;
    if (!IntersectRayAABB(s.c, d, e, t, p) || t > 1.0f)
        return 0;

    // Compute which min and max faces of b the intersection point p lies
    // outside of. Note, u and v cannot have the same bits set and
    // they must have at least one bit set among them
    int u = 0, v = 0;
    if (p.x < b.min.x) u |= 1;
    if (p.x > b.max.x) v |= 1;
    if (p.y < b.min.y) u |= 2;
    if (p.y > b.max.y) v |= 2;
    if (p.z < b.min.z) u |= 4;
    if (p.z > b.max.z) v |= 4;

    // 'Or' all set bits together into a bit mask (note: here u + v == u | v)
    int m = u + v;

    // Define line segment [c, c+d] specified by the sphere movement
    Segment seg(s.c, s.c + d);
    // If all 3 bits set (m == 7) then p is in a vertex region
```

```
    if (m == 7) {
        // Must now intersect segment [c, c+d] against the capsules of the three
        // edges meeting at the vertex and return the best time, if one or more hit
        float tmin = FLT_MAX;
        if (IntersectSegmentCapsule(seg, Corner(b, v), Corner(b, v^1), s.r, &t))
            tmin = Min(t, tmin);
        if (IntersectSegmentCapsule(seg, Corner(b, v), Corner(b, v^2), s.r, &t))
            tmin = Min(t, tmin);
        if (IntersectSegmentCapsule(seg, Corner(b, v), Corner(b, v^4), s.r, &t))
            tmin = Min(t, tmin);
        if (tmin == FLT_MAX) return 0;          // No intersection
            t = tmin;
        return 1;                               // Intersection at time t == tmin
    }
    // If only one bit set in m, then p is in a face region
    if ((m & (m - 1)) == 0) {
        // Do nothing. Time t from intersection with
        // expanded box is correct intersection time
        return 1;
    }
    // p is in an edge region. Intersect against the capsule at the edge
    return IntersectSegmentCapsule(seg, Corner(b, u^7), Corner(b, v), s.r, &t);
}

// Support function that returns the AABB vertex with index n
Point Corner(AABB b, int n)
{
    Point p;
    p.x = ((n & 1) ? b.max.x : b.min.x);
    p.y = ((n & 1) ? b.max.y : b.min.y);
    p.z = ((n & 1) ? b.max.z : b.min.z);
    return p;
}
```

最后，该算法同样适用于 OBB：通过在 OBB 的局部坐标系统中表示球心 C 以及位移向量 **d**，该问题可归结为运动球体与 AABB 之间的测试。

5.5.8 运动 AABB 之间的测试

如前所述，AABB 是凸多面体的一个实例，对于速度分别为 $\mathbf{v}_A$、$\mathbf{v}_B$ 的两个运动 AABB：A 和 B，其相交测试问题可通过 5.5.2 节中描述的“动态”版本的分离轴测试方案加以解决。

该问题首先出现于静态盒体 A 与运动盒体 B 之间的测试，其中将 B 的速度减去 A 的速度求得相对速度 $\mathbf{v} = \mathbf{v}_B - \mathbf{v}_A$。如图 5.38 所示，盒体 B 朝向盒体 A 运动，其中 d_{first} 和 d_{last} 分别代表了 B 首次、最后一次与 A 碰撞的当前距离。因此，碰撞时间值 t_{first} 和 t_{last} 可通过上述距离值除以 B 的相对速度得到。另外，该时间值须针对全部 3 个主轴进行计算。其中，最大 t_{first} 值以及最小 t_{last} 值对应盒体在各轴上相交时间值的交集。同时，若盒体在轴上的投影彼此不相交且相互背离运动，则测试即刻退出并返回“不相交”这一信息。该测试的实现代码如下所示：

```
// Intersect AABBs 'a' and 'b' moving with constant velocities va and vb.
// On intersection, return time of first and last contact in tfirst and tlast
int IntersectMovingAABBAABB(AABB a, AABB b, Vector va, Vector vb,float &tfirst, float &tlast)
{
    // Exit early if 'a' and 'b' initially overlapping
    if (TestAABBAABB(a, b)) {
        tfirst = tlast = 0.0f;
        return 1;
    }

    // Use relative velocity; effectively treating 'a' as stationary
    Vector v = vb - va;

    // Initialize times of first and last contact
    tfirst = 0.0f;
    tlast = 1.0f;

    // For each axis, determine times of first and last contact, if any
    for (int i = 0; i < 3; i++) {
        if (v[i] < 0.0f) {
            if (b.max[i] < a.min[i]) return 0;        // Nonintersecting and moving apart
            if (a.max[i] < b.min[i]) tfirst = Max((a.max[i] - b.min[i]) / v[i], tfirst);
            if (b.max[i] > a.min[i]) tlast = Min((a.min[i] - b.max[i]) / v[i], tlast);
        }
        if (v[i] > 0.0f) {
            if (b.min[i] > a.max[i]) return 0;        // Nonintersecting and moving apart
            if (b.max[i] < a.min[i]) tfirst = Max((a.min[i] - b.max[i]) / v[i], tfirst);
            if (a.max[i] > b.min[i]) tlast = Min((a.max[i] - b.min[i]) / v[i], tlast);
        }

        // No overlap possible if time of first contact occurs after time of last contact
        if (tfirst > tlast) return 0;
    }
    return 1;
}
```

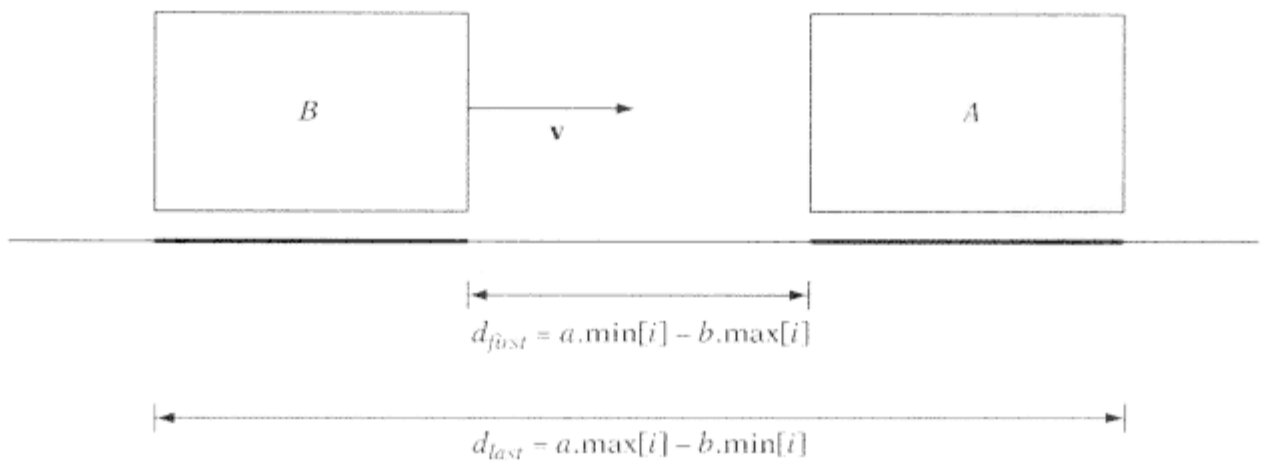


图 5.38　当盒体 B 移向静止盒体 A 时，且盒体 B 的投影首次、最后一次与盒体 A 的投影碰撞时，盒体 B 的移动距离

[Gomez99]中也给出了一个类似的算法。

5.6 小结

本章详细讨论了多种不同类型的测试与查询计算，包括：最近点计算（直接计算目标对象间的距离）；异类型对象间的相交测试（如球体与 OBB）；直线、光线以及线段的相交计算（如与三角形之间的相交测试）；点包含测试（针对多边形和多面体）等。除了静态测试，本章还阐述了动态测试中的相关算法，并着重分析了分离轴测试算法。

尽管本章涵盖了多种测试类型，读者仍会遇到文中未曾提及的某些特定算法。在本章的开篇处曾经提到，理解测试范围的具体含义以及相关的数学分析过程，将有助于读者领会其中的计算思想，并对相关测试算法实现正确的演绎。这种继承的能力是非常重要的，因为相交测试算法的数量、范围毕竟有限。即使存在，也很可能无法满足用户的特定测试需求。这里，[Schneider02]不失为一类优秀的资源，其中收录了类型较为全面的几何测试内容，并可作为本章的有效补充。与之相比，[Eberly01]和[Bergen03]虽然在内容的全面性方面不及[Schneider02]，但仍具有较强的参考性。另外，“GPU 精粹 1～5”中也收藏了针对特定测试算法的独立文献（[Glassner90]、[Arvo91]、[Kirk92]、[Heckbert94]和[Paeth95]）。

第 6 章　层次包围体技术

采用包围体包围物体对象，并在几何图元测试之前首先针对包围体执行计算，这将有效地改善测试性能。然而，虽然测试过程得到了简化，但两两组间测试数量仍未改变。同时，测试结果仍保持原有的渐进时间复杂度，且包围体技术只是通过常数级对测试过程加以调整。此处，通过将包围体整合至树结构中，即层次包围体技术（BVH），其时间复杂度可降至 log 级别。

其中，原包围体集合将对应层次树中的多个叶节点，于是叶节点可形成多个小集合，并可采用一个较大的包围体加以限定。这一过程可以递归方式执行，最终将构造出一棵层次树结构并在树根节点处呈现出一个独立包围体。图 6.1 显示了基于 5 个对象的 AABB 层次树结构。

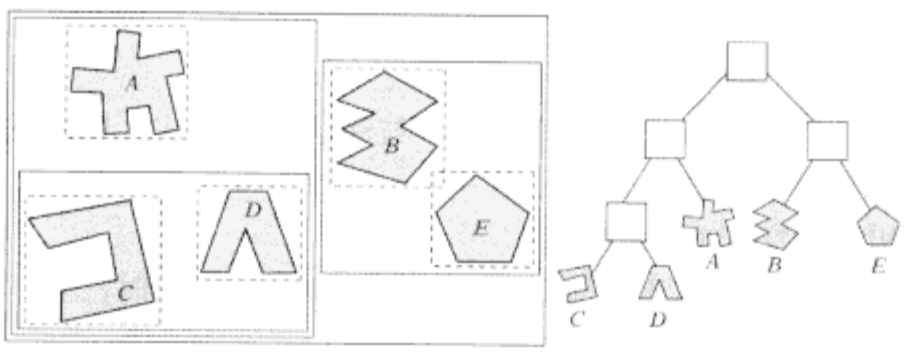


图 6.1　5 个简单对象的包围体层次结构。这里，相应包围体使用了 AABB

采用相应的层次结构时，在碰撞测试过程中，如果父节点不存在相交，则无须对子节点进行检测。其中，该层次包围体还可用于场景图、光线跟踪以及视锥体的剔除计算。

针对层次包围体技术与空间划分策略，相比较而言，二者的主要差别在于：BVH 中的两个或多个盒体可占据同一空间，且物体对象只能置入一个盒体中。相反，在空间划分策略中，划分空间彼此分离且物体对象一般也可以占据多个划分空间。

需要注意的是，父节点的包围体无须包含子节点的包围体。虽然构造这类具有父-子特征的层次结构并非难事，但需要强调的一点是，父包围体中只能够包含分布在子树中的对象图元。

一种创建层次结构的方法是：设计师或美工人员手工构造局部模型层次结构。然而，这种手工创建树型结构的方案并非是一类理想模式。首先，设计师往往趋于感官思考而非空间层次结构。因此，最终结果很可能是：复杂结构设计中的全部细节皆位于同一节点中。从碰撞检测角度加以考虑，这并非是一种最佳分组方案。其次，常见的错误往往出现在对层次结构深度的处理上——过深或过浅都将导致错误的产生。一般来讲，设计师的模型层次结构总是建立在易于编辑的基础之上，而非高效的碰撞查询基础之上——事实证明确实如此。虽然从理论上讲，设计师可以调整其模型层次结构以更好地适应空间特征以及碰撞查询，但这往往会受限于时间进度，从而使设计过程变得低效。一类较好的解决方案是根据所提供的模型自动生成相应的层次结构。但这一自动过程的实现并非易事，且需要多方面考虑各种问题，在 6.1 节中将对此加以讨论。

6.1 层次结构设计问题

目前存在多种方法可以构建层次包围体。6.1.1 节将扼要地阐述良好层次结构所包含的期望特征。6.1.2 节针对各种层次规划，将定义一个通用的性能测试函数，并用于比较各类查询的相关性能。最后，在 6.1.3 节中将讨论树度数对最佳层次结构所产生的影响。

6.1.1 BVH 的期望特征

与包围体类似，[Kaplan85]、[Kay86]、[Hubbard96]中给出了包围体层次结构的某些期望特征：

- ☑ 子树中的节点应彼此靠近。此处，并没有对“靠近”加以显式定义，但随着树深度的增加，节点间应彼此更加紧凑。
- ☑ 层次结构中的节点应是最小盒体。
- ☑ 全部包围体体积之和应保持最小。
- ☑ 应谨慎处理层次结构中根节点附近的节点。与深层节点相比，删除树根节点附近的节点往往意味着移除更多的物体对象。
- ☑ 兄弟节点间的相交体空间应保持最小。
- ☑ 层次结构应参照节点的结构及其数据内容尽量保持平衡。有时，当不再需要某一子树时，平衡树将更有利于层次结构中的删除操作。

对于实时应用程序，尤其是游戏程序，除上述内容以外还应满足下列要求：最坏情况下的查询时间不应超出平均查询时间太多。对于游戏机游戏而言，这一点是尤为重要的——这一类游戏往往要求固定的帧速率（一般为 60fps）。

另外，相应的层次结构应该可以自动生成且无须用户干涉。对于实时应用程序来讲，多数层次结构通常是在预处理阶段加以构造的，而非运行期。对于游戏而言，如果预计算结构的建立时间过长，则对关卡的构建与设计都将带来负面的影响。因此，虽然预计算过程会在一定程度上降低构造时间，但即使采用了预处理方案，平方级复杂度的算法仍然会大大降低计算速度。如果在运行期间构造层次结构，由于构造时间应小于采用层次方案所节省的时间，因此时间消耗主要取决于层次结构自身的建立过程。

最后，碰撞检测过程中常被忽略的一个重要因素是表现包围体层次结构的相关数据结构所占用的内存。例如，在游戏机游戏中，约有 10%的已用内存将分配给碰撞数据。另外，虽然次世代游戏系统的内建内存空间不断增长，但其他子系统（如渲染系统、动画系统，AI 系统）也会导致内存需求出现相应增长，同样，分配给碰撞数据的内存比例基本保持不变。因此，这些与内存相关的约束条件都严重限制了碰撞检测系统的设计。

6.1.2 性能函数

对于层次包围体查询的期望性能，一些研究者曾提出了相关的公式，并分析了各项内容所起的作

用。[Weghorst84]中首先给出了该性能公式，[Gottschalk96]将其应用于层次包围体计算中。随后，[Klosowski98]和[He99]对公式进行了修正，如下所示：

$$T = N_V C_V + N_P C_P + N_U C_U + C_O$$

此处，T 为两个层次结构相交测试时的最终测试性能；N_V 为相交测试时 BV 对的数量；C_V 为相交测试时两个 BV 间的测试消耗；N_P 为相交测试时的图元对数量；C_P 为图元对之间的测试消耗；N_U 为需要更新的节点数量；C_U 为更新这类节点时的所需消耗。另外，必要时，C_O 为一次性操作过程中所需的消耗值（如两个物体间的坐标转换操作）。

在这些变量（如 N_V 和 N_P）中，可以通过构造紧凑的包围体将其数量降至最低；通过实现快速测试，C_V 和 C_P 的消耗值也可降至最低。但是，紧凑的包围体将耗费更多的测试时间；而另一方面，又能够降低测试数量。一般来讲，这一类变量值彼此相互关联：某一变量值的减少往往会导致另一变量值的增加。在现有的条件下，寻找其折中值对于碰撞检测系统而言都并非易事。

6.1.3　树的度数

在层次包围体的属性表现方式中，如何确定树的度数或分支系数往往是一个值得思考的问题。究竟是采用二叉树较好，还是 3 叉树、d-叉树（d 为某一确定值）？抑或节点可以包含任意个子结点的树型结构？通常情况下，具有较高度数的树的深度值一般较小，从而可降低树根至叶节点之间的遍历时间；另一方面，这也会增加每一个访问节点的计算量：随着当前访问节点的子节点的增加，二者相交测试的次数也将增长。相反，对于具有较低度数的树型结构，虽然树的深度值较大，但每一个访问节点的计算量将会减少。对于包含 n 个叶节点的 d 叉树而言，全部节点数量为$(nd-1)/(d-1)$，其中包含了$(n-1)/(d-1)$个内部节点。由此可见，树的度数越大则构建树型结构所需的内部节点数量就越少。

关于树型结构度数的确定，该问题还是具有一定难度的，且至今也不存在一个既定答案。在实际应用中，往往采用二叉树实现这类层次结构。这里，一个重要的原因就是二叉树易于构造，且与其他类型的树型结构相比具有更简练的表现方式以及遍历方法。例如，当自顶向下构造一棵树，并将对象集划分为两个子集时，只需一个分割面即可。可以采用 $2^{m-1}-1$ 种方法将 m 个对象划分为两个（非空）子集，其中，该表达式随子划分数量的递增以指数方式增长（因此不建议使用）。

[Klosowski98]、[Konečný98]中的观点也趋向于采用二叉树。另外，查询的实际性能取决于遍历层次结构时采取何种下降规则。例如，给定一棵包含 n 个叶节点的平衡 d 叉树，“在 B 之前下降 A”与“A、B 同时下降”两种规则（稍后介绍）的消耗分别正比于 $f(d) = d\log_d(n)$和 $f(d) = d^2\log_d(n)$。对于前者，当 d= 2.718 时该式有最小值；对于后者，当 $d = 2$ 时该式有最小值——因而建议使用二叉树，抑或采用 3 叉树作为一种优化方案。经验表明，$d = 2$ 一般作为优先选择方案，这也并非最后结论，尤其是采用某些具有较大度数的树型结构时。

另外，平台架构对于树结构类型的选择问题也饰演着重要的角色。理想情况下，树结构在内存中的布局应使其节点在遍历时呈线性排列。关于如何高效地缓存树型结构，可参见第 13 章。

6.2　层次结构的构建策略

如前所述，随着输入集中元素数量的增长，树型结构的选择方式相应地按照指数方式增长，因此，

采用穷举搜索方式确定最佳树在实际应用中并不可行，并且这类算法将会遗漏相应的最佳树。一般情况下，通常采取启发式规则构建树型结构，并在每一个决策步骤中进行相应的筛选。次优树算法往往易于实现，相对于最优树而言，次优树的数量也较为庞大。

关于树型构造算法，主要分为 3 类：自顶向下、自底向上以及插入算法（如图 6.2 所示）。其中，自顶向下算法将输入集划分为两个（或多个）子集，并利用包围体加以构造且继续在该包围体中实现递归划分。鉴于该算法的易用性，自顶向下算法应用较为广泛。但通常情况下，该算法无法生成最佳树。

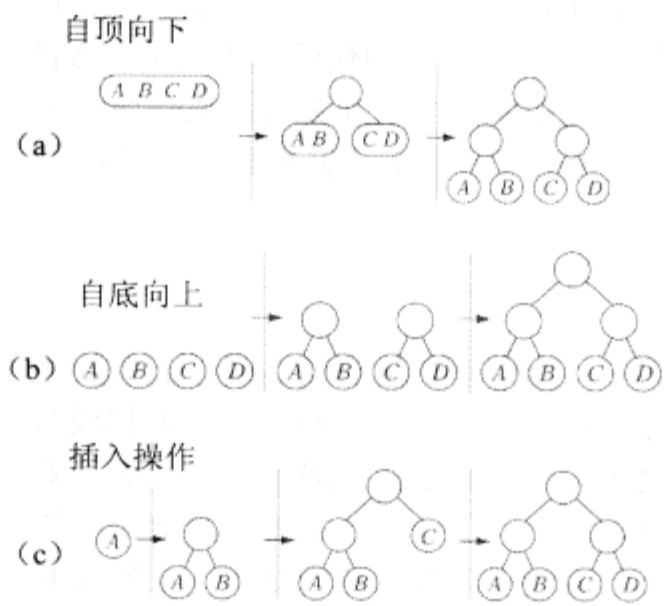


图 6.2　针对 4 个对象构造树型结构。（a）采用自顶向下的构造方法。（b）采用自底向上的构造方法。（c）采用插入操作的构造方法

自底向上算法首先将输入集视为叶节点，并对其进行分组计算以实现一个新的（内部）节点，该过程重复执行直至分组至树根节点。与其他算法相比，虽然自底向上算法能够生成最佳树，但该算法实现起来较为困难。

插入算法通过一次插入一个对象至树中逐步构建树型结构。这里需要对插入位置进行选择，以降低生成树的消耗。插入算法可认为是一类在线（on-line）算法；自顶向下和自底向上算法可视为一类离线算法——算法要求相关图元事先已存在。在线算法的优点是可在运行期间执行更新操作。但是，针对碰撞检测系统层次结构的构造，与插入算法相关的研究还不是十分丰富。

如前所述，虽然多数层次构造处理过程出现在预处理阶段，但是获取快速的构建算法依然是十分重要的。任何复杂度大于 $O(n^2)$的算法对于包含大量图元的层次结构构造过程而言，其速度仍然十分缓慢。

为了简化层次结构的表现形式，在后续小节中将主要采取二叉树。相关算法一般也可应用于 n 叉树甚至是广义树中。

6.2.1　自顶向下的构造方法

自顶向下的构造算法可以按照递归的方式加以描述。算法始于输入图元（或物体对象）集合的包围体。在后续操作中，上述图元将划分至两个子集中，该过程将再次针对这两个子集实现递归调用以

构造分支层次结构，同时作为子节点链接至其父包围体节点上。当输入集合只包含单一图元时，递归过程结束且退出之前须构造该图元的包围体。下列代码片段显示了自顶向下算法的实现过程：

```
// Construct a top-down tree. Rearranges object[ ] array during construction
void TopDownBVTree(Node **tree, Object object[ ], int numObjects)
{
    assert(numObjects > 0);

    const int MIN_OBJECTS_PER_LEAF = 1;
    Node *pNode = new Node;
    *tree = pNode;
    // Compute a bounding volume for object[0], ..., object[numObjects - 1]
    pNode->BV = ComputeBoundingVolume(&object[0], numObjects);

    if (numObjects <= MIN_OBJECTS_PER_LEAF) {
        pNode->type = LEAF;
        pNode->numObjects = numObjects;
        pNode->object = &object[0];      // Pointer to first object in leaf
    } else {
        pNode->type = NODE;
        // Based on some partitioning strategy, arrange objects into
        // two partitions: object[0..k-1], and object[k..numObjects-1]
        int k = PartitionObjects(&object[0], numObjects);
        // Recursively construct left and right subtree from subarrays and
        // point the left and right fields of the current node at the subtrees
        TopDownBVTree(&(pNode->left), &object[0], k);
        TopDownBVTree(&(pNode->right), &object[k], numObjects - k);
    }
}
```

除了选择包围体的类型，另一个影响结果树结构的指导原则是：输入集合如何正确地划分为两个子集。如前所述，包含 n 个元素的集合共存在 $2^{n-1}-1$ 种方案可以将其划分为两个非空子集，显然，只有少数方案真实可行。

为了简化划分过程，通常采用分割超平面将原集合划分为两个子集。有时，会出现所选分割面跨越图元这一情况，且集合划分过程中，对这类跨越图元须采用某种方式加以正确处理。一种方法是将图元一分为二且置入对应子集中。同时，图元分割将使子包围体较小，从而降低或完全消除了相应的相交测试量。但是，对跨越图元实施分割也存在一个缺陷：分割图元可能再次面临分割——这将使得图元数量急剧增加。

另一种常见算法是不对图元进行分割，而是计算图元相对于分割面的质心位置，从而确定该图元属于哪一个子集。此处，可利用质心坐标确定图元的子集划分试图将兄弟包围体间的相交计算量降低至最低。通过这种方法，需要对包围体的宽度进行扩充，扩充范围为图元宽度的一半。当图元被划分至某一子集时，其最坏情况是子集包围体扩展为图元的全部宽度。

6.2.1.1　划分策略

一种简单的划分算法称为中值切割算法。此处，集合将根据某一轴向上的投影分为等尺寸的两子

集，从而生成平衡树。如前所述，考虑到包围体的期望特征，还包括其他一些划分策略，如下所示：

- ☑ 最小化子节点包围体空间（或表面积）之和。包围体与某一子节点包围体的相交几率一般正比于体积空间。因此，最小化包围体空间之和将有效地降低相交测试几率。例如考查光线相交测试：其击中包围体的几率正比于包围体的表面积。
- ☑ 将子节点中的最大包围体（或表面积）最小化。由于上一策略有可能导致两子节点间的包围体尺寸不均衡，该策略将最小化较大包围体空间，从而使得两包围体空间在尺寸上处于相对均衡状态。这可以生成较为平衡的树型查询结构，并改善某些最坏情况下的查询操作。
- ☑ 降低子节点包围体的体空间（或表面积）。该策略将减少相应子节点的相交测试以及访问几率。依据所选用的包围体类型，测试过程将呈现不同的复杂度以及近似度。
- ☑ 尽可能地分离子节点包围体。即使不相交，分离子节点也可最大限度地降低二者的访问几率。
- ☑ 均等分割子节点包围体间的图元。该策略对应着本节开始处提及的中数切割算法。当对树型结构的平衡性要求较高时，可考虑这一方案。
- ☑ 组合应用上述划分策略。

当满足某些标准时，划分过程即停止且其节点可视为叶节点。其中，常见的一些准则包括：

- ☑ 节点只包含单一图元或小于 k 个图元。
- ☑ 包围体空间小于某一限定值。
- ☑ 当前节点深度到达某一限定深度值。

除了上述这些准则，划分过程也可能在早期即出现异常，例如：

- ☑ 全部图元位于分割面的一侧。
- ☑ 父节点、子节点包围体中的图元数量接近。
- ☑ 父节点、子节点包围体尺寸接近。

这一类失效条件也可视为促使划分过程结束的相关准则。另外，在划分过程结束之前，也可尝试其他类型的划分准则。例如，针对基于盒体的树型结构，当沿最大面划分失效后，可依次对次大面、最小面进行测试。仅当全部 3 个面皆失效时划分过程结束。这里，划分过程结束于 k 个图元而非单一图元，这将有助于节省相应的内存空间。但也应该看到，采取叶节点-叶节点间的测试而非单一图元间的测试方案，将导致算法复杂度为 $O(k^2)$。

分割面的选择通常包含两个步骤，首先应确定相应的分割轴，其次计算沿该轴向上的某一位置，相关内容可参见 6.2.1.2 节。

6.2.1.2 分割轴的选择

这里，存在无限多个可能的候选分割轴，需要以某种方式进行筛选以确定最终结果。理论上，可以采用迭代优化算法（如爬山算法）获取最佳分离轴。但是在实际应用中，该算法即使在预处理阶段也将十分耗时。因此，搜索过程只能在较小范围内获取最佳分离轴。在规定范围内，常见的选择方案包括以下几类：

1. x、y、z 局部坐标轴。操作方便是这一类方案的主要特点。该方案形成一个正交集且覆盖各个方向。

2. 选择具有对齐性质的包围体中的轴。例如，AABB 的面法线一般为首选轴，其他一些包围体，（如 k-DOP）往往包含附带的固定轴并可作为分割轴的首选方案。

3．选择父节点包围体中的轴。如果层次结构采用 OBB 加以构造，则当前数据划分的最佳候选方案为该 OBB 中定义的坐标轴。甚至，若层次结构基于球体建立（不存在明显的关联轴），仍可采用一个临时的 OBB 用于计算父节点的数据集，从而获取相应的分割轴。

4．选择具有最大间距的两点上的轴。沿该轴进行划分意味着将子节点包围体空间降至最低。此处，一种与最大间距点近似的次优算法可参见 4.3.2 节中的 **MostSeparatedPointsOnAABB()**函数，该函数采用启发式算法且复杂度为 $O(n)$。

5．选择具有最大方差值的轴。在该轴向上，输入数据具有最大分布特征，因此可视为将子节点包围体空间降至最低。例如，在可体现方差特征的 OBB 所构造的层次结构中，最大方差轴将与包含父节点 OBB 间距最大的面的轴向相对应。

即使上述优化步骤无法实现，一旦选择了某一分割轴，仍然可采用爬山算法中的相关步骤对该轴加以改进。这里，可以对该轴向实施某种干扰，若结果轴工作良好则可替代原始轴。可反复执行这一过程直至满足相应要求。

可以采用统计方法而非主成分分析方法来处理分割轴问题。这类算法虽然值得研究，但仍未引起广泛的重视。这其中包括：投影寻踪算法、独立成分分析算法以及盲信号分离算法。这类技术将从混合观测数据源中分离非观测独立组元。例如，投影寻踪算法将获取最大投影数据熵（而非方差）的方向。由于熵是一种记录数据信息或“兴趣度”的测量方法，且数据聚类间将具有较高的熵值。因此，在将数据划分至多个聚类的过程中，该方向可作为分割轴的最佳候选方案。关于投影寻踪算法和独立成分分析算法的相关介绍，可参见[Stone98]；关于盲信号分离算法，可参见[Cardoso98]。

6.2.1.3　分割点的选择

上述 5 点优化步骤在选择分割点时同样会失效。由于沿某一轴向上存在无数多个分割点，因此选择方案仍需限定在某一范围内。例如：

- ☑ 质心坐标的中值（对象中值）。在对象中值处实施分割将使得子集间的图元呈均匀分布状态，从而可生成较为平衡的树型结构。通过对顶点排序，该中值算法的时间消耗可达到 $O(n \log n)$ 级别。另一种方案稍显复杂，其算法的时间消耗为 $O(n)$，详情可参见[Cormen90]。
- ☑ 质心坐标的平均值（对象平均值）。具有良好平衡特征的树型结构有时无须提供最佳查询时间。当沿最大方差轴进行分割时，[Klosowski98]指出：采用对象均值将优于对象中值方案，并且连续进行均值分割将生成空间紧凑的树型结构、减少相关的操作量并获得快速的查询时间。对象均值算法的时间消耗可达到 $O(n)$级别。
- ☑ 体空间投影中值（空间中值）。空间中值分割算法也是一类可供选择的方案（均等分割体空间）：算法只检测包围体且不包括其中的图元数据，分割点计算过程的时间消耗为常数级。该算法常用于父节点包围体中的既定轴上（如利用最大间距面定义的轴）。
- ☑ 在包围体投影范围内的 k 个均匀分布的顶点处进行分割。该蛮力算法仅在该轴向上测试少量均匀分布的空间顶点，并选择最佳点。
- ☑ 在质心坐标（随机子集）间进行分割。与前述算法类似，在投影质心坐标间进行分割将使跨越分割面的图元数量降至最低，但该算法需要额外投影并排序质心坐标。

图 6.3 显示了一些分割方案。此处，可以逐步构造数据子集而非直接选取分割点。例如，[Zachmann98]中采用了在最大间距点 a、b 构成的轴上进行分割。首先，将与 a 和 b 关联的面加入两个不同的子集中，

其次，将剩余图元赋予某一子集中，且该子集包围体的增长量小于所添加的图元。若两包围体增长量相同或未曾出现增长（取决于完全位于包围体中的图元），则图元添加至具有较少图元的集合中。

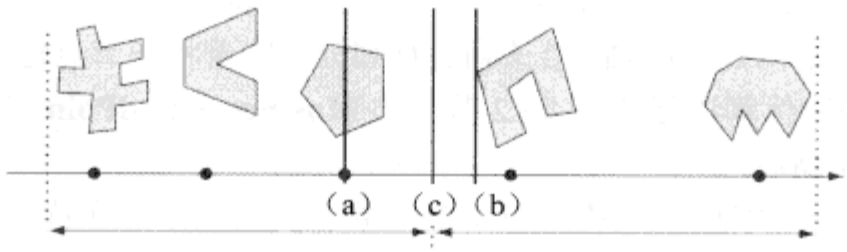


图 6.3　（a）在对象中值处分割。（b）在对象均值处分割。（c）在空间中值处分割

自顶向下方法是最为常见的一类包围体层次构造算法，其优点是算法易于实现且可以快速建立树型结构；缺点是在算法早期且缺乏全部数据信息时即需要执行较为严格的决策判断，因此，一般情况下很难得到最优树。

6.2.2　自底向上的构造方法

与自顶向下方法相比，自底向上方法的实现过程较为复杂且构建过程也比较耗时，但通常能够生成最优树（[Omohundro89]）。采用自底向上方法构造树型结构时，首先需要建立图元的包围体，这些包围体形成了树型结构的叶节点。基于某种合并策略（或称之为聚类策略），可以从包围体结果集中选择两个（或多个）叶节点，并再次构造其包围体且替代集合中的原节点。这一组合过程持续进行，直至集合中只包含树根节点的包围体。

```
Node *BottomUpBVTree(Object object[], int numObjects)
{
    assert(numObjects != 0);

    int i, j;

    // Allocate temporary memory for holding node pointers to
    // the current set of active nodes (initially the leaves)
    NodePtr *pNodes = new NodePtr[numObjects];

    // Form the leaf nodes for the given input objects
    for (i = 0; i < numObjects; i++) {
        pNodes[i] = new Node;
        pNodes[i]->type = LEAF;
        pNodes[i]->object = &object[i];
    }
    // Merge pairs together until just the root object left
    while (numObjects > 1) {
        // Find indices of the two "nearest" nodes, based on some criterion
        FindNodesToMerge(&pNodes[0], numObjects, &i, &j);
        // Group nodes i and j together under a new internal node
        Node *pPair = new Node;
        pPair->type = NODE;
```

```
        pPair->left = pNodes[i];
        pPair->right = pNodes[j];
        // Compute a bounding volume for the two nodes
        pPair->BV = ComputeBoundingVolume(pNodes[i]->object, pNodes[j]->object);

        // Remove the two nodes from the active set and add in the new node.
        // Done by putting new node at index 'min' and copying last entry to 'max'
        int min = i, max = j;
        if (i > j) min = j, max = i;
        pNodes[min] = pPair;
        pNodes[max] = pNodes[numObjects - 1];
        numObjects--;
    }
    // Free temporary storage and return root of tree
    Node *pRoot = pNodes[0];
    delete pNodes;
    return pRoot;
}
```

这里仍遵循前述小节中所描述的逻辑特征。一类具有应用价值的合并策略是：选择一组节点并使其包围体空间最小。当然，可以采取蛮力算法搜索两个节点并实现合并，即考查全部可能的组合并计算其包围体，最终得到具有最小包围体的两个节点。需要注意的是，该蛮力算法的时间复杂度为 $O(n^2)$。另外，构造一棵完整的树还需循环 $n-1$ 次，因此，构造过程的全部时间复杂度为 $O(n^3)$。

6.2.2.1　改进后的自底向上构造方法

一类较为完善的实现方式可以充分改进上述蛮力算法。这里，不再连续、重复计算节点对之间的最小包围体，节点可记录其对应的首选组合（pairing）节点和二者的包围体空间。因此，在任意给定时刻，当前节点及其存储的组合节点之间可实现最佳合并策略。其中，节点-组合节点-包围体三元组可以采用一类典型的数据结构加以维护。例如，选择优先队列（堆或二叉搜索树）实现包围体的排序可以加快最小包围体的访问速度。

当出现新的组合节点时，多数已存储的最小包围体组合节点将保持不变，且涉及某个新增组合节点时，已存储的相关组合内容才会受到影响。更重要的是，当组合节点产生变化时，其包围体空间必定增加。因此，组合节点可实现延迟计算——即该组合节点置入优先队列时。

实际上，该算法已衍变为一种迭代计算，其中当前最佳组合节点从队列中被移除。具体操作为：若节点的组合节点存在，则简单地丢弃该组合节点并在适当时置入新的最佳组合节点，否则计算节点的最佳组合节点。如计算结果与已存储的组合节点匹配，则该结果节点包含最小包围体且二者可进行相应的合并操作；如计算结果匹配失败，则对应的组合节点定在初期被丢弃。此时，可将该节点及其组合节点重新置入优先队列，并赋予新的包围体优先值。

需要补充的一点是，当获取一查询节点后，如何快速地计算组合节点并构造最小包围体？对于存储部分构造完毕的、自底向上的树结构，动态包围体树（稍后将介绍一种自顶向下的扩充插入算法）则是一类较好的数据结构。针对此类数据结构，可从树根节点处开始搜索，并降至与查询包围体相交的全部子节点，从而定位组合节点。当在层次结构中获取查询包围体后，组合节点即为该包围体父节点的某一子节点（即兄弟节点）。下列代码片段显示了上述改进算法的实现过程：

```
Node *BottomUpBVTree(Object object[], int numObjects)
{
    PriorityQueue<Pair> q;
    InsertionBVTree t;
    // Bound all objects in BV, forming leaf nodes. Insert leaf nodes into a
    // dynamically changable insertion-built BV tree
    InitializeInsertionBVTree(t, object, numObjects);

    // For all nodes, form pair of references to the node and the node it pairs
    // best with (resulting in the smallest bounding volume). Add all pairs to
    // the priority queue, sorted on increasing volume order
    InitializePriorityQueue(q, object, numObjects);

    while (SizeOf(q) > 1) {
        // Fetch the smallest volume pair from the queue
        Pair *p = Dequeue(q);

        // Discard pair if the node has already been paired
        if (HasAlreadyBeenPaired(p->node)) continue;

        // Recompute the best pairing node for this node to see if
        // the stored pair node is still valid
        Node *bestpairnode = ComputeBestPairingNodeUsingTree(t, p->node);
        if (p->pairnode == bestpairnode) {
            // The store pair node is OK, pair the two nodes together;
            // link the nodes together under a new node
            Node *n = new Node;
            n->left = p->node;
            n->right = p->pairnode;

            // Add new node to BV tree; delete old nodes as not possible to pair with
            Delete(t, p->node);
            Delete(t, p->pairnode);
            Insert(t, n);

            // Compute a pairing node for the new node; insert it into queue
            Node *newbestpairnode = ComputeBestPairingNodeUsingTree(t, n);
            p = Pair(n, newbestpairnode);
        } else {
            // Best pair node changed since the pair was inserted;
            // update the pair, reinsert into queue and try again
            p = Pair(p->node, bestpairnode);
        }
        Enqueue(q, p, VolumeOfBVForPairedNodes(p));        // Queue, pair, priority
    }
    return Dequeue(q)->node;
}
```

一种基于构造球体树的类似算法可参见[Omo-hundro89]。

6.2.2.2　其他类型的自底向上构造策略

将常规节点分组为两个物体对象时，可生成最小包围体的组合对象极有可能是两个彼此最为邻近的对象。因此，合并策略常简化为查询节点与其最近邻接节点之间的合并操作。

对于给定的查询点，从相应点集中确定其最近点（或对象）可称作最近邻接问题。经过深入研究，该问题已有多种不同的解决方案。当对象数量较少时，可采用低消耗的蛮力算法；而对于大量的物体对象，基于分桶方案（bucketing scheme）或 *k*-d 树的算法往往更加有效。*k*-d 树算法实现起来则更为直观（详情可参见 7.3.7 节）。*k*-d 树可以按照自顶向下的方式加以构造，当在对象中值处二分当前对象集时，其平均时间复杂度为 $O(n \log n)$。

最近邻接查询常限定于 *k*-d 树中的某一局部范围内，并记录当前节点的范围值（参见 7.3.7 节）。初始状态下，该值记为查询对象与根节点对象之间的距离。待获取较近对象之后，范围值也将相应地减少。同时，对于距离大于当前范围值的半空间将予以剔除，因其不可能包含近对象数据。另外，若正负半空间皆被保留，则应首先下降（遍历）包含查询对象的该半空间：这通常能够得到更小的范围值，从而加速计算过程。

k-最近邻接问题也可采取类似的方法，只需添加包含 *k* 个对象的优先队列，以记录当前 *k*-最近对象并通过队列中的最新（最远）元素更小范围距离。平均情况下，*k*-d 树搜索和插入操作的时间复杂度为 $O(\log n)$。关于 *k*-d 树的进一步讨论，可参见第 7 章和第 13 章。

如前所述，上述计算过程并不能保证生成相应的平衡树。实际上，对可能导致退化树的输入数据进行核对不失为一种简便的方法。另外，可在每次迭代过程中构造多组最近点以改善树型结构的平衡性。例如，若每次迭代过程中均可实现全部可能的 $n/2$ 个组合，则最终的结果树将具有良好的平衡特征。这将降低树型结构构造过程中的迭代次数，复杂度也相应地从 $O(n)$ 降至 $O(\log n)$，从而最终改善了其时间消耗。在选择一组最佳节点时，最坏情况下可能需要查询全部组合节点。一种折中方案为：针对某一较小值 *k* 构造 *k* 个组合节点。

对于构造较大的对象聚类，可采用特定的算法。一类方案是将对象视为完全图中的某一顶点，并计算该图的最小生成树（MST），其中各边权值设定为两个邻接对象间的距离，或类似的聚类量值。该 MST 随后通过消除“过长边”而分解为相应的元素，最终保留的分离（但内部连接）元素集即为对象聚类。

MST 可采用 Prim 算法或 Kruskal 算法加以计算——两种算法都属于贪心算法范畴，即选择并执行当前最优解。贪心算法在大多数问题上都工作良好，但并非一般意义上的优化算法。MST 问题可采用上述两种算法中的任意一种。两种算法中，Prim 算法在概念上更加简单且易于实现。该算法始于原集合中的任一顶点，并在非邻接顶点与树型结构间搜索最低权值边，从而扩充树。随后，此边与该顶点连接至当前树。该过程重复执行，直至全部顶点连接完毕。关于 MST 的其他信息，包括 Prim 算法和 Kruskal 算法描述，可参见[Cormen90]和[Skiena98]的简明读本。

6.2.2.3　自底向上的 *n* 叉聚类树

当采用 MST 求解过程作为相关计算的某一关键步骤时，一类值得重视的解决方案是自底向上的聚类算法，参见[Garcia99]。这里，给定一个包含 *n* 个包围球的集合，该算法将在球体间构造邻接图，且图中各边表示球体组合间的分组权值（相关描述可参见后续内容）。根据该图可计算 MST，并从中依次构造球体的层次结构。

由于完全邻接图中的边数量为 $O(n^2)$，这将严重影响 MST 算法的执行时间。Garcia 等人将各包围体的连接数量限制在某一常数 k，从而将复杂度降至 $O(n)$。通过使用相应的空间划分策略，算法还可进一步降至大约 k 个最近邻接节点，但其时间复杂度仍为 $O(n)$。

聚类函数一般与现有各边相关。此处，给定球体 S_i 和 S_j 且半径分别为 r_i、r_j，两球心间距离为 d_{ij}。Garcia 等人首先在 S_i 和 S_j 间提出了下列聚合公式：$a_{ij} = r_i r_j / d_{ij}^2$。其中，若两球体尺寸越大且彼此越靠近，则 a_{ij} 值就越大——这从直观的角度描述了聚类的构造方式。

为了避免在构建早期出现较大的聚类，聚类函数最终定义为 r_{ij}^3 / a_{ij}，其中 r_{ij} 为 S_i 和 S_j 间最小包围体的半径。当生成的包围球较大时，这将有助于降低具有高聚合度的球体组合。

对于包含边权值的邻接图，一旦构造完毕即可计算该图的 MST。其中，MST 的各边可按其权值升序的顺序加以排序。通过排序边且每次考查一条边，最终可构造二叉聚类树，将彼此关联的两个节点合并至一个新的聚类中。若聚类的组合内容已包含当前节点，则该聚类将重组为一个新的聚类。当聚类构造完毕后，将包含节点包围球间的最小包围球，最后，各连接边的权值将对应其组合权值。基于 MST 的 BCT 构造过程，其时间消耗为 $O(n \log n)$。

至此，计算过程似乎告一段落并可以使用生成的二叉树。但如前所述，算法的最后一步需要将 BCT 转化为 n 叉树，即针对某一节点合并具有相似组合权值的连接聚类。当 MST 计算完毕且表中各边按照其权值升序排序后，权值可组合至由多个族（family）构成的某一集合中。

在排序表中，前两个权值元素将生成第一个族 F_0。此处，分别定义 u_i、s_i 为族 F_i 中权值的均值和标准方差值，若 $w < u_i+2s_i$，则表中后续权值 w 也可视为族 F_i 中的元素。当某一权值加至族中时，需要更新该族的相应均值和标准方差值。另外，若权值在族中不存在，则该值与表中其相邻权值元素将生成一个新族。该过程重复执行，只至表中全部权值元素加至族中。最终，当全部族计算完毕，通过合并同族内的邻接聚类可将 BCT 转化为 n 叉树。

6.2.3 扩充（插入）构造策略

最后一类构造方案可称作扩充算法或插入算法。这里，构造过程将始于一棵空树且每次插入一个对象，从而建立相应的树型结构。其中，对象搜索相应的插入位置并执行插入操作，且依据相关权值，树型结构得以逐步扩充。通常，与插入对象所关联的权值采用下列方式加以计算：其包围体的权值加上插入导致的其祖先包围体的权值。

如果插入对象大于现有节点，插入过程将会在层次结构的顶部结束其操作；较小的插入对象将位于现有相关节点的内部，且插入过程将会在层次结构的底部结束其操作；另外，若对象距现有节点距离较远，插入过程也将会在层次结构的顶部结束其操作。综上所述，针对原对象集合，这里所生成的结果树能够反映真实的聚类状态。

由于树型的最终结构将依赖于所插入的对象，同时，对象的插入策略也基于当前树型结构，因此对象的插入顺序将变得十分重要。采用某些编写工具生成的顺序对象，将会产生具有某些缺陷的层次结构，且无法反映真实的对象聚类状态。尤其是对数据实施排序也将会面临同样的问题。为了避免退化现象，在执行插入操作之前，可尝试将对象随机混合。

一类简单的插入算法只是执行根节点-叶节点间的遍历，即连续地降至子节点以使插入操作简单易

行。随后，从回溯路径上的各节点中选择相应的插入节点。对于每一个插入对象，搜索过程的复杂度为 $O(\log n)$，且全部复杂度为 $O(n \log n)$。另一种较为高级的算法是考查全部树型结构，采用最佳-优先方式并利用优先队列维护队首搜索结果，并且算法总是降至当前最佳节点处。上述两种算法的具体内容可参见[Omohundro89]，其中使用了球体树。

插入算法的速度不低于自顶向下的相关方案，并且能产生较好的计算结果。算法可视为一类在线计算，亦即在处理过程起始阶段无须展现全部对象。这一类在线算法也可用于更新现有的树型结构，并适用于动态系统中。令人稍感意外的是，鲜有碰撞检测系统采用这类扩充构造算法。

6.2.3.1　Goldsmith-Salmon 扩充构造算法

[Goldsmith87]中描述了如何构造包围体层次结构并应用于光线测试中，其改进算法可参见[Haines88]。如前所述，算法将在最佳位置一次插入一个对象。通过降至子节点，可获得一条从根节点至叶节点的路径，以及包围体表面积最小增量。最终，对象作为一个子节点执行插入操作。

这里，采用表面积方式则是源于投影几何中的某些计算结果，即 3D 凸体的平均投影面积为其表面积的 1/4。进一步讲，若随机光线击中父节点包围体 A，则击中其下包围体 B 的条件概率将正比于二者的比值。此处，根节点包围体可方便地记作 A，通过直接比较条件概率可避免相应的除法操作。

由于表面积计算只用于比值操作，因此可设定为常数。对于 x、y、z 向的包围体，其面积为 $x(y+z)+yz$；对于一个半径为 r 的球体，其面积为 r^2。

当前，若光线击中根节点包围体，则至少需要执行 k 次相交测试，其中 k 为根节点下子节点的数量。这一方式适用于任一节点，而非只针对根节点。如父节点被击中，其子节点击中的条件概率为：击中某一节点的平均权值将 k 倍于节点表面积与根节点表面积之比。特别地，根节点的权值为其子节点数量之和，叶节点的权值则为 0（仅供计算其父节点权值）。树型结构的权值为全部节点权值之和，且时间复杂度为 $O(n)$。

树权值也可在其构建过程中采取增量的方式加以计算：在层次结构的插入-遍历过程中，还可以向子节点传递“继承权值”这一增量值，且根据相应的对象插入操作，该权值对应于祖先节点包围体的体增量值。这一类基于对象权值的插入操作分为下列 3 种情况：

1．对象与叶节点组合并形成一个新节点。此时，针对继承权值的增量值为 $d = 2Area$(新节点)。该情形也适用于对象与原根节点组合并形成一个新根节点。

2．对象作为一个新子节点加入至某一当前节点。此处，当前节点的权值为 $c = k\,Area$(当前节点)；新节点的权值为 $c' = (k+1)Area$(新节点)。因此，增量值为 $d = c' - c = k\,(Area$(新节点) $- Area$(当前节点)) $+ Area$(新节点)。

3．对象递归至孩子包围体并加入至树型结构的底层。此时，针对当前检测节点，子节点的数量保持不变。然而，在树型结构的底层插入对象将导致节点尺寸上的变化。这里，权值差为 $d = k\,(Area$(新节点) $- Area$(当前节点))。

针对全部插入方案，当权值增量值计算完毕后，即可在每一步中选择最优方案。但有时也会出现多棵子树包含相同的权值增量这一情况。通常，当对象完全位于现有包围体中时，这一状况会出现在构造过程临近尾声时。针对这一情况，Goldsmith 和 Salmon 建议，选择距对象最近的包围体——其中心位置距当前对象最近——并将对象插入该包围体中；或采取随机选择方案，从而阻断相关联结。

Haines 给出了一类较好的选择方案：针对各类插入操作，分别采用上述方法 1 和方法 2 并从中选

出最优结果。作为一个比较有说服力的实例，Haines 考查了包含两个子节点的父节点，其中一个子节点较大（占父节点的 50%），而另一个子节点则很小（占父节点的 1%）。当加入一个新增的较小对象时，较大子节点将不产生任何增量变化，但该增量值却是较小子节点尺寸的 3 倍——达到父节点的 3%。按照最初的方案，对象应插入至较大子节点下方。然而，当根节点产生相交时，较大子节点的相交计算将占据 50%的测试时间。因此，新增对象的相交计算亦将占据全部测试时间的一半。相比较而言，如果新增对象插入在较小子节点处，则其相交计算只占据 3%的测试时间。因此，综合使用方法 1 和方法 2，并选择包含较小增量值的节点，可获得较好的插入位置。

6.3 层次结构的遍历

当包围体树型结构的顶部相交时，需要制定相关的下降规则以确定两个包围体层次结构的相交状态。这里，针对两个层次结构，究竟是按照前后顺序下降还是同时下降？下降规则可视为相交测试代码中的核心部分，后续内容将探讨多种实现方案。

其中，两类最基本的树型遍历算法是广度优先搜索算法和深度优先搜索算法（如图 6.4（a）和图 6.4（b）所示）。广度优先搜索算法（BFS）首先在某一深度层上考查各节点，之后将处理树中更深层的内容；深度优先搜索算法（DFS）将按照树型深度而非广度进行搜索，且当搜索至叶节点时，算法将进行路径回溯。完全意义上的广度优先以及深度优先遍历可视为无启发信息或盲目搜索算法。无启发信息搜索算法并不检测、生成与遍历结构数据相关的决策，算法只是查看结构自身并确定下一个将要访问的节点。

与无启发信息算法形成鲜明对比的是启发式搜索算法。算法将使用通过启发式规则获取的相关域信息。该类算法包括最佳-优先搜索（如图 6.4（c）所示）。最佳-优先搜索可视作一类贪心算法，且总是移至某种搜索策略（如与既定目标之间的距离）下的最佳评估节点。可通过下列方式确定“最佳评估”的走向：维护一个节点的优先队列，扩展当前最佳节点（队列中的首元素）并将其子节点入队。重复这一过程直至搜索失败或发现目标节点。

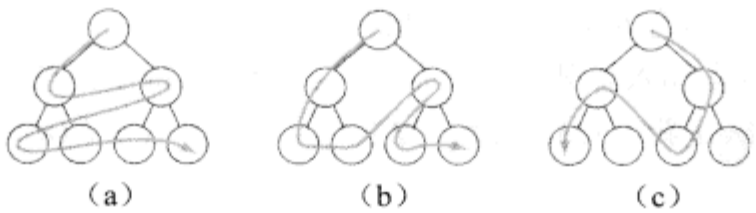


图 6.4 （a）宽度优先搜索。（b）深度优先搜索。（c）一种可能的最佳-优先搜索顺序

DFS 似乎是碰撞检测系统中最常用的选择方案。增强的 DFS 算法常采用某一简单的启发规则来引导搜索的走向，从而改进缺乏最佳-优先搜索思想的、基础型盲目 DFS 搜索方案。

与 DFS 算法相比，BFS 算法存在的问题是：在遍历过程中，采用堆栈方式存储全部节点将占用大量的内存空间。对于近似查询计算，各包含 n 个叶节点的两个二叉树其栈空间相当于同时存储 n^2 个节点-节点对所需的容量。在中断式（interruptible）碰撞检测系统中，主要采用 BFS 算法。其中，需要强调的一点是，如果（或当）查询操作出现中断，层次结构中的每一部分都将等待大约同一时间量。

类似地，相对于启发式引导的 DFS 算法，BFS 算法须经过精心调试方可在性能上得到相应的改善。

最后，在处理智能节点、重排序及管理优先队列时的额外时间消耗，将等同于深度优先算法中固有的、下降至子节点过程中的时间消耗。

6.3.1　下降规则

现在返回到前述讨论的层次结构下降问题。给定两个层次结构 *A* 和 *B*，其中存在多种方法对其实现遍历。例如，可先对 *A* 实施全遍历，再对 *B* 进行遍历，或者二者也可实现同步遍历。作为一个直观的例子（可参见[Chung98]），此处考查下列情形：一只飞鸟（层次结构 *A*）穿梭于布满管道的炼油厂（层次结构 *B*）。存在以下几种下降规则：

- ☑ 首先下降 *A*，然后下降 *B*。将层次结构 *A* 降至叶节点并随后下降 *B*——这并不是一种明智的解决方案。就该例而言，如果飞鸟位于炼油厂中部某处，则飞鸟层次结构中的叶节点将会与油厂层次结构顶部包围体相交。这绝非是一种最佳选择：将导致层次结构 *B* 的递归次数为 *A* 中的叶节点数量。其中，层次结构 *A* 的出现反而适得其反——采用该层次结构大大地增加了工作量。
- ☑ 首先下降 *B*，然后下降 *A*。针对该例，先行下降较复杂的油厂层次结构，且随后下降飞鸟的层次结构将在一定程度上改善计算结果。然而，与飞鸟模型相比，炼油厂模型可能还包含诸多细节部分（如螺母和螺栓等细节），这将产生与上述内容类似的结果。*B* 中的多个叶节点仍需要与 *A* 进行测试。
- ☑ 对较大包围体执行下降操作。通过动态确定当前最大层次结构并执行下降操作，可避免上述两算法中的相关问题。初始状态下，可先对油厂的层次结构执行下降操作，当进行至其细节部分时，切换遍历过程并对鸟类的层次结构执行下降操作。同时，当鸟类的细节部分小于油厂模型中的细节部分时，将再次切换至遍历过程油厂模型中。对于后续的待测包围体，该方案提供了全部包围体中的最大缩减量，因而是一类高效的下降规则。如前所述，在进行尺寸比较时，有效的测试方案包括体空间、表面积以及最大维长。
- ☑ 同步下降 *A* 和 *B*。此处，并不单独对某一层次结构执行下降操作，而是两个层次结构同时下降。同步下降将有助于快速遍历至叶节点，减少内部节点-节点（以及节点-叶节点）间的相交测试，并且在下降过程中不包含额外的评估消耗。但是，在飞鸟-炼油厂这个例子中，当前方案无法像前述规则那样有效地删除搜索空间。
- ☑ 在 *A*、*B* 间轮流执行下降操作。在层次结构间可依次执行下降操作，例如，可以首先下降 *A*，而后是 *B*，随后再次下降 *A*，依此类推。该规则易于实现，且一如同步遍历操作，此方案也无须对下降过程实施评估。
- ☑ 基于相交状态的下降操作。另一种选择方案是：对层次结构中交叠较多的部分优先执行下降操作，而后是交叠较少的部分。其核心思想可表述为：包围体间相交的可能性越大，其对应对象产生碰撞的几率也就越大。
- ☑ 采用上述组合方案，或基于遍历历史操作的其他组合规则。

通过炼油厂这个实例可知，高效的遍历方法将取决于相应的数据结构。下面将探讨基于深度优先遍历下降规则的框架实现方式。另外，对于较大的数据集合，采用混合算法（如树型网格）通常会比独立的树型层次结构更加高效。混合算法将在 7.3.8 节中加以讨论。

6.3.2 通用的启发式深度优先遍历

采用一类简单的、基于层次递归的程序即可实现大多数下降规则。首先，若顶部包围体并未相交，该程序则简单地返回；否则，若当前两节点皆为叶节点，则调用底层几何体碰撞检测例程。另外，计算过程中还应对下降规则实现相应的评估，且代码针对下降规则所选的子节点实现递归调用。代码如下所示[①]：

```
// Generic recursive BVH traversal code.
// Assumes that leaves too have BVs
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{
    if (!BVOverlap(a, b)) return;
    if (IsLeaf(a) && IsLeaf(b)) {
        // At leaf nodes. Perform collision tests on leaf node contents
        CollidePrimitives(r, a, b);
    } else {
        if (DescendA(a, b)) {
            BVHCollision(a->left, b);
            BVHCollision(a->right, b);
        } else {
            BVHCollision(a, b->left);
            BVHCollision(a, b->right);
        }
    }
}
```

其中，函数 **BVOverlap()**用于计算两个包围体之间的相交状态。若输入参数为叶节点而非内部节点，则函数 **IsLeaf()**返回真。函数 **CollidePrimitives()**则用于对所含图元执行碰撞测试，并将反馈的碰撞信息累计至当前 CollisionResult 结构中。

函数 **DescendA()**则实现了相应的下降规则，若对象层次结构 *A* 需相对于层次结构 *B* 执行相应的下降操作，则函数返回真，否则返回假。另外较为重要的一点是：该函数能够正确处理 *A* 中叶节点与 *B* 之间的关系，从而无须遍历至叶节点。这里，下降规则包括“下降节点 *A*”、“下降节点 *B*”以及“下降较大节点”几种。在该框架中，相关代码如下所示：

```
// 'Descend A' descent rule
bool DescendA(BVTree a, BVTree b)
{
    return !IsLeaf(a);
}

// 'Descend B' descent rule
bool DescendA(BVTree a, BVTree b)
{
```

[①] 此处的代码内容借鉴了[Gottschalk00]。

```
    return IsLeaf(b);
}

// 'Descend larger' descent rule
bool DescendA(BVTree a, BVTree b)
{
    return IsLeaf(b) || (!IsLeaf(a) && (SizeOfBV(a) >= SizeOfBV(b)));
}
```

虽然递归版本可读性强且易于理解，但还不是最有效的形式。存在一种迭代版本且显式地使用了栈式存储结构，从而可避免递归函数调用所产生的消耗。更为重要的是，若只是存在单一接触点，则该方案将使得代码提前退出。

需注意的是，在代码的递归版本中，当以某种顺序遍历树型结构时，须以相反的顺序入栈。迭代版本的实现代码如下所示：

```
// Non-recursive version
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{
    Stack s = NULL;
    Push(s, a, b);
    while (!IsEmpty(s)) {
        Pop(s, a, b);

        if (!BVOverlap(a, b)) continue;
        if (IsLeaf(a) && IsLeaf(b)) {
            // At leaf nodes. Perform collision tests on leaf node contents
            CollidePrimitives(r, a, b);
            // Could have an exit rule here (eg. exit on first hit)

        } else {
            if (DescendA(a, b)) {
                Push(s, a->right, b);
                Push(s, a->left, b);
            } else {
                Push(s, a, b->right);
                Push(s, a, b->left);
            }
        }
    }
}
```

其中，函数 **Push()**，**Pop()**和 **IsEmpty()**实现了一类抽象的栈式数据类型。在对非递归版本的相关流程加以研究后，可在节点对入栈过程中发现一些冗余操作：主循环的下一次迭代计算将即刻弹栈该节点对。对代码稍加改变且将入栈数据直接赋予相关变量即可避免这一类情况：

```
// Stack-use optimized, non-recursive version
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{
```

```
    Stack s = NULL;
    while (1) {
        if (BVOverlap(a, b)) {
            if (IsLeaf(a) && IsLeaf(b)) {
                // At leaf nodes. Perform collision tests on leaf node contents
                CollidePrimitives(r, a, b);
                // Could have an exit rule here (eg. exit on first hit)
            } else {
                if (DescendA(a, b)) {
                    Push(s, a->right, b);
                    a = a->left;
                    continue;
                } else {
                    Push(s, a, b->right);
                    b = b->left;
                    continue;
                }
            }
        }
        if (IsEmpty(s)) break;
        Pop(s, a, b);
}
```

按照这里给出的方式，递归遍历函数皆可转换为对应的迭代版本。

6.3.3 同步深度优先遍历

同步遍历无法通过上述框架直接加以实现。这里，由于两个包围体同时下降，两节点之间最多需要 4 次递归调用而非两次。同步遍历的代码实现如下所示：

```
// Recursive, simultaneous traversal
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{

    if (!BVOverlap(a, b)) return;
    if (IsLeaf(a)) {
        if (IsLeaf(b)) {
            // At leaf nodes. Perform collision tests on leaf node contents
            CollidePrimitives(r, a, b);
            // Could have an exit rule here (eg. exit on first hit)
        } else {
            BVHCollision(a, b->left);
            BVHCollision(a, b->right);
        }
    } else {
        if (IsLeaf(b)) {
            BVHCollision(a->left, b);
            BVHCollision(a->right, b);
```

```
        } else {
            BVHCollision(a->left, b->left);
            BVHCollision(a->left, b->right);
            BVHCollision(a->right, b->left);
            BVHCollision(a->right, b->right);
        }
    }
}
```

比较不同种类之间的操作数量可以得到一些有趣的结果。对于上述算法，当同步遍历下降并检查其中的 4 个节点组合时，将额外进行两次递归调用，且与前述有向算法相比，同步遍历的工作量只是其 2/3（尤其是全遍历时）。

特别地，考查下列最坏情形：分别包含 n 个叶节点的两棵完全二叉树处于某一相对位置，包围体叶节点存在相交，但结构间却并未发生碰撞。针对这种情况，同步遍历将执行$(2^{2(n-1)}-1)/3$ 次内部节点间的测试，以及 $2^{2(n-1)}$次叶节点-节点、节点-叶节点和叶节点-叶节点间的测试，因此累计为$(2^{2n-1})/3$ 次测试。

比较而言，有向叶节点算法（leaf-directed）中的下降规则将执行 $2^{n-1}-1$ 次内部节点-节点间的测试，以及$(2n-1)2^{n-1}$ 次叶节点-节点、节点-叶节点和叶节点-叶节点间的测试，因此累计为 $2^{2n-1}-1$ 次测试。当 n 趋于无限大时，其极限值即验证了“同步遍历的工作量只是有向算法的 2/3”这一结论。

但对于包含启发信息的遍历算法而言，则很难提供具体的数字——遍历模式完全依赖于所使用的下降规则。但可以肯定地讲，由于实现代码具有相同的结构且遍历路径固定，测试的全部次数将等同于有向叶节点遍历算法。但是，对于“下降较大节点”这一类情况，很难讲同步遍历算法会比有向算法更加节省资源，因为在该情形下，采用“最佳节点导向”这一方式实现了更加高效的搜索算法。

6.3.4　优化的有向叶节点深度优先遍历

同步启发式引导算法也存在一类缺陷，涉及同一叶节点的碰撞检测过程很可能随时间呈分布状态。如果测试过程包含某些叶节点数据的转换操作或叶节点的包围体，则查询时与叶节点相关的转换操作每次都需重复执行。如这类转换操作成本高昂，则有必要在初次转换完毕后对数据采取某种缓存机制。

对于这一类较为复杂的缓存方案，一种折中方法即是前面所提到的“下降节点 A”规则：首先将层次结构 A 降至叶节点，随后是层次结构 B——这将一次性地转换 A 中的叶节点数据。但该折中方案并非是一类放之四海皆准的缓存机制，且仅对层次结构 A 方可采取这种有效的缓存方式。鉴于该方案的简易性，它可作为一种反映缓存机制完善度的指标。相应的代码片段如下所示：

```
// This routine recurses over the first hierarchy, into its leaves.
// The leaves are transformed once, and then passed off along with
// second hierarchy to a support routine
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{
    if (!BVOverlap(a, b)) return;
    if (!IsLeaf(a)) {
        BVHCollision(a->left, b);
        BVHCollision(a->right, b);
```

```
        } else {
                a2 = TransformLeafContentsOnce(a);
                BVHCollision2(r, a2, b);
        }
}

// The support routine takes what is known to be a leaf and a full
// hierarchy, recursing over the hierarchy, performing the low-level
// leaf-leaf collision tests once the hierarchy leaves are reached
void BVHCollision2(CollisionResult *r, BVTree a, BVTree b)
{
        if (!BVOverlap(a, b)) return;
        if (!IsLeaf(b)) {
                BVHCollision2(a, b->left);
                BVHCollision2(a, b->right);
        } else {
                // At leaf nodes. Perform collision tests on leaf node contents
                CollidePrimitives(r, a, b);
        }
}
```

6.4 包围体层次结构示例

本章中所讨论的层次构造算法适用于任意包围体类型，为了进一部说明其使用状况，本节将详细阐述相关文献中所推荐的以及真实系统所使用的某些特定算法。需要注意的是，相关算法并非是结论性的技术方案，这里只是体现某种具体实现方式。

6.4.1 OBB TreesOBB 树

OBB 树是一类自顶向下的构造方案，可参见[Gottschalk96]。假定初始状态下存在一个与原图元结合紧密配合的 OBB，其中，对图元中的全部面执行相关计算将得到连续的协方差公式，并可由此获取与盒体各轴相关的特征向量，与盒体各轴实现对齐将可获得紧密配合的 OBB，可参见 4.4.3 节。给定该盒体后，数据集合将在 OBB 最大轴向上进行分割以实现划分操作。在基于协方差的 OBB 中，该轴可视为最大分布轴。另外，对象均值（源于图元顶点在各轴上的投影）在这里可用作分割点。

对于跨越分割面的图元，可将其归于质心坐标所处的半空间子集中。此处，若在最大轴上无法生成两个非空半空间子集，可以按长度降序方式尝试其他轴。如果全部轴均失效，该数据集将被视为无效集合。然而，在目前的已知算法实现中，如果初始划分失效且不采取替换轴的方式，数据集将只是依据对象中值简单地划分为两个均等的子集。

当两个面平行且十分接近时，OBB 树能够较为完美地处理这一情况。另外，OBB 层次结构将呈现二次收敛以匹配几何体，AABB 和球体仅是线性收敛。亦即，若 OBB（$O(m)$）须在给定的范围值 $O(m^2)$ 内逼近几何体，则在同一任务中，该条件对 AABB 或球体也是必不可少的。最后，与 AABB 或球体树

相比，OBB 树性能函数中的 N_V 值和 N_P 值将趋于更小。然而，与 AABB 或球体间的相交测试相比，两个 OBB 间的相交测试性能仍然慢一个量级。

6.4.2　AABB 树和盒体树

在[Bergen97]中，作者利用自顶向下的方案阐述了 AABB 二叉树的构造过程。在每一步构建过程中，都采用了 AABB 对图元集合实现紧密包围。通过沿最大面方向上分割 AABB，图元集合将被划分为两个子集。同时，根据投影中点相对于分割面的最终位置，图元将被划分至相应的子集中。这种利用分割面限定图元的划分方案将子集中 AABB 相交测试数量降至最低。

上述递归构造过程重复进行，直至子集中只包含一个图元。另外，可将分割点视为空间中值，继而实现 AABB 的划分。van den Bergen 曾使用该算法与对象中值分割算法进行了比较，结论为：空间中值算法性能占优。在某些特殊的情况下，例如，全部图元皆位于某一子集中，此时集合可采用基于对象中值的相关算法加以划分。

具有启发信息的遍历算法常与“下降最大节点”规则联合使用，从而实现树的遍历。此处，当对象旋转时并不重对齐其 AABB 时，OBB 测试只是比较转换后的相应有向 AABB。由于同一相对方向共享于转换后的 AABB 组合，因此转换矩阵在每次查询操作时只需计算一次，从而简化了 OBB 测试。

van den Bergen 还指出，对于旋转后的 AABB 测试，只执行 15 个轴测试中的前 6 个测试将会进一步提高计算速度。当然，这一类折中方案会全面降低测试消耗，但这也会导致无效图元碰撞测试数量（消耗）的增加。同时，该优化方案对于 OBB 树亦是如此。

盒体树也是一类由对象坐标系统中定义的 AABB 采用递归方式加以构造的层次结构，参见[Zachmann95]。此处，分割面将包围体分割为（无须均等的）两部分，其结果子集将直接形成多个 AABB。因而，子盒体中的图元将被划分至对应的集合中。另外，跨越分割面的相应图元将分别划分至两个盒体中——这将导致图元间的复制操作。总体来讲，盒体树的构造方案与 *k*-d 树十分类似。

由于跨越分割面的图元将被复制，因此在选择分割面时需综合考虑树的平衡性与减少跨越图元数量之间的关系。在计算分割面之前，Zachmann 首先考查了 AABB 中是否可剔除某些较大的无效空置空间，从而使其更好地包围几何体。此处，“较大”定义为：空置盒体与其父节点盒体之比率大于某一预设常数。该操作须在 AABB 全部 3 个轴向上进行测试，并选择一个较优的计算结果。最后，递归操作的结束点可分为以下几种情况：遍历过程达到某一既定深度；集合中的图元数量小于某一特定极限值；子盒体中包含的图元数量接近于其父盒体。

由于盒体树构造于模型空间中，因此测试过程中须执行对象空间的转换操作。然而，由于 AABB 间将共享其顶部节点的方向数据，因而在层次结构遍历过程中，多数计算可得到复用。当测试旋转 AABB（现为 OBB）时，可采用分离轴测试方案，且相应的 15 个轴在测试过程中将保持恒定——这将极大地简化相关计算。另外，Zachmann 还提出了另一种基于剪裁的相交测试方案，且在树型结构的遍历过程中，Zachmann 使用了同步遍历算法。

在随后的描述中，Zachmann 并未使用分割面确定子盒体的边界划分，而是采用了两个平面沿分割轴两侧分别包围该跨越图元。若分割始于两平面中的最右平面，则“左”子盒体为父 AABB 的左部；对于“右”子盒体，同理可得。换句话讲，除了其中的一个面，子 AABB 将等同于父 AABB。采用这种表达方式，在给定了父体空间后，子空间可仅使用两个（浮点）值加以表示，参见[Zachmann00]。

van den Bergen 算法和 Zachmann 算法可应用于任意多面体以及多边形汤之上，但其可靠性仅体现于表面碰撞检测中，同时，算法并不检测“一个对象完全包含另一对象”这一状况。

6.4.3 采用 8 叉树子划分的球体树

出于简洁性，一类常见的算法是利用对象空间的 8 叉子划分构造球体树，参见[Tzafestas96]。首先，可使用一个较为紧凑的包围立方体封闭当前对象，而后，该立方体将被递归地划分为 8 个尺寸均等的组成部分，或称作 8 分体。包含对象各组成部分的每一个 8 分体将再次进行递归划分，直至到达某一既定树深度值。在相关构造过程中，生成的各个 8 叉树节点将采用一个外接球体以构造最终的球体树。同时，8 分体中心点将视为球心。对于包含 s 个面的规则立方体，球体半径为 $r = s\sqrt{3}/2$；若初始包围体为一般 AABB 而非立方体，则包含 x、y、z 向各面的 8 分体半径为 $r=\sqrt{x^2+y^2+z^2}/2$。

一些冗余球体的存在将会影响碰撞检测的计算，为了剔除它们，递归过程将会停止于全遮挡的 8 叉树节点，参见[O'Sullivan99]。因此，这将导致碰撞检测过程包含未经测试的内部节点。

采用上述算法生成的球体树一般都处于松散匹配状态。Hubbard 建议，可采用模拟退火算法实现对象球体的紧密配合，但仍保持对象的保守覆盖方式，参见[Hubbard95]。算法将对球体进行随机选择，并接受更为紧凑的球体，有时也会接受低于局部最小值的球体。但正如 Hubbard 指出的那样，测量物体包围球的紧密度并不是一件易事。另外，模拟退火算法的计算速度十分缓慢，且球体间的分布状态也极不规则。因此，该算法的实际应用价值十分有限。

6.4.4 采用球体覆盖表面的球体树

当利用自顶向下的基础算法构造树型结构时，[Quinlan94]采用覆盖原对象表面的球体输入集合计算并生成了球体树。此处，各输入多边形将被规则的球体网格所覆盖，且球心位于多边形平面上。据此，覆盖表面以一种类似于扫描转换的方式加以处理。最终，各球体将构成结果树中的相应叶节点，且使用其对应的覆盖多边形加以标记。

将叶节点球体集合划分为两个大小（约）相等的两个子集，可以实现层次结构的构造过程，且针对每一个子集递归地调用该算法，即可建立两棵子树。随后，将子树作为子节点添加至新的父节点中，即完成了子树至树的整合过程。在每一步中，可使用一个 AABB 包围球心，以将球体集划分为两个子集。随后，AABB 将沿其最大面方向划分为两部分，基于球心与两个 AABB 间的相对位置，球体被划分至相应的子集中。

需要注意的是，上述算法假定表面为凸多边形，且并不检测对象的内部区域。

6.4.5 生成-修剪球体覆盖

这里，算法本身并不构造相应的层次结构，而只是简单地计算球体集合，Greenspan 和 Burtnyk 给出了完全包围给定对象的球体并集的计算方法，参见[Greenspan96]。该算法主要包含两步，首先执行生成步骤：对象被多个球体完全覆盖；其次执行修剪步骤，即移除冗余球体。此处，“冗余”是指：若球体被移除，仍存在其他球体完全覆盖当前对象，则称该球体为冗余球体。

算法的生成部分由两个参数控制。第一个参数称为“过调量（overshoot）”，该参数用于控制球体可超出对象表面的扩展量；第二个参数为间距，定义为两个球心之间的最小距离。算法将围绕对象生成一个紧凑而划一的 3D 网格，且网格单元边设置为上述间距参数。根据网格单元相对于对象的位置——完全位于对象的内部、完全位于对象的外部、位于对象表面——可标记为“内部”、“外部”以及“表面”，且在初始状态下，各网格单元赋值为 0。在未曾标记为“外部”的网格单元处，居中设定一个球体。此处，球体半径设定为大于“过调量”的某一值，即球体未超出对象边界。

在修剪步骤中，无论是被单个（较大）球体包围还是被多个球体集合的并集包围，都无法避免冗余球体的出现。对于前者，当前各球体将与其他球体进行比较以查看冗余性：若是，则移除该球体。测试过程需要 $O(n^2)$次球体-球体间的比较。对于后者，需要对剩余球体集合进行迭代计算，且针对每一个完全被球体包含的网格单元，其关联值须相应地加 1。当该过程处理完毕后，标有“表面”特征值且值为 1 的网格单元即被单个球体所包含。对于被完全包围的对象而言，所有该类球体将置入一个结果集中。因此，球体被识别、移除并加入相应的结果集中。最后，网格单元标记为“已处理”且无须再次执行计算。

当所有该类球体已添加至结果集中，且不存在值为 1 的网格单元时，剩余表面网格单元即被多个球体所覆盖。反过来讲，每一个剩余球体也必定被多个球体所包围。因此，当球体仍保持对对象的全覆盖时，即可从候选集中删除该球体。在删除该球体后，其中的网格单元关联值须相应地减 1。当出现值为 1 的网格单元时，对应球体可按前述方式添加至结果集中。当候选球体已全部处理完毕（或简单地讲，全部表面网格单元都已标记为“已处理”）时，相应的处理过程即可结束。最后，即使算法最终未能从球体结果集中构造相应的层次结构，但仍可将其作为球体输入集，并利用前面所讨论的任何层次结构构造算法生成相应的球体树。

6.4.6　*k*-DOP 树

在[Klosowski98]中，*k*-DOP 树用于计算飞行物体与大型静态场景之间的碰撞检测。采用自顶向下的构造方法，算法将三角形集合包围至一个 *k*-DOP 中，并将该输入集合划分为两个子集，且针对两个子集执行递归划分。最终，当节点中包含的三角形数量达到一个既定阀值 t 时，递归过程结束。此处，对于静态层次结构，t=1；对于飞行物体，t=40。查询遍历过程按如下方式进行：首先对场景结构执行全下降操作，随后进行物体对象的下降操作。

针对 *k*-DOP 树的构造过程，存在 4 种不同的划分策略可供比较。依据不同的标准，轴选取方案将仅限于坐标轴 x 轴、y 轴或 z 轴中的一个。相关标准分别是：最小化体包围体之和；最小化最大包围体；采用具有最大方差的轴；沿父包围体最大轴进行划分。另外，在计算分割点时，分别使用了轴向投影三角形质心坐标的均值和中值。

结果表明，与中值相比，在均值处进行划分通常能够生成较小的包围体。“最小化体包围体之和”策略总是可以产生最小的包围体层次结构。相比较而言，采用“最小化最大包围体”策略、“具有最大方差的轴”策略、“沿父包围体最大轴进行划分”策略将分别增大 7%、10%、33%。[Klosowski98]针对多于 3 个坐标轴这一状况进行了相关测试，特别地，使用了 k-DOP 中的全部 $k/2$ 个定义的方向轴。但计算结果显示，这并没有取得明显的改善效果，反而使平均预处理时间增加了 30%。

通常，根节点及其附近节点被访问的频率较高，算法针对根节点采用了较为耗时的爬山算法，旨

在更新过程中实现 k-DOP 的翻转（tumble）操作。对于其他节点，可以采用消耗较低的近似“DOP-of-DOP”算法。

在采用 k-DOP 进行协同计算时，[Konečný98]将图元划分为两个均等的集合，且并未使用该 k-DOP 内、$k/2$ 个定向轴集合中具有最大方差的相应轴（通过投影质心坐标计算得到）。

[Klosowski98]分别比较了 6-DOP、14-DOP、18-DOP 以及 26-DOP，并发现 18-DOP 具有最佳执行效果。作为对比，[Zachmann98]则比较了 6-DOP、8-DOP 和 14-DOP 且结论为：6-DOP（即 AABB）将优于其他两种 DOP。在后续计算中，Zachmann 使用了第 4 章所讨论的非标准 k-DOP，并对 $k = [6\cdots32]$ 范围内的 k-DOP 加以考查。在全部测试中，虽然不存在最优的 k 值，但总体而言，当 $k = 24$ 时其对应的 k-DOP 将具有较好的性能（[Zachmann00]）。由于测试是在不同的条件下完成的，因而很难对计算结果进行直接比较。

当顶点各组成成分采用 double 类型（8 个字节），且三角形向量索引采用 4 字节整数类型时，Klosowski 等人指出：共需$(16k+108)n$ 个字节用于存储场景中的所有 n 个三角形，且包含了该层次结构自身。当 k=6、14、18 和 26 时，则针对每一个输入三角形，分别需要 204、332、396 和 524 个字节。比较一下，基于 OBB 的碰撞检测包 RAPID（也采用了 double 类型），每一个输入三角形则使用了 412 个字节。更为糟糕的是，对于一个包含 169944 个三角形的数据集，若采用 26-DOP，其尺寸将达到 69MB。

6.5 合并包围体

无论层次结构采用自顶向下方法还是渐进方式加以构造，其包围体通常都需要依据数据的相应维度从头创建。在自底向上构造方法中，针对开销较大的包围体-数据直接匹配操作，一类可替代的方案是将子包围体合并为一个新的包围体。本节将阐述 AABB、OBB 以及 k-DOP 的合并操作。

6.5.1 合并两个 AABB

计算两个 AABB 合并后的 AABB 十分简单：合并后的 AABB 其各面的选取应与两个给定 AABB 的最小值面和最大值面吻合。例如，若 AABB 采用最小值-最大值表达方式，合并后的 AABB 的计算过程如下列代码所示：

```
// Computes the AABB a of AABBs a0 and a1
void AABBEnclosingAABBs(AABB &a, AABB a0, AABB a1)
{
    for (int i = 0; i   <    2; i++) {
        a.min[i] = Min(a0.min[i], a1.min[i]);
        a.max[i] = Max(a0.max[i], a1.max[i]);
    }
}
```

对于采用其他表达方式的 AABB，其合并计算亦并非难事。

6.5.2　合并两个球体

在计算两个球体的最小包围球时，最好全面考察下列两种情形：一个球体被另一个球体完全包围；两球体分离或部分相交。为了区分不同情形，定义两个球体为 S_0 和 S_1，球心坐标为 C_0 和 C_1，球体半径分别为 r_0 和 r_1（如图 6.5 所示），$d = \|C_1 - C_0\|$ 且记为 C_0 和 C_1 之间的距离。于是，若 $|r_1 - r_0| \geq d$，则其中一个球体将完全被另一个球体包围。对此，稍加分析即可得出相关结论。首先，假定两个球体同心，则有 $d = 0$ 且两球体表面间的距离为 $|r_1 - r_0|$。内置球体可逐渐穿越其外部球体，即两球心分离（d 值增加）。

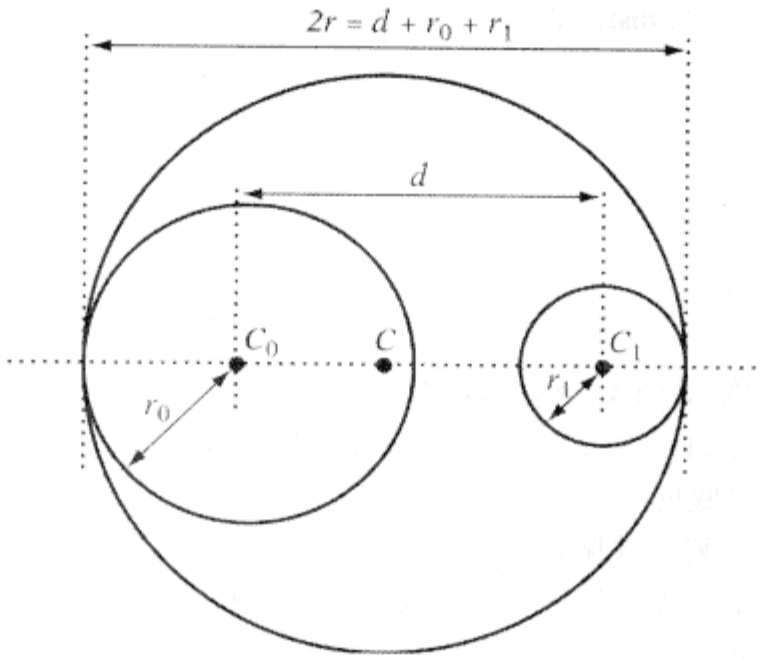


图 6.5　合并球体 S_0 和 S_1

在第一种情况下，即一球体内嵌于另一球体中，无须计算并生成新的球体——只需返回较大球体即可。同时，这也避免了开销较大的平方根运算。对于第二种情况，即球体间部分相交或完全分离，新球体的半径 r 为两球体表面间最大距离的一半：

$$r = (d + r_0 + r_1)/2$$

球心 C 可按下列方式加以计算：从 C_0 处背向 C_1 移动 r_0 个单位，再向 C_1 移动 r 个单位：

$$\begin{aligned} C &= C_0 - r_0(C_1 - C_0)/\|C_1 - C_0\| + r(C_1 - C_0)/\|C_1 - C_0\| \\ &= C_0 + (r - r_0)(C_1 - C_0)/\|C_1 - C_0\| \end{aligned}$$

代码如下所示：

```
// Computes the bounding sphere s of spheres s0 and s1
void SphereEnclosingSpheres(Sphere &s, Sphere s0, Sphere s1)
{
    // Compute the squared distance between the sphere centers
    Vector d = s1.c - s0.c;
    float dist2 = Dot(d, d);

    if (Sqr(s1.r - s0.r) >= dist2) {
        // The sphere with the larger radius encloses the other;
```

```
            // just set s to be the larger of the two spheres
            if (s1.r >= s0.r)
                s = s1;
            else
                s = s0;
        } else {
            // Spheres partially overlapping or disjoint
            float dist = Sqrt(dist2);
            s.r = (dist + s0.r + s1.r) * 0.5f;
            s.c = s0.c;
            if (dist > EPSILON)
                s.c += ((s.r - s0.r) / dist) * d;
        }
}
```

6.5.3 合并两个 OBB

合并两个 OBB 则稍显复杂。一种较为直接的方法是采用第 4 章所使用的技术。例如，可将两个盒体的全部 16 个顶点直接传递给函数 **ComputeOBB()**（针对各顶点，采用了协方差矩阵的特征向量作为盒体的数据轴），且可以与 4.4.4 节中的“最小主成分分析”技术结合使用。后者在[Barequet96]所描述的盒体树算法中，则采用了自底向上的构造算法。

但是这一类技术往往存在某些缺陷，即计算特征向量的相关迭代算法其开销一般都较大，因而并不适用于实时计算中。一种非迭代的替代算法可参见[Eberly00]，并建议组合两个 OBB 间的旋转操作，将其转换至四元数并实现二者的插值计算。最终，将结果再转换回矩阵，从而得到 OBB 中的各轴，OBB 的相应范围值则可通过轴上的投影得到。需要注意的是，在[Eberly00]给出的伪代码中，将首先计算 OBB 的中心坐标，随后才确定 OBB 的范围值。一般情况下，这并不能生成较为紧凑的 OBB。为此，中心坐标应在范围值之后进行计算。

6.5.4 合并两个 *k*-DOP

如同合并 AABB，*k*-DOP 的合并计算也十分简单。针对每一个 *k*-DOP 轴，结果包围体 *k*-DOP 按如下方式定义：选择两个 *k*-DOP 各自较小值中最小的一个，且选择各自较大值中最大的一个。其实现过程十分类似于 AABB 的合并代码：

```
// Computes the KDOP d of KDOPs d0 and d1
void KDOPEnclosingKDOPs(KDOP &d, KDOP d0, KDOP d1, int k)
{
    for (int i = 0; i < k / 2; i++) {
        d.min[i] = Min(d0.min[i], d1.min[i]);
        d.max[i] = Max(d0.max[i], d1.max[i]);
    }
}
```

6.6　高效的树型表达方式及遍历

到目前为止，关于层次结构的构造及遍历的讲解还是较为抽象的。然而，要达到业界的实现标准，对遍历代码以及树型的表达方式进行优化就变得十分重要。由于在现代计算机体系结构中，内存访问以及分支预测失效是导致计算瓶颈的主要原因，因而存在两类相关的优化方案，分别实现了数据结构的最小化以及缓存友好的数据排列方式，从而加快了相应数据的访问速度。

本节将阐述几种数据的表现方式以及遍历的优化方案，以加速碰撞查询的计算过程。然而，由于缓存方式的难以预测性，某些较为高级技术并不能体现预期的加速结果。如果遇到此类情况，尽量保持遍历代码短小精悍、简单易懂往往是实现加速计算的简单方式。高效的树型表达方法还将在第 13 章中结合内存优化技术加以讨论，

6.6.1　数组表达方式

这里定义一个包含 n 个节点的完全二叉树作为碰撞层次结构。通过逐级广度优先的方式将节点映射至 n 个元素的数组中，以实现树型结构的存储。

```
// First level
array[0] = *(root);
// Second level
array[1] = *(root->left);
array[2] = *(root->right);
// Third level
array[3] = *(root->left->left);
...
```

当映射过程建立完毕后，针对位于 **array[i]**的节点，其对应的左孩子节点为 **array[2*i+1]**，右子节点为 **array[2*i+2]**。这里，摒弃了指向左右孩子节点的指针，并将这一类“无指针”节点存储于数组中（如图 6.6 所示）。在了解了子节点-索引规则后，最终映射过程如下列递归函数所示：

```
// Given a tree t, outputs its nodes in breadth-first traversal order
// into the node array n. Call with i = 0
void BreadthFirstOrderOutput(Tree *t, Tree n[], int i)
{
    // Copy over contents from tree node to breadth-first tree
    n[i].nodeData = t->nodeData;
    // If tree has a left node, copy its subtree recursively
    if (t->left)
        BreadthFirstOrderOutput(t->left, n, 2 * i + 1);
    // Ditto if it has a right subtree
    if (t->right)
        BreadthFirstOrderOutput(t->right, n, 2 * i + 2);
}
```

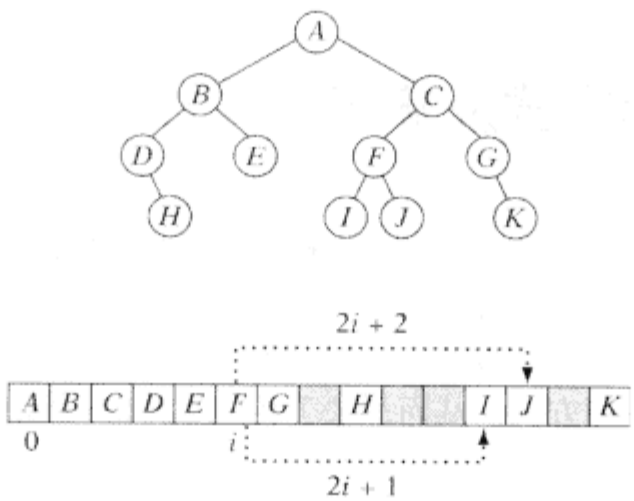


图 6.6　采用数组格式（下图）存储的二叉树（上图）。对于位置为 i 的数组节点而言，其子节点的位置分别为 $2i+1$ 和 $2i+2$。应注意的是，该方式较为浪费内存空间（采用灰色标识）

在较好地实现了平衡树后（符合完全树的标准），将进一步节省内存空间且遍历过程无须执行指针的读取。但对于非平衡树而言，仍然需要按照完全平衡树的格式分配额外的节点空间。更加糟糕的是，向完全平衡树加入单一节点，为了保持树型的完全平衡性，需要加倍分配相应的存储空间。

当节点信息（即有效负载数据）小于其子节点指针数据尺寸，或树型结构接近完全平衡时，上述表达方法十分有效。例如，当采用中值分割策略构建层次结构，且能够在一定程度上保证树型结构接近平衡时，该方案将是一类有效的表达方式。

6.6.2　前序遍历

当无法保证树型结构的平衡性时，仍然可以采取一类有效的表示法输出相关数据。如果树节点采用前序遍历，左子节点总是出现在其父节点之后。通过这种方式，虽然仍保留指向右子节点的链接，但只需一位用于指明是否存在左子节点（紧随当前节点）。图 6.7 显示了一棵二叉树并以前序遍历方式输出数据。

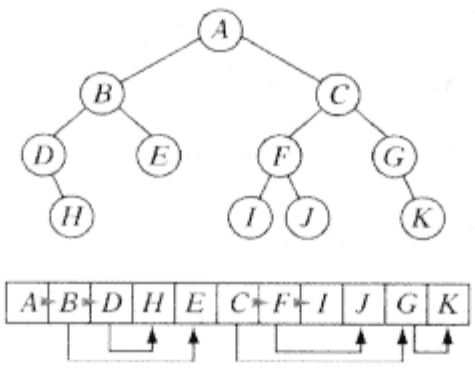


图 6.7　针对图 6.6 中的树型结构，采用前序输出节点。这里，节点将包含一个指向其右子节点的指针（显示为箭头）。同时，节点还需要一个位数据以标识是否存在左子节点（并紧随父节点）。其中，该数据以灰色三角形箭头表示

对于采用基于指针的树，当以前序遍历的方式将数据输出至数组中时，其实现函数也比较简单。唯一复杂之处在于：当父节点写入内存且更新右链接指针时，其指向的节点位置未知。这可以通过二

次遍历树型结构或采取（较好的）栈式回填技术加以解决，代码如下所示：

```
// Given a tree t, outputs its nodes in preorder traversal order
// into the node array n. Call with i = 0.
int PreorderOutput(Tree *t, Tree n[], int i)
{
    // Implement a simple stack of parent nodes.
    // Note that the stack pointer 'sp' is automatically reset between calls
    const int STACK_SIZE = 100;
    static int parentStack[STACK_SIZE];
    static int sp = 0;

    // Copy over contents from tree node to PTO tree
    n[i].nodeData = t->nodeData;
    // Set the flag indicating whether there is a left child
    n[i].hasLeft = t->left != NULL;
    // If node has right child, push its index for backpatching
    if (t->right) {
        assert(sp < STACK_SIZE);
        parentStack[sp++] = i;
    }
    // Now recurse over left part of tree
    if (t->left)
        i = PreorderOutput(t->left, n, i + 1);
    if (t->right) {
        // Backpatch right-link of parent to point to this node
        int p = parentStack[--sp];
        n[p].rightPtr = &n[i + 1];
        // Recurse over right part of tree
        i = PreorderOutput(t->right, n, i + 1);
    }
    // Return the updated array index on exit
    return i;
}
```

除了将子节点中的指针数量减少为一半，该表达方式还易于实现缓存技术。其中，左子节点可能已采取缓存处理且随其父节点同步读取——这将使遍历过程更加高效。

6.6.3　采用偏移量而非指针

一类典型的树型实现方式是采用（32 位）指针表示节点的子链接。然而，过度使用指针将导致严重的后果。通常，当采取数组的方式分配树型节点时，可使用一个 16 位的索引值代表距数组起始处的偏移量。该方法对于静态树和动态树都是有效的。针对静态树，通过定义相对于父节点的偏移量，节点的表示范围将更加广泛。

考虑到遍历时的一些额外开销，可以组合使用指针和索引。这里，可定义一个保留位并将其值表示为相对于独立指针数组或偏移量数组的索引。

6.6.4 采用缓存友好的结构（非二叉树）

在现代计算机体系结构中，执行时间通常受限于与内存访问数据相关的缓存机制，而非所执行指令的数量。因此，采用某些“非常规”的数据结构反而可减少相应的内存占有量，并实现较为友好的缓存访问机制——虽然这会使遍历过程复杂化。

一类可行的表现方式是将二叉树中的 3 个节点（父节点、左子节点和右子节点）合并为融合了 3 个节点的单一节点。此处，原 3 节点集合中包含了两个内部链接（用于连接子节点的父节点），以及 4 个外部链接（用于连接树型结构的其他部分）。而新节点不需任何内部节点，且只保留 4 个外部链接。

对于一棵包含 4 层（15 个节点）的完全二叉树而言，则需要 14 个内部连接。而其对应的 3 节点树则只包含两层（4 个节点）以及 4 个内部链接（如图 6.8 所示）。对于一棵包含 6 层（63 个节点）的二叉树，则需要 62 个内部链接，而 3 节点树只需 20 个节点。总体而言，3 节点树的链接数量约为对应完全二叉树链接数量的 1/3，其缩减量甚至优于前序遍历。另外，3 节点树可采用广度优先的方式加以输出，且可对层次结构中的前 n 层数据实现缓存。

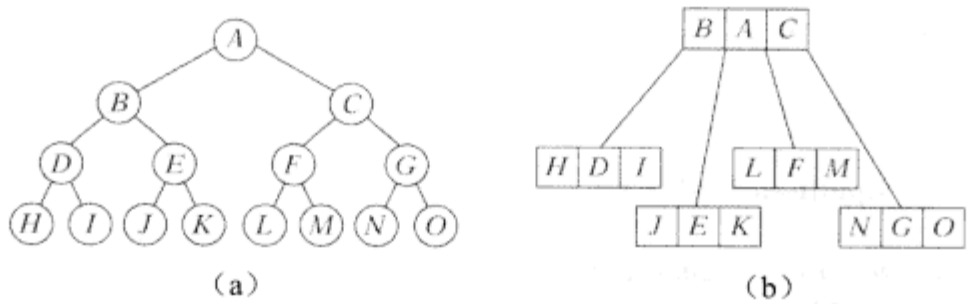


图 6.8 （a）4 层二叉树。（b）对应的 2 层 3 节点树

3 节点树的缺陷是需要在遍历树型结构时采取某些额外的处理。另外，当 3 节点树不是完全树时，3 节点中将出现某些空节点，从而造成内存的浪费。因此，需要增加一个标志变量以表示节点是否已用还是为空。由此可知，3 节点树更适用于密集树型结构；而对于稀疏树型结构，则优先采用前序遍历。

当然，可将该树型结构与前述相关算法结合使用。例如，与采用子节点指针相比，前序存储 3 节点树并使用相对于父节点的偏移量能够将链接所需内存量降至最低。相关技术也曾出现在[Intel99]中，由于 3 个球体可以很好地在缓存单元（cache line）中实现协调工作，算法使用了 3 个球体的并集而非一个球体作为包围体。

6.6.5 树节点和图元排序

6.3.3 节曾讨论了同步遍历算法，且遍历过程中节点间出现了 4 次递归调用。如算法中所示，调用顺序完全取决于树中节点间的相对顺序。若当前操作仅为简单的相交测试而非计算碰撞信息，抑或利用距离值来剔除层次结构中的相应分支，则计算过程应尽早提供相交测试结果。

一种可能性是尝试在运行期内指定最佳顺序，从而安排较为合理的递归调用。但这实现起来还是有些难度的，因为针对于结构层次的当前位置，其下方结构仍未可知。一类可行的方案是（且不存在运行开销）：在构建过程中启发式地重置父节点的子节点，并尽早提供可能出现相交的子节点。依据

[Haines91a]，可能的节点排序方案包括以下几类：

- ☑ 按照叶节点优先的方式（相对于其他节点）排列子节点。如碰撞结果取决于图元所处的层次，且由于全部图元皆包含于叶节点中，则令叶节点在前、非叶节点在后，从而尽可能提前图元的测试顺序。
- ☑ 前置最小子树。同理，可将相应节点视为层次结构中子树的根节点。通过前置包含若干子孙节点的子树根节点，也可提前访问叶节点。
- ☑ 根据命中测试（hits-to-tests）比率排序子节点。在具体应用中，可按此方案重构层次结构，将发生正碰撞的节点前置，并后置从不产生碰撞或很少产生碰撞的节点。
- ☑ 沿主查询方向由近到远排序子节点。如果查询计算趋于某一特定方向，例如，对于一个表现地表的层次结构，可对节点实施排序并前置海拔较高的地表节点。

该排序方案也可应用于叶节点中的图元之上。当叶节点中包含多个图元时，图元可依据彼此的相似性进行排序。例如，若包含多种类型的图元：三角形、四边形、Bezier 面以及 NURBS 面，可首先对多边形进行测试，随后再执行开销较大的表面图元测试。

另外，也可令图元包含某些额外的剔除信息。例如，可通过一组代表静态场景的多边形集合构建一棵 OBB 树，并与上述图元进行碰撞计算。通常，该树叶节点 OBB 中不太可能全部是“陡峭”多边形。因此，可重新排列多边形并在多边形表中前置排列包含较高海拔顶点的多边形。随后，即可重组多边形的顶点并在顶点表中前置具有较高海拔的顶点。当前，可通过法线以及叶节点（处于已知位置）中第一个顶点定义一个平面，且该平面可用于快速剔除与 OBB 间的碰撞，而后，方可执行开销较大的其他测试。

其他类型的重构方式还包括：排序顶点并前置可构成最大边的两个顶点；设定顶点并结合多边形质心坐标（可利用存储顶点加以计算）构造多边形的包围球（球心的质心坐标和顶点定义了球体半径）。

6.6.6　递归遍历

对于树型结构（二叉树或其他类型的树型结构）可以很自然地采用递归方式加以遍历。然而，正如 6.3.2 节中指出的，递归并非是高效的代码实现方式，且在不同程度上存在某种缺陷。多数语言在递归退出时都无法直接返回一个值。虽然可添加一个标志变量并通知代码何时退出，但递归仍需要返回至顶部调用函数。

较为理想的解决方案是采用栈式存储，如 6.3.2 节所示。由于代码不再以递归方式执行，因而函数可以在任一时刻退出。另外，在保存了某些变量值后，代码可在任意点实现中断/恢复操作。显式地使用栈指针还可以方便地检测到遍历代码中的某些错误（如无穷递归）。同时，栈指针可隐式地计算栈深度，并将遍历过程限制在某一最大深度值处。

这里不使用递归还可以避免调用递归函数时所产生的开销。由于针对局部变量一般只生成一个栈，这既可占用较少的栈内存空间，又可有效地减少数据缓存垃圾的生成。

尽管递归方式存在诸多缺陷，但在开发前期阶段，采用具有良好可读性的递归代码将有助于快速建立原型并实现敏捷测试。另外，在 C 和 C++语言中，都存在相应的非递归解决方案使得代码能够从递归测试中提前结束。在 C 语言中，可选择使用库函数 **setjmp()**退出递归调用。针对以递归方式执行的通用启发式深度优先遍历算法，其查询版本的代码如下所示：

```
#include <setjmp.h>
jmp_buf gJmpBuf;

int BVHTestCollision(BVTree a, BVTree b)
{
    int r = setjmp(gJmpBuf);
    if (r == 0) {
    BVHTestCollisionR(a, b);
    return 0;
    } else return r - 1;
}
// Generic recursive BVH traversal code
// assumes that leaves too have BVs
void BVHTestCollisionR(BVTree a, BVTree b)
{
        if (!BVOverlap(a, b))
            longjmp(gJmpBuf, 0 + 1);     /* false */
        if (IsLeaf(a) && IsLeaf(b)) {
            if (PrimitivesOverlap(a, b))
                longjmp(gJmpBuf, 1 + 1);  /* true */
    } else {
            if (DescendA(a, b)) {
                    BVHTestCollisionR(a->left, b);
                    BVHTestCollisionR(a->right, b);
        } else {
            BVHTestCollisionR(a, b->left);
            BVHTestCollisionR(a, b->right);
        }
    }
}
```

需要注意的是，由于 **longjmp()**函数不存在 0 返回值，因而无法获取布尔值为 false 的返回值。对此，可以将返回结果加 1，而后在主程序中将其减 1。另外，在 C++语言中，将不再使用 **setjmp()**函数，较好的选择方案是采用异常处理机制。

6.6.7 分组查询

在游戏中，同一区域内的碰撞测试是十分常见的，例如车轮与地面之间的碰撞测试。理想情况下，这类查询可预置于相应的层次结构中，并且可采用前述任一算法对其实施遍历。然而在某些情况下，预构造层次结构并不可行。在汽车这个例子中——对于即将爆炸的汽车，其外形将发生巨大的变化。

当前，若针对车轮执行独立碰撞测试，由于汽车远小于世界场景，因此针对层次结构顶部的查询遍历，二者基本相同。因此，这将有助于在层次结构中实现集中查询，直至查询操作进入代表不同组件的树型分支处为止。

一类解决方案是针对查询对象动态地计算层次结构。但对于少量查询对象（如上述的 4 个车轮），这并不是必需的。这一类对象可组合至单一包围体中。该包围体将沿其测试的层次结构下移，直至到

达相应的节点（与其子节点相交）。此时，针对对象与场景层次结构之间的碰撞查询，当前节点可有效地替代根节点。这里，假定独立查询为球体与场景层次结构之间的测试，如下所示：

```
Sphere s[NUM_SPHERES];
...
for (int i = 0; i < NUM_SPHERES; i++)
    if (SphereTreeCollision(s[i], worldHierarchy))
        ...
```

其中，函数 **SphereTreeCollision()**在测试时可对输入的首节点进行替换，如下所示：

```
bool SphereTreeCollision(Sphere s, Tree *root)
{
    // If an alternative start node has been set, use it;
    // if not, use the provided start root node
    if (gStartNode != NULL) root = gStartNode;

    ...original code goes here...
}
```

首节点的设定/重设过程如下所示：

```
// Specifies an alternative starting node, or none (if null)
Tree *gStartNode = NULL;

// Set a new alternative start node
void BeginGroupedQueryTestVolume(Sphere *s, Tree *root)
{
    // Descend into the hierarchy as long as the given sphere
    // overlaps either child bounding sphere (but not both)
    while (root != NULL) {
        bool OverlapsLeft = root->left && SphereOverlapTest(s, root->left.bvSphere);
        bool OverlapsRight = root->right && SphereOverlapTest(s, root->right.bvSphere);
        if (OverlapsLeft && !OverlapsRight) root = root->left;
        else if (!OverlapsLeft && OverlapsRight) root = root->right;
        else break;
    }
    // Set this as the new alternative starting node
    gStartNode = root;
}

// Reset the alternative start node
void EndGroupedQueryTestVolume(void)
{
    gStartNode = NULL;
}
```

因此，上述代码可方便地转换为分组查询，从而节省了遍历时间，如下所示：

```
Sphere s[NUM_SPHERES];
```

```
...
// Compute a bounding sphere for the query spheres
Sphere bs = BoundingSphere(&s[0], NUM_SPHERES);
BeginGroupedQueryTestVolume(bs, worldHierarchy);
// Do the original queries just as before
for (int i = 0; i < NUM_SPHERES; i++)
    if (SphereTreeCollision(s[i], worldHierarchy))
    ...
// Reset everything back to not used a grouped query
EndGroupedQueryTestVolume();
...
```

需要注意的是，初始查询代码没有产生任何变化，这里只是加入了相应的分组查询代码，最终对原有代码进行优化。

可将这里所进行的优化视为一类缓存机制，且在帧间记录可供替换的首节点，并通过漫游树的方式（如同移动对象组）对首节点逐步更新。作为一种优化碰撞查询的技术手段，在 6.7 节中将进一步探讨缓存的实现方案。

6.7 通过缓存机制改善查询

缓存机制是一类非常重要的碰撞查询优化方案，其基本思想十分简单。通过记录对象最近一次（或多次）碰撞查询信息，使得该信息可用于后续查询。基于具有空间、时间一致性的可显示对象，通过这一类辅助计算可加速其测试过程。

缓存信息可分为两类：正值（用于两个碰撞对象间的快速检测）和负值（用于辅助计算两个对象间的划分）。另外，用于快速解决决策问题的某些特定信息也称为“证据”数据。

除了层次结构中的“证据”数据和首节点，缓存机制还可用于记录距离值的上限和下限，该值可针对后续查询进行对象的排除/包含操作。在后续内容中，对于缓存机制并不进行全方位的描述，但会通过几个示例分析其工作方式。

为了简化缓存操作，可对每一个独立的查询设定一个关联 ID，且该 ID 可用于数据缓存哈希表中的键值。

另外，还需要考虑游戏的重启、对象瞬移以及激活/钝化等操作，类似的事件都会对缓存系统带来隐患。最后，当出现上述异常事件时，须对缓存结构进行重置或保护操作。

6.7.1 表面缓存：缓存相交图元

一类较为直观的缓存方案是：当对象间产生碰撞时，缓存对象中的相交图元，且这一类图元可视为碰撞过程中的“证据”数据。在后续操作中，若相同的两个对象再次执行相交测试，将优先测试缓存图元以考查其是否仍处于碰撞状态。若是，则查询可即刻返回且无须检查全部层次结构。但此类缓存算法只适用于“简单的”查询测试，当需要计算详细的碰撞信息时，该缓存方案则无能为力。

这里存在两种缓存规划方案用于存储图元：共享式或分布式缓存机制。共享式缓存是一类全局缓

存机制且包含了任意数量的数据对；分布式缓存则是一类局部缓存机制，其中图元存储于对象内部。

共享式缓存通常使用哈希表加以实现，其中碰撞图元可由两个对象 ID（如指针地址）所创建的键值加以注册。共享式缓存的优点是系统可以方便地管理与多个碰撞相关的某一对象，其缺点是在进行数据对查找时搜索哈希表会产生额外开销。例如，当对象不再处于碰撞状态时须移除相应的数据对。然而，移除操作也并非想象中的那样简单，且通常采取一种延迟更新的策略，其中，只有当后续查询操作也失效时，才执行数据对的删除操作。但该方案将会滞留某些数据对并占据缓存空间，并最终可能导致退化现象的发生。

分布式缓存的优点是可方便地获取缓存图元。然而，若各对象中只是某一图元加以缓存，则该对象与其他多个对象间的碰撞将会使缓存空间过早地更新，从而导致该对象的无效渲染。例如，假定一行中包含 3 个对象 A、B 和 C，A 依次与 B、C 产生碰撞。当首先对 A、B 进行测试时，A、B 各自包含其“证据”图元；随后，B、C 之间进行测试，并产生新的“证据”图元存储于 B 中，最终覆盖了其原有的图元。当 A、B 之间再次进行测试时，当前缓存图元并未发生碰撞，因而需要再次执行一次全碰撞测试（[Rundberg99]）。

这一问题可通过在对象内增加缓存图元的数量加以解决。然而，缓存 n 个图元则至少需要执行 n^2 次测试以确定对象间的碰撞状态。因此，在实际应用中，图元数量 n 大约限制为 2～6。共享式缓存或全局缓存的代码片段如下所示：

```
int ObjectsCollidingWithCache(Object a, Object b)
{
    // Check to see if this pair of objects is already in cache
    pair = FindObjectPairInCache(a, b);
    if (pair != NULL) {
        // Is so, see if the cached primitives overlap; if not,
        // lazily delete the pair from the collision cache
        if (PrimitivesOverlapping(pair->objAPrimitives, pair->ObjBPrimitives))
            return COLLIDING;
        else DeleteObjectPairFromCache(a, b);
    }
    // Do a full collision query, that caches the result
    return ObjectCollidingCachePrims(Object a, Object b);
}

int ObjectCollidingCachePrims(Object a, Object b)
{
    if (BVOverlap(a, b)) {
      if (IsLeaf(a) && IsLeaf(b)) {
        if (CollidePrimitives(a, b)) {
            // When two objects are found colliding, add the pair
            // along with the witness primitives to the shared cache
            AddObjectPairToCache(a, b);
            return COLLIDING;
        } else return NOT_COLLIDING;
    } else {
    ...
        }
```

```
    }
    ...
}
```

[Rundberg99]不仅针对缓存图元执行测试。当缓存图元并未相交时，Rundberg 指出：其邻接图元也将存在碰撞的可能性，并转而对其周边图元进行测试。当然，该过程假定邻接图元具有良好的可访问性。最终，Rundberg 得出以下结论：该方案将平均计算速度提高 1.1～2.0 倍。

6.7.2 前界面追踪

当两个对象层次结构之间彼此测试时，需要执行一定数量的节点-节点间的比较测试。这些节点对也可视为树中的构造节点，即碰撞树。针对启发式规则以及交互式下降规则，碰撞树可设计为二叉树；而针对同步下降规则，可考虑使用 4 叉树。

若两对象间不再处于碰撞状态，即树遍历过程止于非相交状态的节点-节点对时，碰撞树中的全部内部节点即为相应的碰撞节点对。此处，相关的组合叶节点为首次处于非碰撞状态的节点-节点对。

最终结果显示，这一类叶节点形成了碰撞树的前界面，并记为两个对象处于分离状态的“证据”数据（如图 6.9 所示）。

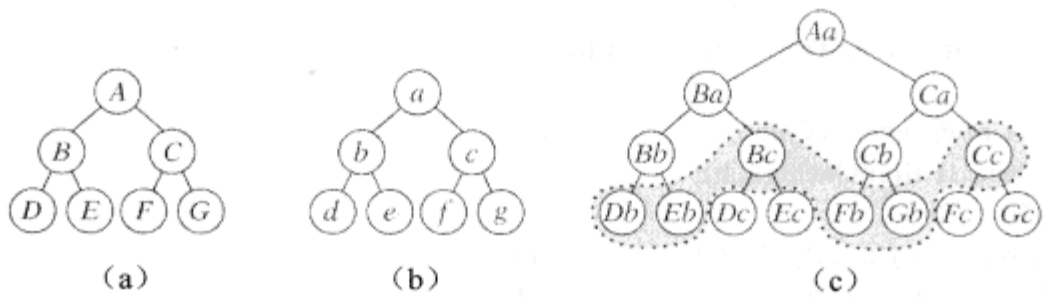


图 6.9 （a）对象 1 的层次结构。（b）对象 2 的层次结构。（c）由某种遍历方式构造的碰撞树。阴影部分表示为该结构的前界面。其中，（假定）对象间处于分离状态

鉴于对象间一般处于平稳的相对运动状态，该前界面通常也将呈现出一种稳定的状态，最多只是出现某些局部变化。因此，无须从根节点重复构造、遍历碰撞树，可利用这种一致性记录、复用各查询间的前界面数据，且只需对产生变化的相关节点采取渐进式更新。

这里参考 Li 和 Chen 在[Li98]中的相关内容，假定前界面包含 n 个叶节点且碰撞树采用二叉树（这意味着采取“下降最大节点”这一下降规则）。在包含了 n 个叶节点的二叉树中，内部节点的数量为 $n-1$。当对前界面节点加以维护且在内部节点间移动时，采用前界面节点作为起始点将比从根节点处开始测试节省工作量。

若无须移动前界面节点，且仅其父节点处于碰撞状态，则至少 $n/2$ 个节点需执行碰撞检测。全部始于前界面节点的、须测试的节点数量为 $n+n/2=3n/2$。这再次表明，与从根节点处开始遍历相比，其结果将至少节省 $2n$ 个节点。

在移动了前界面节点后，需检测的相关节点数量取决于前界面节点移动的层次数量。如果只是移动了一层，则全部被测试的节点数量为 $n+n/2+n/4=7n/4$。然而在这种情况下，从根节点处开始遍历显然开销更小一些：只需测试 n 个节点——确切地讲，是 $n/2+(n/2-1)=n-1$ 个节点。

这里，给定节点下移、静止、上移的概率分别为 p_D、p_S 和 p_U，则通过下式，免测试节点的期望数量值为：

$$s = p_D n + p_S(2n - 3n/2) + p_U(n - 7n/4)$$
$$= p_D n + p_S n/2 - p_U 3n/4$$

此处，假定上述操作有等概率值，即 $p_D = p_S = p_U = 1/3$，则免测试节点的数量仅为 $s = n/4$——这和前面的分析内容多少有些出入。

实际上，更新前界面节点的开销主要取决于节点在碰撞树中是处于静止状态还是实施了上移操作。另外，为了能够方便地测试父节点的相交状态，在相关数据结构中，应适当地维护父节点指针，以使其包含前界面节点信息。

这里，Li 和 Chen 采取了一种较为智能的替代方案：只是在查询过程中检测前界面节点的尺寸（测试节点的数量）。若尺寸增加，则预测对象间彼此靠近，且保留当前的前界面节点以用于下次查询；若尺寸保持不变，则预测对象间彼此背离运动，且在下一次查询过程中重建前界面节点。通过该方式，无须检测前界面节点的父节点，从而可节省大量计算开销。类似的性能函数如下所示：

$$s = p_D n + p_S n - p_U 0$$

其中，当 $p_D = p_S = p_U = 1/3$ 时，有 $s = 2n/3$——计算开销获得了显著的改善。更为重要的是，该函数针对任意概率分布都将呈现非负值，因而算法的执行效果不会低于根节点遍历方案。

若前界面节点在上移操作过程中突然下移，为了避免这种不稳定状态下的重建过程，Li 和 Chen 在一次查询内采取了重建延迟方案。实际上，仅当连续两次查询间前界面节点未出现增长时才需对其进行重建操作。这里，Li 和 Chen 采用球体树实现了相关测试，其结论为：针对基础的前界面节点更新算法，平均计算速度提高了 40%；而采用更新延迟方案，平均计算速度提高了 84%——这已十分接近 100%的优化率。若更新延迟操作出现于多次查询之间，则将导致计算性能的下降。

综上所述，由于所节省的计算量将正比于前界面节点中的叶节点数量，因此对于大型层次结构，对前界面节点实现维护将会获得显著的效果；而对于小型层次结构，鉴于可忽略的节点数量较少，其性能的提升往往并不明显。

6.8　小结

包围体层次结构可用于加速对象的粗略测试以及精细测试过程。对于粗略测试过程，可构建场景对象的 BVH 以剔除其范围外（即组对象的包围体不与当前查询对象的包围体相交）的对象组；对于精细测试过程，BVH 通常构建于（复杂）对象的局部元素之上，且两两组合测试将图元间的计算限定在包围体相交的对象局部元素之上。

本章还考查了 BVH 的期望特征，并探讨了 3 种 BVH 的构造策略——自顶向下算法、自底向上算法以及基于插入方法的构造算法。自顶向下算法主要将对象数据连续划分至相应子集中，这其中也包含了相应的划分策略。本章还分析了 3 种树型遍历算法——深度优先算法、广度优先算法以及最佳-优先搜索算法。在执行两个 BVH 之间的测试时，需要考查其子树的访问顺序，因此本章还分析了相关的下降规则。

本章的最后还介绍了高效的树型表达方式以及遍历算法，相关内容还将在第 13 章中加以分析。

第7章 空间划分

首先回忆第 2 章介绍的相关内容：对象间的两两组合测试往往计算开销较大；同时，粗略测试阶段主要是将上述组合测试限定在可能出现相交的近距离对象之上。空间划分技术则以另一种方式提供了相应的粗略测试方案：将空间划分为不同的区域并测试对象间是否在同一空间区域相交——这将大大地降低组合测试的数量。本章将介绍 3 种空间划分算法：网格、树以及空间排序算法。

7.1 均匀网格

一类有效的空间子划分方案可利用均匀网格对空间实施覆盖，该网格可将空间划分为多个均等的区域，或称之为网格单元。这样，每个对象都将与其覆盖的网格单元关联。针对这种情况，由于存在网格单元重叠的多个对象将处于碰撞状态，则碰撞测试将限定在与当前对象共享网格单元的那些对象。两个对象的间距越大，其网格单元重叠的几率也就越小。采用相应的空间网格后，彼此可能相交的对象将执行进一步的组合测试。

基于网格排列的均匀特性，对与特定坐标对应的网格单元的访问将变得简单而快捷：只需利用世界坐标值除以网格单元的尺寸即可获得网格单元的坐标。当给定特定网格单元的坐标后，其邻接网格单元的定位也将变得十分容易。这种概念上以及实现上的简洁性使得网格在空间子划分计算中成为首选方案。

7.1.1 网格单元的尺寸

基于性能方面的考虑，网格算法需要选择适当的单元尺寸。与网格单元尺寸相关的 4 项内容将对计算性能产生直接影响（如图 7.1 所示）。

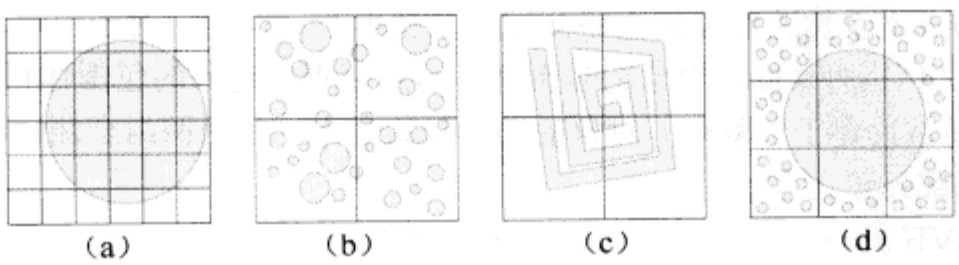


图 7.1 网格单元的尺寸问题。（a）网格过于密集。（b）（与对象尺寸相比）网格过于稀疏。（c）（与对象复杂程度相比）网格过于稀疏。（d）既密集又稀疏的网格

1．网格过于密集。若网格单元的尺寸较小，则采用对象关联信息进行更新的网格单元数量将过于庞大，这将耗费额外的时间和空间。该问题可形象地表达为：在体积与大象相仿的对象与火柴盒大小

的单元格之间进行匹配。同时，考虑到需要定位、更新的对象关联数量，单元网格过于密集也会对物体的移动产生不良影响

2．网格过于稀疏（相对于对象尺寸）。如果对象尺寸较小而网格单元尺寸过大，则每一个网格单元中将会包含过量的物体对象。由于网格单元内部对象间皆采用两两组合测试，因此这将导致最坏情况下的、全部组合间的测试。

3．网格过于稀疏（相对于对象的复杂程度）。此处，网格单元将与对象在尺寸上实现较好的匹配。然而，过于复杂的对象将对其组合比较测试产生不良影响。较为理想情况的状态是：设置较小的网格单元，且对象分解为多个组分以实现与当前网格单元的良好匹配。

4．相对于较小的对象，网格单元过于稀疏；而相对于较大的对象，网格单元又过于密集。对于尺寸相差较大的多个物体对象，这类既“密集”又“稀疏”的网格单元同时存在的情况是可能的。

关于第4种情况，还将在7.2节中加以描述（即层次网格）。针对前3种情况，为了与任意方向上的最大物体对象匹配，需要将网格单元尺寸设置得较大一些（但不可过大）。通过这种方式，可保证对象所覆盖的网格单元数量不超过4个（对于3D网格则不超过8个）。稍后可看到，减少对象所覆盖的网格单元数量将是一项十分重要的内容——这将减少对象插入、更新所需的计算量，并执行最终的相交测试。

这里假定在每一帧之间不存在对象相互穿越这一状况，且网格单元与对象的尺寸相仿，则网格内对象的最大数量将维持在一个较小的常量范围内。其结果为，在一个网格单元内部采用全组合测试方法处理碰撞问题时，将只涉及少量的（两两）对象组合测试。

当选定网格单元的大小以后，可将场景空间的直径除以网格单元的直径，从而得到网格的最终尺寸。另外，如果还考虑到存储网格后的内存使用情况，则单元格尺寸的增加是以性能降低为代价的。然而，网格单元数量的增加并不总意味着占用更多的内存消耗（更多相关内容可参见后续内容）。

在光线跟踪技术中，一种较为常见的确定网格尺寸的方案是 $n^{1/3}$ 规则：给定 n 个对象并将空间划分为 $k \times k \times k$ 个网格，其中 $k = n^{1/3}$。这里，网格单元与对象之间的比率可称之为网格密度。近期的研究结果表明：应尽量避免网格密度值为1，[Havran99]则建议采用较大的密度值。

曾有人提出，网格单元的最佳尺寸很难确定，并且如果单元尺寸未加以正确设定，则无论是在性能上还是在复杂的访存密集型应用中，网格计算的开销都将是十分巨大的（参见[Cohen95]）——但根据以往的实际操作经验，这一观点并不正确。实际上，对于大多数应用而言，较好的参数值与较差的参数值的选择几率是均等的。

7.1.2 采用链表数组表示的网格

存储网格中的对象时，一种较为自然的方式是根据网格尺寸分配相应的数组，并将网格单元与数组元素一一对应。当网格单元中包含多个对象时，数组元素可指向一个链表；若网格单元内容为空，则数组元素设定为NULL。如果对象在表头插入，则插入复杂度仅为 $O(1)$。对于单链表，更新、删除任一对象的复杂度为 $O(n)$。然而，若采用双向链表并在对象间维护指向对象所处的全部网格单元链表项的指针，相应的更新、删除操作复杂度也可维持在 $O(1)$。

较为简单的链表维护方式是将链接置入对象内部。当采用单元格尺寸对其数量加以限定时，对象内嵌链接方式不失为一种较好的解决方案，这不仅改善了数据缓存一致性问题，还避免了特定系统中

链接网格单元的内存分配问题，从而简化了编码过程。

但这一类紧凑数组（dense array）也存在一个缺陷：对于大型网格，若只在网格单元元素中存储表头数据，其内存需求也将随之增长。另外，增大网格单元的尺寸也并不可取，这会降低内聚对象区域内的计算性能——这一点需要额外关注。后续几个小节将会介绍几种低内存占用率的网格表达方式。

7.1.3 哈希存储与无限网格

相对于网格的紧凑数组存储方式，一类有效的替代方案是将各网格单元映射至由 n 元固定集形成的一个哈希表中（如图 7.2 所示）。此处，哈希表元将包含对象的链表。“网格”在这里更多地是体现一种概念上的含义，且并没有占用相应的内存空间。下面是一个简单的乘法型哈希函数示例，将一个由网格单元坐标值构成的组元映射至表元索引：

```
// Cell position
struct Cell {
    Cell(int32 px, int32 py, int32 pz) { x = px; y = py; z = pz; }
    int32 x, y, z;
};

#define NUM_BUCKETS 1024

// Computes hash bucket index in range [0, NUM_BUCKETS-1]
int32 ComputeHashBucketIndex(Cell cellPos)
{
    const int32 h1 = 0x8da6b343;        //Large multiplicative constants;
    const int32 h2 = 0xd8163841;        //here arbitrarily chosen primes
    const int32 h3 = 0xcb1ab31f;
    int32 n = h1 * cellPos.x + h2 * cellPos.y + h3 * cellPos.z;
    n = n % NUM_BUCKETS;
    if (n < 0) n += NUM_BUCKETS;
    return n;
}
```

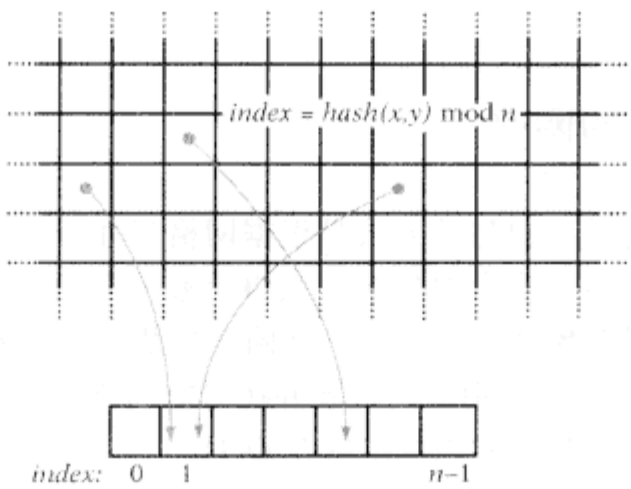


图 7.2 通过哈希函数将一个（无限大的）2D 网格映射为哈希桶

需要注意的是，这里显示的哈希函数虽然易于构造，但就其性能而言，也只能用“不好不坏”加以形容（无法保证该哈希函数的健壮性）。关于更多较为高级的哈希函数构造方案，可参见[Binstock96]、[Jenkins97]以及[Knuth98]。

按照这一类方式实现哈希映射，大大降低了对网格尺寸方面的要求。这里，映射函数将世界坐标转换为有限数量的网格单元（因此，无须在访问邻接网格单元时进行边界检查）。虽然场景还可能包含无限多个网格单元，但在某一时间范围内，与对象相交的网格单元数量仍是有限的。因此，哈希网格的内存存储将与对象的数量相关，而与网格的尺寸无关。

这里不要使用哈希表对网格单元的内容（是否为空）进行查询，可以对全部网格单元预设定一个全 1 的位紧凑数组，从而对网格单元中的内容是否为空实现快速查询（当然，这类优化措施只限于固定尺寸的网格）。

对于哈希表中的冲突问题（即两个键值映射至同一哈希表元），一般采取两种解决方案：开放式散列法和封闭式散列法。在开放式散列法（也称为分离链接）中，每一个表元（桶）将包含一个记录链表。当在哈希表中实现添加操作时，将计算表元索引并在对应表元的链表起始处添加数据项。此处，每一个存储记录都将包含一个键/值对。此时，键代表了最初的网格单元坐标；值则代表了该网格单元中对象链表的表头。由于这种方式采用了嵌套表的方式，因此相关数据结构的更新操作（例如网格单元及其对应的表元中的添加、删除操作）将变得稍显复杂。

相比较而言，封闭式散列法（或称为开放地址法）则直接将记录存储于哈希表元中。当添加项的表元索引指向一个已占据位置时，将继续计算表元索引，直至发现一个空的表元位置并执行插入操作。因此，这一类方法只是简单地测试后续表元，且必要时将在表尾处进行回绕，直至找到一个有效的位置，故也称作线性试探法。虽然线性试探法具有较好的缓存特征，但当哈希表被填满后，其性能将变得十分低效。另外，采用线性探测法时，哈希表的利用率不足最佳性能时的一半。关于试探算法，还包括其他一些较好但稍显复杂的解决方案，读者可参考[Knuth98]以考查线性试探算法和置换试探算法的详细分析过程。

封闭式散列法的一个问题是：对象无法从哈希表中直接删除，且必须采用延迟删除的方式以保证后续查询操作成功。这里，延迟操作将对相关位置进行标记，以防止插入操作误认为该位置已被占用，从而错误地终止其他对象的哈希表搜索过程。

第 3 种方案是使用开放式散列法，但去除首级链表且令表元直接包含对象表——这将生成简洁、低开销的代码。但是，由于缺乏存储键，将会出现下列两个问题：首先，当对特定网格中的内容执行测试时，对应表元可能包含目标对象以及来自于其他网格单元中的对象，即呈现同一表元间的映射；另外，为了避免对无关对象执行开销较大的精确测试，需前期进行相应的深度测试，即利用包围体测试剔除无关对象。总之，这种剔除操作是一类较好的全局策略，其中，非相交对象可共享哈希表元而无须考虑哈希表的表现方式。

其次，相关问题还会出现在同一查询内的多个网格单元间的测试中。若两个或多个网格单元“重叠”于同一非空哈希表元，则同一查询中其相关内容将被检测多次。然而，这一看似严重的问题可通过时间戳算法加以解决，其中，哈希表元在每次查询时将被标记，该标记包含了一个唯一 ID 且随每次查询操作而改变。例如，在某次查询过程中，若哈希表元已标记为当前 ID，则该表元已测试过并可忽略。关于时间戳算法，7.7.2 节将进一步探讨。

[Reinxard00]针对对象移出网格覆盖区域这一情况给出了相应的网格扩展算法。其中，将网格分为

逻辑和物理两部分：当对象移出网格覆盖范围时，逻辑网格可实现相应扩展。然而，通过将逻辑网格单元的坐标对物理网格尺寸取模，这些额外引入的网格单元将以环绕的方式与现有物理网格对应。因此，逻辑网格的尺寸现在可以用来确定：物理网格边界附近的对象是否与其对边邻接网格单元中的对象产生碰撞。

7.1.4 静态数据存储

若网格数据为静态数据（例如相关内容以场景多边形而非静态对象呈现），则无须使用链表（[Haines99]）。这里可采用数组存储全部网格单元数据。图 7.3 分别显示了使用链表和数组存储的同一网格数据。

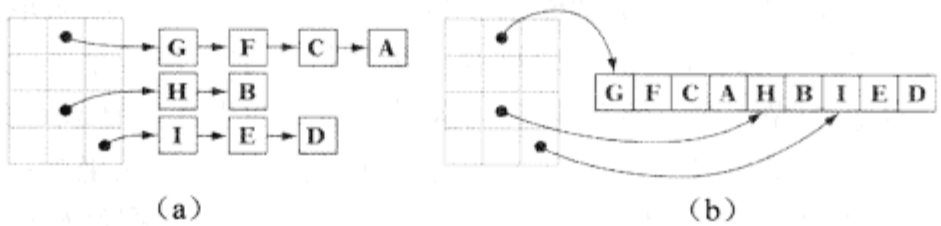


图 7.3 （a）采用链表存储静态数据的网格。（b）采用数组存储静态数据的网格

网格数据可通过两步处理操作置入数组中。第一步，循环处理静态数据。这里并不将数据赋予与其相交的网格单元中，而是增加各网格单元对应的计数器。当全部数据处理完毕后，即获取全部 n 个网格单元的计数值，且 a_k 为单元格 C_k 中的对象数量。可按如下方式分配包含 m 个元素的数组：

$$m = \sum_{0 \leq i < n} a_i$$

同时，各网格单元将分别关联一个索引号 b_k 至当前数组中。其中：

$$b_0=0$$

$$b_k=b_{k-1}+a_{k-1}$$

在第二步中，再次对当前数据重复处理。此时，与网格单元 C_k 相交的 a_k 个对象将根据数组内的关联索引 $b_k \sim b_k+a_k-1$ 加以存储。由于各单元网格内的全部数据连续地存储于数组中，且各存储项不存在指针链接，因此与链表形式的网格相比，基于数组的网格将占用更少的内存且更能体现高效的缓存性。

7.1.5 隐式网格

针对网格的存储特征，还存在一类隐式存储方案，即采用多个数组间笛卡儿积的方式。此处，网格体现为两个数组（在 3D 环境中为 3 个数组），其中数组一对应网格的行数据，数组二对应网格的列数据。如前所述，每一个数组元素指向一个对象链表。针对行数组和列数组（如图 7.4 所示），当对象与网格单元交叠时，可将其置入对应链表从而实现插入操作。对象的相交测试则通过下列方式实现：考查该对象是否与其交叠的行网格单元和列网格单元中的其他对象相交。

与紧凑数组这一存储方式相比较，该方案具有较好的表插入灵活性。例如，若对象位于单独的隐式网格单元中，则该对象将被插入至两个表中（行表和列表），而在前述方案中，对象只被置入一个表中；然而，若对象与一个 4 × 5 隐式网格单元交叠，则该对象只需插入至 8 个表中（4 个行表和 4 个

列表），而非 16 个表，从而改善了最坏情况下的插入操作。另外，当对象只与少数网格单元交叠时，相交测试的开销反而变大：不仅要测试表中多个对象，还需要对多个表进行检测。

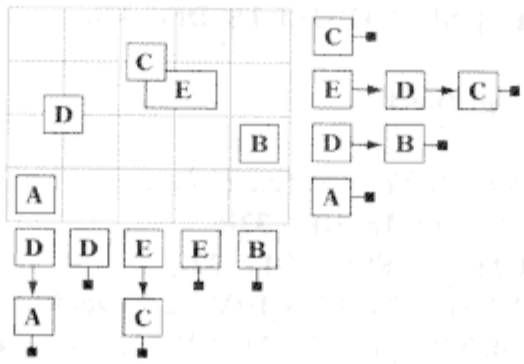


图 7.4　一个 4 × 5 网格可隐式定义为 9 个（4 + 5）链表的交集。这里，共计 5 个对象插入至链表中并显示了其在网格中所处的位置

一种较为新颖的存储算法是采用下列方案：针对网格的每一行和每一列以及场景对象，分配一个全 1 的位数组。因此，对象相对于网格的置入操作简单地变为针对对象对应的网格单元，只需在位数组中设定相应位即可（如图 7.5 所示）。

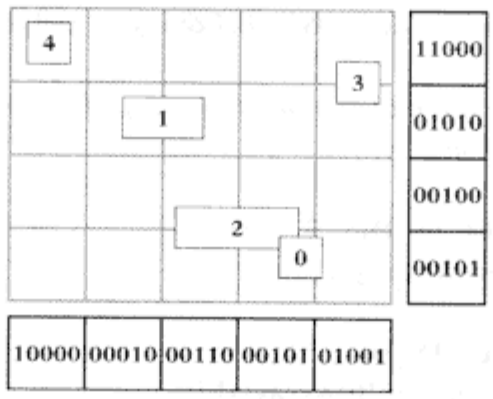


图 7.5　一个 4 × 5 网格可隐式地定义为 9 个（4 + 5）位数组的交集。这里，共计 5 个对象插入至网格中。因此，某一位数组位集合中的数位位置值将与相应行或列的对象对应

当且仅当网格单元行、列位数组中均呈现对应对象的位设置，该对象才位于网格单元中。其结果为：基于当前网格单元对象的表现方式，测试简化为在两个位数组间执行 AND 位操作。

当对象与多个网格单元相交时，位操作符的分配律使得相应计算量大大降低。例如，假设某一对象分别与行 r_i、r_i+1 以及列 c_j、c_{j+1} 的网格单元相交，则对象间的相交计算可按下列方式执行：4 个网格单元(r_i, c_j)，(r_i, c_{j+1})，(r_{i+1}, c_j)和(r_{i+1}, c_{j+1})进行“&”位操作（交集）后，再进行“|”操作（并集）。如果 **r[i]**和 **c[j]**分别对应行网格单元 *i* 和列网格单元 *j* 的位掩码，则结果位集 **b** 如下所示：

```
b = (r[i] & c[j]) | (r[i] & c[j+1]) | (r[i+1] & c[j]) | (r[i+1] & c[j+1]);
```

这里，使用位操作符的分配律将使得计算进一步简化为：

```
b = (r[i] | r[i+1]) & (c[j] | c[j+1]);
```

上式可归纳为任意数量相交网格单元的计算公式，其中首先对交叠的行网格单元以及列网格单元进行 OR 操作；而后对结果位域执行 AND 操作并得到最终结果。该算法的代码如下所示：

```
// Define the two global bit arrays
const int NUM_OBJECTS_DIV_32 = (NUM_OBJECTS + 31) / 32;      // Round up
int32 rowBitArray[GRID_HEIGHT][NUM_OBJECTS_DIV_32];
int32 columnBitArray[GRID_WIDTH][NUM_OBJECTS_DIV_32];

void TestObjectAgainstGrid(Object *pObject)
{
    // Allocate temporary bit arrays for all objects and clear them
    int32 mergedRowArray[NUM_OBJECTS_DIV_32];
    int32 mergedColumnArray[NUM_OBJECTS_DIV_32];
    memset(mergedRowArray, 0, NUM_OBJECTS_DIV_32 * sizeof(int32));
    memset(mergedColumnArray, 0, NUM_OBJECTS_DIV_32 * sizeof(int32));

    // Compute the extent of grid cells the bounding sphere of A overlaps.
    // Test assumes objects have been inserted in all rows/columns overlapped
    float ooCellWidth = 1.0f / CELL_WIDTH;
    int x1 = (int)floorf((pObject->x - pObject->radius) * ooCellWidth);
    int x2 = (int)floorf((pObject->x + pObject->radius) * ooCellWidth);
    int y1 = (int)floorf((pObject->y - pObject->radius) * ooCellWidth);
    int y2 = (int)floorf((pObject->y + pObject->radius) * ooCellWidth);
    assert(x1 >= 0 && y1 >= 0 && x2 < GRID_WIDTH && y2 < GRID_HEIGHT);

    // Compute the merged (bitwise-or'ed) bit array of all overlapped grid rows.
    // Ditto for all overlapped grid columns
    for (int y = y1; y <= y2; y++)
        for (int i = 0; i < NUM_OBJECTS_DIV_32; i++)
            mergedRowArray[i] |= rowBitArray[y][i];
    for (int x = x1; x <= x2; x++)
        for (int i = 0; i < NUM_OBJECTS_DIV_32; i++)
            mergedColumnArray[i] |= columnBitArray[x][i];

    // Now go through the intersection of the merged bit arrays and collision test
    // those objects having their corresponding bit set
    for (int i = 0; i < NUM_OBJECTS_DIV_32; i++) {
        int32 objectsMask = mergedRowArray[i] & mergedColumnArray[i];
        while (objectsMask) {
            // Clears all but lowest bit set (eg. 01101010 -> 00000010)
            int32 objectMask = objectsMask & (objectsMask - 1);
            // Get index number of set bit, test against corresponding object
            // (GetBitIndex(v) returns log_2(v), i.e. n such that 2^n = v)
            int32 objectIndex = GetBitIndex(objectMask) + i * 32;
            TestCollisionAgainstObjectNumberN(objectIndex);
            // Mask out tested object, and continue with any remaining objects
            objectsMask^= objectMask;
        }
    }
}
```

下面这一类采用位数组且经过简化后的隐式网格算法，可作为一种粗略剔除方案。这里不采用位

数组显示某一对象在网格中的行、列位置，而是使用一个位来表示相应行、列中该对象的状态：如果与该对象交叠的相应行、列中的对应位没有被设定，则该对象不处于相交状态，且无须执行下一步测试。

表 7.1 显示了下列 3 种网格表达方式的内存使用状况：采用位数组的隐式网格、紧凑数组（采用双向链表）以及稀疏数组（仍采用双向链表）。当对约 1000 个对象、尺寸为 100×100 的网格进行测试时，采用位数组的隐式网格算法将占用更少的内存，同时，该算法以一种缓存友好的方式使用内存，并可作为一种高效的空间划分算法加以使用。

表 7.1　3 种不同表达方式的内存使用状态（以字节计）：隐式网格（采用位数组）、紧凑数组（采用双向链表）以及稀疏数组（采用双向链表）

		网格尺寸（$m \times n$）				
		10 × 10	50 × 50	100 × 100	500 × 500	
网格中的对象	10	40	200	400	2,000	隐式网格（位数组） $bytes=[objects/8](m+n)$
	100	260	1,300	2,600	13,000	
	1,000	2,500	12,500	25,000	125,000	
	10,000	25,000	125,000	250,000	1,250,000	
	10	480	10,080	40,080	1,000,080	紧凑数组（双向链表） $bytes=4mn+objects \cdot (4+4)$
	100	1,200	10,800	40,800	1,000,800	
	1,000	8,400	18,000	48,000	1,008,000	
	10,000	80,400	90,000	120,000	1,080,000	
	10	240	560	960	4,160	稀疏数组（双向链表） $bytes=4(m+n)+objects \cdot (4 \cdot 2+4 \cdot 2)$
	100	1,680	2,000	2,400	5,600	
	1,000	16,080	16,400	16,800	20,000	
	10,000	160,080	160,400	160,800	164,000	

7.1.6　使用均匀网格的对象间的测试

如前所述，网格单元的尺寸通常应大于最大对象的尺寸。通过这一方式，可以保证对象最多与其邻接网格单元相交，从而极大地简化了相交测试代码。但当对象置入网格单元并进行对象间的测试时，仍需注意一些关键问题。例如，对象应置入一个较具代表性的网格单元，还是其交叠的全部网格单元中？如果只是使用单一网格单元，应采取何种对象特征确定其占据的网格单元？下面将根据测试的查询方式对上述问题加以分析：每次执行单一测试（且对象可在连续测试间移动），抑或同时执行全部测试。

7.1.6.1　每次执行单一测试

考查下列情形：对象只位于一个网格单元中。若对象与其对应网格单元发生关联，除了测试该关联网格单元内的对象，还需要额外测试其邻接网格单元。同时，网格单元的确定将依据对象与网格单

元的相交方式，以及对象与网格单元所采用的关联特征。此处，“关联特征”一般按照下列方式选取：对象包围球的中心位置，或轴对齐包围盒的最小顶点位置（一般为左上角顶点）。

如果对象基于其中心位置放置，则邻接网格单元中的任一对象都可能与当前网格单元交叠，并且与当前对象发生碰撞。因此，这里不考虑具体是哪一个邻接网格单元，当前对象与所有邻接网格单元相交，且针对其中的对象进行碰撞测试，相关代码如下所示：

```
// Objects placed in single cell based on their bounding sphere center.
// Checking object's cell and all 8 neighboring grid cells:
check object's cell
check northwest neighbor cell
check north neighbor cell
check northeast neighbor cell
check west neighbor cell
check east neighbor cell
check southwest neighbor cell
check south neighbor cell
check southeast neighbor cell
```

这里共计测试了 9 个网格单元（在 3D 环境下为 27 个网格单元），这也许并不是所期望的答案。如果使用轴对齐包围盒的最小顶点，仅当当前对象与邻接网格单元交叠，才需对后者进行测试。该测试的 2D 伪代码如下所示：

```
// Objects placed in single cell based on AABB minimum corner vertex.
// Checking object's "minimum corner" cell and up to all 8 neighboring grid cells:
check object's "minimum corner" cell
check north neighbor cell
check northwest neighbor cell
check west neighbor cell
if (object overlaps east cell border) {
    check northeast neighbor cell
    check east neighbor cell
}
if (object overlaps south cell border) {
    check southwest neighbor cell
    check south neighbor cell
    if (object overlaps east cell border)
        check southeast neighbor cell
}
```

不难看出，最坏情况下需要检测全部 9 个网格单元；而在最佳状态下，只需测试 4 个网格单元。因此，与中心点方案相比，这便是采用最小顶点放置对象的优越性所在。另外，如果对象被置入其全部交叠的网格单元中，单独的对象测试只需检测那些与其 AABB 相交的网格单元——全部碰撞对象已确保位于这些网格单元中。该测试简化为：

```
// Objects placed in all cells overlapped by their AABB.
// Checking object's "minimum corner" cell and up to 3 neighboring grid cells:
check object's "minimum corner" cell
```

```
if (object overlaps east cell border)
      check east neighbor cell
if (object overlaps south cell border) {
      check south neighbor cell
      if (object overlaps east cell border)
            check southeast neighbor cell
}
```

当前，最坏情况下只需测试 4 个网格单元，且最佳状态下只需测试一个网格单元。虽然这有助于实现快速测试，但当对象产生移动时，该方案将使更新代码变得十分复杂（在 3D 网格中至少需更新 8 个网格单元），与单一网格单元存储方式相比，很可能会占用更多的内存空间。除此之外，该方案还会重复报告两个对象间的碰撞结果——因此，还需要对组合碰撞测试状态进行适当的维护（参见 7.7.1 节）。

7.1.6.2　每次执行全部测试

这里，如果不采取一次测试一个对象的方式而是同时测试全部对象，则单一网格单元的置入问题可利用对象组合测试的互换性得以优化。亦即，对象 *A*、*B* 间的检测将等同于对象 *B*、*A* 间的检测。例如，假定当前网格单元中的对象与其东侧邻接网格单元执行碰撞检测，并随后到达该东侧网格单元且执行相应的邻接网格单元测试，此时即可省却其西侧网格单元的检测过程。其结论为：全部产生碰撞的对象还需与其他对象执行碰撞测试。换句话说，该方案并不检测对象静态网格内的某一移动对象。需要注意的一点是，网格单元间的全部交叠关系将体现在最终的测试结果上。例如，一类常见的错误则认为，无须对 NE-SW（东北-西南）网格单元进行测试，但网格单元 *A*、*B* 中的两个对象可能在第 3 个网格单元中碰撞（如图 7.6 所示）。

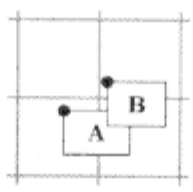


图 7.6　对象 *A* 和 *B* 根据其左上角位置置入相关网格单元中。此时，碰撞可能出现在第 3 个网格单元中。因此，对象间的相交测试还需检测其 NE 和 SW 邻接网格单元

这里，继续讨论对象按中心位置放置这一情形：该过程无须检测全部邻接网格单元。亦即，只需检测一半数量的邻接网格单元即可，剩余的一半反向测试可由前述讨论的互换关系加以确定。因而，测试过程如下所示：

```
// Objects placed in single cell based on their bounding sphere center.
// All objects are checked for collisions at the same time, so collisions
// in the opposite direction will be handled when checking the objects
// existing in those cells.
check object's cell
check east neighbor cell
check southwest neighbor cell
check south neighbor cell
check southeast neighbor cell
```

该算法需测试全部对象，因此这 5 个测试不能与其他任一边界相交测试结合使用，否则将会导致

“测试丢失”。例如，假定对象 *A* 不与其东侧邻接网格单元交叠，则该邻接网格单元未加测试。但是，东侧网格单元中的对象 *B* 却有可能与其西侧网格单元相交（与对象 *A* 相交），而该东侧网格单元与其西侧网格单元间目前未经测试，因此 *A*、*B* 间处于未检测状态。进一步讲，由于某些（并非全部）碰撞检测在帧间会出现重复测试，因此该算法需要对相交状态进行相应记录（如可存储于碰撞对矩阵中）。

当对象采用最小顶点置于单一网格单元中时，也可以实现类此的简化过程。这里所进行的边界交叠检测可用于减少后续邻接网格单元的测试数量。该测试过程如下所示：

```
// Objects placed in single cell based on AABB minimum corner vertex.
// All objects are checked for collisions at the same time, so collisions
// in the opposite direction will be handled when checking the objects
// existing in those cells.
check object's "minimum corner" cell
check southwest neighbor cell
if (object overlaps east cell border)
    check east neighbor cell
if (object overlaps south cell border) {
    check south neighbor cell
    if (object overlaps east cell border)
        check southeast neighbor cell
}
```

在邻接网格单元的测试过程中，可围绕网格边界填充一层仅供检测使用的网格单元。这样，便无须再对网格坐标进行范围检测。

7.1.7 网格的其他注意事项

关于网格的高效实现方式，还需考虑下列相关事项。首先，这里定义的网格表达方式将使其数据在帧间保持一致，且宜采用渐进式更新方案。虽然从整体上讲这是一类较好的执行方案，但通过制定相关参数，可得到更加简洁的算法，使得网格的表达更为清晰，并在每一帧中可顺利完成全部对象的初始化读取操作。例如，若大多数或全部对象处于运动状态，则相对于更新网格数据结构，重新读取网格的计算开销将更小。另外，各帧中重构网格时也要求其数据结构尽量保持简洁，从而减少内存消耗（因而采用单链表而非双向链表）。在每帧中重新添加对象意味着对象添加完毕后，即可对其进行相关测试，从而避免返回“*A* 与 *B* 碰撞”、“*B* 与 *A* 碰撞”这一类问题信息。

其次，无须将对象全部置入网格中。当针对某些特定分组对象进行碰撞测试时，可将一组对象数据置入网格中。随后，对于第二组对象与网格之间的测试，该对象数据不必再次置入网格中，它们可作为查询型对象加以使用。这一类与分组相关的例子包括：第一组对象为敌人，而另一组对象为玩家的子弹。

最后一个值得思考的问题是网格单元体以及网格单元的方向。例如，假定当前网格主要用于平面中的线段和对象集之间的测试（详细讨论可参见 7.4.2 节）。通常，与水平或垂直线段相比，以某一角度入射的线段一般会跨越更多的网格单元。因此，这将涉及两个网格，同时，两个网格之间包含了同一对象集且二者呈 45° 角排列。根据给定线段的方向，相应的网格即可用于相交测试，这将在一定程度上提高测试效率。

图 7.7 显示了规则的均匀网格单元与交错的六边形网格单元排列方式。这里，假定球体 A、B 的尺寸不超过网格单元各边尺寸的一半，二者分别置于一个网格单元中且每次执行单一测试。基于该方案，如果对象皆采用其中心位置置于网格单元中，则对象 A 需要与 9 个网格单元执行碰撞测试，而对象 B 只需 7 个。若对象 A、B 皆为盒体并使用左上角最小顶点置入网格单元中，则对象 A 只需测试 5 个网格单元（中心网格单元、东侧网格单元、南侧网格单元、东南侧网格单元、西北或东南侧网格单元），而对象 B 只需测试 4 个网格单元（中心网格单元、东侧网格单元、西南侧网格单元以及东南侧网格单元）。

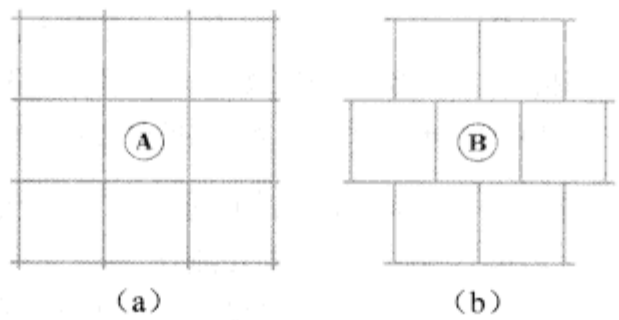


图 7.7　（a）在规则网格中，网格单元 A 具有 8 个邻接单元。（b）在六边形网格中，网格单元 B 仅有 6 个邻接单元

7.2　层次网格

均匀网格的最大问题是无法简洁地处理变尺寸对象。当对象尺寸远大于网格单元的尺寸并与多个网格单元交叠时，更新移动对象的位置将变得十分耗时。若调整网格单元的尺寸并使其包含较大对象，随着位于同一网格单元内部的较小对象数量的增加，网格的空间识别能力也随之降低，最终将导致测试性能的降低。但这一尺寸问题可通过层次网格有效地加以解决，层次网格是一类适用于装载动态移动对象的网格结构。

这里所描述的层次网格包含了多个具有不同尺寸的网格单元，且相互交叠并覆盖同一空间。根据具体的应用环境，可分为 2D 层次网格和 3D 层次网格。这里定义一个包含 n 层网格的层次结构，令 r_k 为 k 层网格单元的尺寸，则 $r_1 < r_2 < \cdots < r_n$，网格按照网格单元尺寸递增的方式加以排列。其中，第一层和第 n 层分别对应层次结构中的最底层和最高层。需注意的是，层数量可以不固定。

为了实现快速的插入和删除操作，对象将被插入至某一层上的一个网格单元中。由于对象可能触及该层中的其他网格单元，为了减少邻接网格单元的碰撞测试数量，相关对象应插入至层次网格中的对应层中，其中，网格单元应能够容纳当前对象的包围体。定义一个对象 P，其所处的 L 层记为 $L = Level(P)$。通过该方式，可保证对象与网格单元的交叠数量不会超过 4 个（在 3D 环境下为 8 个）。此处，若对象的中心位置（基于其包围球）或左上角最小顶点位置（基于其 AABB）位于某一网格单元，则可执行相应的插入操作。具体实例如图 7.8 所示。

网格单元可定义为任一相对尺寸，并且应根据现有对象的尺寸加以选择。在实际操作过程中，关于层次结构的构建过程，存在一类简单的算法且运行效果良好。其中，令第一层的网格单元仅包含最小的对象，且后续层次中网格单元的尺寸分别加倍。因而有：$k+1$ 层的网格单元将是 k 层网格单元宽度的 2 倍。重复这一加倍计算过程，直至最高层的网格单元能够包围最大对象。综上所述，该构建过程可有效地减少对象交叠的网格单元的数量。同时，针对某些特定实例，该方案仍有可改进之处。例

如，若仅存在两个对象（的尺寸），则只需包含两层的网格结构即可。

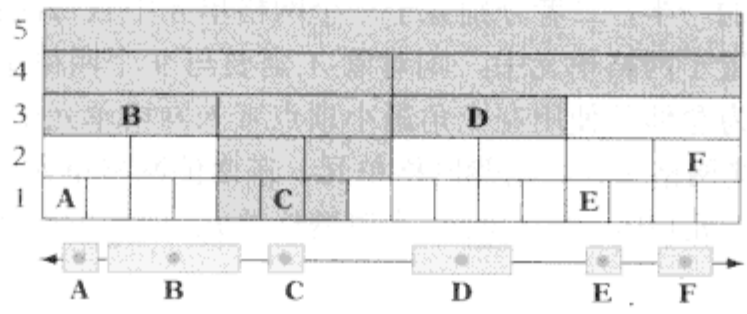


图 7.8　1D 层次结构网格。6 个对象 A～F 分别插入至相应网格层上的网格单元中，且相关网格单元分别包含对象的中心点坐标

对象 A 与存储于层次网格中的对象间的碰撞测试可通过遍历层次网格进行计算。在各层中，对象 A 不仅要与其包围体相交的网格单元执行测试，还需要测试某些邻接网格，其所包含的对象与当前网格单元相交（如 7.1.6 节中所述）。需要注意的是，仅检测对象所处网格单元的邻接网格单元有时是不够的。例如，某些大型对象在与网格进行测试时，层次网格中的最底层将会包含多个网格单元，且它们并非是邻接网格单元。

如果全部 n 个对象执行同步测试，测试过程无须处理层次网格中的各个层——各对象可在其插入层处开始测试，并遵循下列原则：较小的对象将与较大对象所处的网格层进行测试。

总体而言，层次网格结构与树型结构（如 8 叉树）差异并不大，并采用数组形式加以实现。二者的区别仅在于层次网格结构缺少“树顶”。亦即，层次网格结构只是包含了树型结构的某些层数据。另外，树型结构的顶层数据一般不包含任何对象，只是通过该节点实现遍历操作从而到达对象所处的底层节点。因此，层次网格结构省略了这一类空数据层，并以此获得速度上的提升。

为了进一步加速碰撞测试的计算过程，需要记录层次网格各层中的对象数量，且不包含对象的相关层将不在测试范围之内。

若层次网格中的各层按照紧凑数组的方式分配数据，其内存开销将十分庞大。因此，较好的实现方式是采用哈希存储，如 7.1.3 节中所述。

7.2.1　基本的层次网格实现方式

本节将介绍一类简洁、直观的层次网格实现方式。这里所使用的哈希表将采用一种无键值的开放式散列法，同时，还利用了相关的包围体测试以及时间戳算法以避免计算某些不相关或已处理过的数据。下列代码显示了一类较为紧凑的层次网格的数据结构：

```
struct HGrid {
    uint32 occupiedLevelsMask;                  // Initially zero (Implies max 32 hgrid levels)
    int objectsAtLevel[HGRID_MAX_LEVELS];       // Initially all NULL
    Object *objectBucket[NUM_BUCKETS];          // Initially all zero
    int timeStamp[NUM_BUCKETS];                 // Initially all zero
    int tick;
};
```

这里假定对象根据包围球实现层次网格中的插入操作。出于操作的简洁性，代码将采取在哈希表

元内使用单链表的方式链接对象，且链表指针内嵌于对象内部。另外，若使用双向链表，则删除操作的时间复杂度将为 $O(1)$而非 $O(n)$；同时，若将层次网格单元与数组（包含对象指针的）相关联，则删除操作的时间复杂度也可达到 $O(1)$。后者需要某些额外的记录操作以处理数组的溢出问题，但总体上讲，该算法仍不失为一类快速、缓存友好的选择方案。下列数据结构显示了相关对象所包含的变量：

```
struct Object {
    Object *pNextObject;    // Embedded link to next hgrid object
    Point pos;              // x, y (and z) position for sphere (or top left AABB corner)
    float radius;           // Radius for bounding sphere (or width of AABB)
    int bucket;             // Index of hash bucket object is in
    int level;              // Grid level for the object
    ...                     // Object data
};
```

这里的网格单元定义为正方形形状（在 3D 层次网格中为立方体）。同时，某一特定层网格单元内部的最大球体直径与网格单元边的比率如下所示：

```
const float SPHERE_TO_CELL_RATIO = 1.0f/4.0f;   // Largest sphere in cell is 1/4*cell size
```

一般情况下，较小的比率值意味着在某一特定层中，需要检测的网格单元的数量也较少。但这也表明每一个网格单元中对象的平均数量将会增加。适当的比率值往往是与具体应用相关的。在算法的遍历过程中，网格单元的边尺寸可按下列因子实现增长：

```
const float CELL_TO_CELL_RATIO = 2.0f;          // Cells at next level are 2*side of current cell
```

因此，向层次网格中添加一个对象可按下列方式执行：

```
void AddObjectToHGrid(HGrid *grid, Object *obj)
{
    // Find lowest level where object fully fits inside cell, taking RATIO into account
    int level;
    float size = MIN_CELL_SIZE, diameter = 2.0f * obj->radius;
    for (level = 0; size * SPHERE_TO_CELL_RATIO < diameter; level++)
        size *= CELL_TO_CELL_RATIO;

    // Assert if object is larger than largest grid cell
    assert(level < HGRID_MAX_LEVELS);

    // Add object to grid square, and remember cell and level numbers,
    // treating level as a third dimension coordinate
    Cell cellPos((int)(obj->pos.x / size), (int)(obj->pos.y / size), level);
    int bucket = ComputeHashBucketIndex(cellPos);
    obj->bucket= bucket;
    obj->level = level;
    obj->pNextObject = grid->objectBucket[bucket];
    grid->objectBucket[bucket] = obj;

    // Mark this level as having one more object. Also indicate level is in use
    grid->objectsAtLevel[level]++;
```

```
    grid->occupiedLevelsMask |= (1 << level);
}
```

从层次网格中移除某一对象的相关操作为：

```
void RemoveObjectFromHGrid(HGrid *grid, Object *obj)
{
    // One less object on this grid level. Mark level as unused if no objects left
    if (--grid->objectsAtLevel[obj->level] == 0)
        grid->occupiedLevelsMask &=~(1 << obj->level);
    // Now scan through list and unlink object 'obj'
    int bucket= obj->bucket;
    Object *p = grid->objectBucket[bucket];
    // Special-case updating list header when object is first in list
    if (p == obj) {
        grid->objectBucket[bucket] = obj->pNextObject;
        return;
    }
    // Traverse rest of list, unlinking 'obj' when found
    while (p) {
        // Keep q as trailing pointer to previous element
        Object *q = p;
        p = p->pNextObject;
        if (p == obj) {
            q->pNextObject = p->pNextObject;        // unlink by bypassing
            return;
        }
    }
    assert(0);                                      // No such object in hgrid
}
```

在层次网格中进行对象间的碰撞测试时，假定对象的数量未知，这样将对网格结构中的全部层实行遍历操作（网格中应包含相关对象）。对应的测试代码如下所示：

```
// Test collisions between object and all objects in hgrid
void CheckObjAgainstGrid(HGrid *grid, Object *obj,
                         void (*pCallbackFunc)(Object *pA, Object *pB))
{
    float size = MIN_CELL_SIZE;
    int startLevel = 0;
    uint32 occupiedLevelsMask = grid->occupiedLevelsMask;
    Point pos = obj->pos;

    // If all objects are tested at the same time, the appropriate starting
    // grid level can be computed as:
    // float diameter = 2.0f * obj->radius;
    // for ( ; size * SPHERE_TO_CELL_RATIO < diameter; startLevel++){
        // size *= CELL_TO_CELL_RATIO;
        //occupiedLevelsMask >>= 1;
    //}
```

```
// For each new query, increase time stamp counter
grid->tick++;

for (int level = startLevel; level < HGRID_MAX_LEVELS;
                size *= CELL_TO_CELL_RATIO, occupiedLevelsMask >>= 1, level++) {
    // If no objects in rest of grid, stop now
    if (occupiedLevelsMask == 0) break;
    // If no objects at this level, go on to the next level
    if ((occupiedLevelsMask & 1) == 0) continue;

    // Compute ranges [x1..x2, y1..y2] of cells overlapped on this level. To
    // make sure objects in neighboring cells are tested, by increasing range by
    // the maximum object overlap: size * SPHERE_TO_CELL_RATIO
    float delta = obj->radius + size * SPHERE_TO_CELL_RATIO + EPSILON;
    float ooSize = 1.0f / size;
    int x1 = (int)floorf((pos.x - delta) * ooSize);
    int y1 = (int)floorf((pos.y - delta) * ooSize);
    int x2 = (int) ceilf((pos.x + delta) * ooSize);
    int y2 = (int) ceilf((pos.y + delta) * ooSize);

    // Check all the grid cells overlapped on current level
    for (int x = x1; x <= x2; x++) {
        for (int y = y1; y <= y2; y++) {
            // Treat level as a third dimension coordinate
            Cell cellPos(x, y, level);
            int bucket = ComputeHashBucketIndex(cellPos);

            // Has this hash bucket already been checked for this object?
            if (grid->timeStamp[bucket] == grid->tick) continue;
            grid->timeStamp[bucket] = grid->tick;

            // Loop through all objects in the bucket to find nearby objects
            Object *p = grid->objectBucket[bucket];
            while (p) {
                if (p != obj) {
                    float dist2 = Sqr(pos.x - p->pos.x) + Sqr(pos.y - p->pos.y);
                    if (dist2 <= Sqr(obj->radius + p->radius + EPSILON))
                        pCallbackFunc(obj, p);      // Close, call callback function
                }
                p = p->pNextObject;
            }
        }
    }
}// end for level
}
```

当采用回调函数处理两个碰撞对象的测试函数时，应注意该回调函数不应改变层次网格自身的数据结构。由于函数 **CheckObjAgainstGrid()**的操作位于网格数据结构的内部，对其数据结构擅自改动将会产生重入问题：轻则出现碰撞测试的丢失，重则导致测试过程的崩溃。

7.2.2 其他类型的层次网格表达方式

相对于其他方案，层次网格使测试过程获得了较大的性能改善，但该方案仍存在一个问题。当对象一次执行单一测试时，较大对象有可能占据层次结构底层的大量网格单元，并与其中的对象相交，这将产生较大的测试开销。[Mirtich96b, Mirtich97]针对这一问题提出了两种处理方案，其中，层次网格被称作层次空间哈希表。

第一种方案按尺寸降序方式插入对象，继而首先将对象插入至 k 层网格内，然后是 $k-1$ 层。通过这一限制，新插入的对象只可与其插入层以及更高层中的网格单元执行测试，因此无须进行低层网格单元的测试，由于基于插入顺序，其内容当前仍为空。同时，为了能够更好地处理对象的动态插入、删除操作，需要一个单独的数据结构以维护当前对象排序后的插入操作及其测试顺序，如图 7.9 所示。

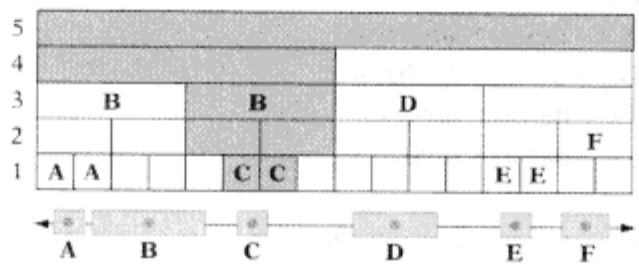


图 7.9 在 Mirtich（第一个）方案中，对象将插入至与当前插入层相交的所有网格单元中。如图 7.8 所示，深色网格单元表示在执行对象 C 的碰撞检测时需要测试的网格单元

在第二种方案中，对象将不再插入至某一层上的单一网格中，而是插入至其交叠的全部网格中，且测试将始于网格结构中的最低层，其中网格单元的尺寸足以容纳当前对象。最后，针对较高层中的全部交叠网格单元，重复执行这一步骤。

在第二种方案中，给定网格中的任一对象组合(A, B)，其相交测试转化为：在层 L=max($Level$(A), $Level$(B))处二者是否占据同一网格单元。亦即，二者是否相交于各自初始插入层中的较大层。

该方案的一个操作前提是测试组合已知，且需要单独记录组合信息。当对象 A、B 存储于同一网格单元中时，与组合(A, B)关联的计数值将加 1，同时，该组合还需要加入至对应的数据结构中以供记录。若其中任一对象移出二者所占据的公共网格单元，则上述计数值将减 1。当计数值为 0 时，该组合数据须从数据结构记录中移除，这也意味着二者的空间位置已渐行渐远。只有处于相关数据结构记录中的组合才可执行进一步的精确碰撞测试。此处，对象的移动开销将变得十分巨大，但基于空间一致性，算法很少对较大的网格单元实施更新操作。综上所述，该算法可视为开销较大的网格单元的更新操作与较少的网格单元测试数量之间的折中方案。

7.2.3 其他层次网格

就目前的结果来看，层次网格并不太擅长处理世界场景中的几何体，其中常包含大量的、分布不均匀的小型静态数据对象。由于密集网格的开销较大，因此表现最底层数据时常受限于相应的内存存储条件。类似地，当采用哈希表这一类稀疏表现方式时，若网格单元的相交计算量较大，则缓存失效现象以及其他某些计算都将会降低测试速度。因此，针对这一类特定问题，可考虑使用其他类型的层

次网格。相关算法多源于光线跟踪计算。

其中，最为直观的选择方案是递归网格算法（[Jevans89]），其均匀网格的构造过程将通过初始图元输入集加以实现。对于包含（至少）k 个图元的各网格单元，可通过递归方式构造一个高质量的网格。当网格单元包含的对象数量小于 m，或深度值小于 n（通常为 2 或 3）时，该递归构造过程结束。与前述层次网格相比，当前结果将使得最底层网格不再横跨场景世界，且采用密集型网格表达形式时，其内存占用量也将显著降低。

两种其他类型的替代方案分别为均匀网格层次结构（[Cazals95]）和适应性网格（[Klimaszewski97]）。两类方案均采用对象聚类算法对大型均匀网格内、对应子网格中的图元实现分组计算。更多与算法相关的信息和比较方案（以及其他算法）可参见[Havran99]。

7.3　树

在前述章节中，针对包围体层次结构应用环境，曾对树型结构进行过探讨。同时，树型结构对于空间划分也可构造良好的表达形式。本节将讨论两种空间划分方案：8 叉树和 k-d 树。

7.3.1　8 叉树（以及 4 叉树）

8 叉树可视为一类典型的、基于树型结构的空间划分算法，即它是一种 3D 世界场景空间体的层次结构划分方案。顾名思义，树中每一个父节点都包含 8 个子节点，且每一个节点都代表一个与该节点相关的体空间。其中，根节点为包含全部场景的最小轴对齐立方体。随后，该立方体将沿 x、y 和 z 轴等分且被划分为 8 个均等的子立方体（也称作网格单元体、8 分体或子块），如图 7.10 所示。这一类子立方体形成了该节点的子节点，并继续按上述方式对这些子节点递归子划分。当树型结构到达某一最大深度或立方体数量小于某一预定值，则递归构造过程结束。通过这种划分方式，树中每一个节点关联的空间体将涵盖全部派生的子孙节点。

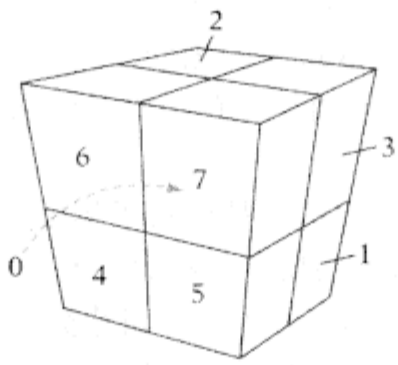


图 7.10　8 叉树根节点下的 8 个子节点序号

虽然节点的几何体边界尺寸在 8 叉树遍历时可向下传递并加以修正，但与节点相关的数据结构通常包含节点的位置以及相关尺寸——这将占据额外的内存空间，但却简化了相应的树型操作。另外，如果 8 个等子节点空间分割过程允许在 3 个轴上实现非对称划分，则需要将该体空间信息额外存储于这些子节点中。

```
// Octree node data structure
struct Node {
    Point center;          // Center point of octree node (not strictly needed)
    float halfWidth;       // Half the width of the node volume (not strictly needed)
    Node *pChild[8];       // Pointers to the eight children nodes
    Object *pObjList;      // Linked list of objects contained at this node
};
```

在 2D 环境中，与 8 叉树对应的层次结构为 4 叉树。此时，场景世界将被一个轴对齐的正方形所包围，并在每一次递归过程中被划分为 4 个较小的正方形。

正如二叉树可采用二维数组而非指针加以表示，4 叉树和 8 叉树亦是如此。若树节点存储于数组 **node[N]**中，则针对于某一父节点 **node[i]**，其 4 叉树子节点可表示为 **node[4*i+1]～node[4*i+4]**；8 叉树子节点为 **node[8*i+1]～node[8*i+8]**。这一类存储方式要求树型结构为完全树。

其中，n 层完全二叉树将包含 2^n-1 个节点；n 层完全 d 叉树则含有$(d^n-1)/(d-1)$个节点。采用这一类表达形式，一个 7 层的完全 8 叉树最多包含 300000 个节点，从而将多数树型结构限制在 5～6 层。

由于树型结构需要预分配内存空间以满足特定的层次深度，且结构间相对固定，因此数组形式适用于静态场景以及静态 8 叉树，而指针形式则更适用于需要对 8 叉树持续更新的动态场景。

7.3.2 8 叉树对象的分配

8 叉树可用于存储静态数据或动态数据。对于前者，相关数据可以是场景中的各个图元；对于后者，数据一般为场景中运动的相关实体对象。

针对静态数据存储，可采用自顶向下的方法直接构造 8 叉树。初始状态下，全部图元可视为与根节点体空间关联。在对根节点体空间实施划分后，图元集合将被重新划分至其相交的子单元体中，且若图元与多个子单元体相交，还需对其进行复制操作。该过程递归执行，直至子单元体包含的图元数量小于某一预设值，或图元全部置入子节点中（不存在子划分），递归操作结束。

这里将相关图元置入对应的子节点空间可按下列方式执行：首先通过 8 叉树 x 轴分割面划分图元，跨越图元都将被分为两部分，并占据分割面的两侧；随后，数据集合再根据 y 轴分割面实现类似的划分；最后，4 个结果集根据 z 轴分割面完成最终的划分。此时，8 个结果集可置入对应的子节点中。通过考查图元顶点是否位于分割面两侧，可知是否有必要对该图元实施分割操作。关于多边形图元的分割计算，其详细讨论可参见 8.3.4 节。

一类替代方案并未对相关图元进行划分，而是针对子体空间和图元执行 AABB-图元间的相交测试，以检测该图元是否划分至体空间中。该算法虽然从概念上易于理解，但执行起来颇为耗时。

为了避免多个 8 叉树节点间的几何体复制操作，可事先将原图元集合存储至数组中，图元的副本集合可传递至构造函数中。副本图元将实现对原图元的引用。当图元划分完毕后，两部分都将保持对原有图元的引用。随着树结构中叶节点的创建，将对原图元集合的引用加以存储，而非分割后的复制几何体。由于当前树型结构的静态特性，该过程最好在预处理计算时完成。

对于静态场景而言，对穿越相交 8 分体的图元实现复制操作是可行的。然而，该方法并不适用于动态对象——当对象运动时，需对树型结构进行更新操作。针对动态场景，一种较好的方法是将对象限定在最底层 8 叉树单元体中，当对象产生运动时，只需更新最小数量的指针信息即可。但该方法也

存在一个缺陷，即对象所处层可能与根据其尺寸得到的计算结果不符（较高）。特别地，在不考虑对象和节点空间相对尺寸的情况下（如图 7.11 所示），任何跨越分割面的对象都将会产生“阻塞”现象，这一问题将在后续小节中加以讨论。此处，仅设定 8 叉树结构处于动态变化中。

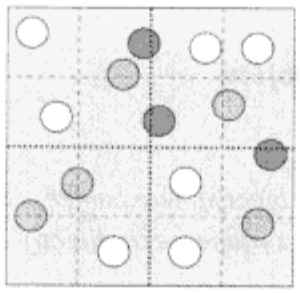


图 7.11　第一层子划分的 4 叉树节点采用点线表示；下一层子划分节点则采用虚线表示。其中，深灰色对象将与第一层分割面相交并呈现阻塞状态。浅灰色对象在向下扩展一层后也将呈现阻塞状态。这里，仅白色对象下降两层

初始状态下，8 叉树为空树并随着对象的插入而逐渐扩充。其中，可通过预分配操作降低树型动态增长所产生的开销。下列代码将根据特定的深度值，实现 8 叉树的预分配操作：

```
// Preallocates an octree down to a specific depth
Node *BuildOctree(Point center, float halfWidth, int stopDepth)
{
    if (stopDepth < 0) return NULL;
    else {
        // Construct and fill in 'root' of this subtree
        Node *pNode = new Node;
        pNode->center = center;
        pNode->halfWidth = halfWidth;
        pNode->pObjList = NULL;

        // Recursively construct the eight children of the subtree
        Point offset;
        float step = halfWidth * 0.5f;
        for (int i = 0; i < 8; i++) {
            offset.x = ((i & 1) ? step : -step);
            offset.y = ((i & 2) ? step : -step);
            offset.z = ((i & 4) ? step : -step);
            pNode->pChild[i] = BuildOctree(center + offset, step, stopDepth - 1);
        }
        return pNode;
    }
}
```

由于对象被限制在单一单元体中，因此可采用链表方案处理某一网格体内的多个对象，并将链表的链接指针内嵌于对象中。下列代码显示了当前对象的数据结构：

```
struct Object {
Point center;           // Center point for object
float radius;           // Radius of object bounding sphere
```

```
...
Object *pNextObject;        // Pointer to next object when linked into list
};
```

插入操作的代码如下所示：

```
void InsertObject(Node *pTree, Object *pObject)
{
    int index = 0, straddle = 0;
    // Compute the octant number [0..7] the object sphere center is in
    // If straddling any of the dividing x, y, or z planes, exit directly
    for (int i = 0; i < 3; i++) {
        float delta = pObject->center[i] - pTree->center[i];
        if (Abs(delta) < pTree->halfWidth + pObject->radius) {
            straddle = 1;
            break;
        }
        if (delta > 0.0f) index |= (1 << i);        // ZYX
    }
    if (!straddle && pTree->pChild[index]) {
        // Fully contained in existing child node; insert in that subtree
        InsertObject(pTree->pChild[index], pObject);
    } else {
        // Straddling, or no child node to descend into, so
        // link object into linked list at this node
        pObject->pNextObject = pTree->pObjList;
        pTree->pObjList = pObject;
    }
}
```

可对 **InsertObject()**函数稍加修改，一旦发现子节点指针为 NULL，即可分配一个新节点并链接至树中，并将对象插入至该新节点中。代码如下所示：

```
if (!straddle) {
    if (pTree->pChild[index] == NULL) {
        pTree->pChild[index] = new Node;
        ...initialize node contents here...
    }
    InsertObject(pTree->pChild[index], pObject);
} else {
        ...same as before...
}
```

若节点内不再包含对象，则执行相应的删除操作时需要保留一个节点内部的、用于访问其父节点的指针（或类似的机制）并用于相关的更新操作。可适当延迟新节点的构造过程，直至该阶段包含了 *n* 个对象。随后，该节点将被划分为 8 个子节点，相关对象也随之置入其中。当采取动态方式构建子节点时，构建 8 叉树所需的全部数据将形成一个初始根节点，其他节点将根据需要加以创建。动态 8 叉树的碰撞测试可采取自顶向下的递归方式完成，其中每一个非空节点内的对象都需要与该节点下方子

树中的全部对象执行测试。相关代码如下所示：

```
// Tests all objects that could possibly overlap due to cell ancestry and coexistence
// in the same cell. Assumes objects exist in a single cell only, and fully inside it
void TestAllCollisions(Node *pTree)
{
    // Keep track of all ancestor object lists in a stack
    const int MAX_DEPTH = 40;
    static Node *ancestorStack[MAX_DEPTH];
    static int depth = 0;          // 'Depth == 0' is invariant over calls

    // Check collision between all objects on this level and all
    // ancestor objects. The current level is included as its own
    // ancestor so all necessary pairwise tests are done
    ancestorStack[depth++] = pTree;
    for (int n = 0; n < depth; n++) {
        Object *pA, *pB;
        for (pA = ancestorStack[n]->pObjList; pA; pA = pA->pNextObject) {
            for (pB = pTree->pObjList; pB; pB = pB->pNextObject) {
                // Avoid testing both A->B and B->A
                if (pA == pB) break;
                // Now perform the collision test between pA and pB in some manner
                TestCollision(pA, pB);
            }
        }
    }

    // Recursively visit all existing children
    for (int i = 0; i < 8; i++)
        if (pTree->pChild[i])
            TestAllCollisions(pTree->pChild[i]);

    // Remove current node from ancestor stack before returning
    depth--;
}
```

7.3.3　位置码和 8 分体的定位

本节将计算 8 叉树所属空间内的某一查询点。这里，该查询点所处的叶节点 8 分体无须与 8 叉树节点坐标进行直接的比较计算。令查询点的坐标定义为浮点数 x、y、z 并映射至范围为[0…1]的二进制分式中。

例如，若 8 叉树中 x 的范围为[10…50]，则 x 坐标值 25 映射为(25−10)/(50−10)=0.375=0.01100000。通过该方式，x 坐标二进制分式中的最高有效位则可以确定下列事实：若为 0，该查询点位于通过 8 叉树中心点的 yz 平面的左侧；若为 1，则位于右侧。当综合考虑 x、y、z 向二进制分式中的最高有效位时，可得到与根节点的子节点相关的一系列序列值：0～7。类似地，次高有效位可用于确定查询点所

处节点的相应子节点。依此类推，直至到达叶节点。

因此，上述二进制分式中的交错位串可视为一个键值，即位置码，且可用于描述节点在树型结构中的相应位置。同时，以位置码显式的相关节点可称作 Morton 序（如图 7.12 所示）。

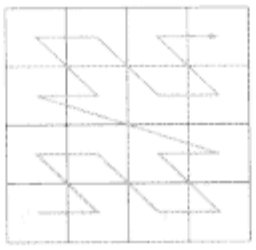


图 7.12　依据 Morton 序定义的 4 × 4 网格单元

当给定父节点的位置码后，即可方便地计算出其子节点的位置码 **childKey=(parentKey << 3) + childIndex**，即将父节点的位置码左移 3 位，再加上（或执行 OR 操作）子节点的索引号（0～7）。位置码非常适用于采用内存优化的 8 叉树结构。

7.3.4　基于哈希存储的线性树

对于均衡树，采用指针方式将比使用平面二维数组更加节省内存，但 8 叉树节点仍需存储 8 个子节点指针——这也构成了树型存储的主要内容。

另一种非指针型的选择方案是线性树。线性树只存储包含相关数据的 8 叉树节点，其中，各节点都将保存其自身的位置码。同时，父节点及其全部子节点的位置码可通过事先存储的位置码值加以计算。因此，8 叉树节点将不再需要指向子节点的指针，其结构将更加紧凑，如下所示：

```
// Octree node data structure (hashed)
struct Node {
    Point center;          // Center point of octree node (not strictly needed)
    int key;               // The location (Morton) code for this node
    int8   hasChildK;      // Bitmask indicating which eight children exist (optional)
    Object *pObjList;      // Linked list of objects contained at this node
};
```

节点的尺寸可以显式存储，或者也可源于位置码的深度值。对于后者，需在位置码中设定一个标志位以能够区分诸如 011 和 000000011 这一类位串，并分别将其转换为 1011 和 1000000011。相关代码如下所示：

```
int NodeDepth(unsigned int key)
{
    // Keep shifting off three bits at a time, increasing depth counter
    for (int d = 0; key; d++) {
        // If only sentinel bit remains, exit with node depth
        if (key == 1) return d;
        key >>= 3;
    }
    assert(0);          // Bad key
}
```

为了能够快速地访问由位置码定义的 8 叉树节点，可将节点的位置码作为键值并存储于哈希表中。其中，访问各节点的时间复杂度为 $O(1)$，而基于指针的树型结构为 $O(\log n)$。

另外，为了避免无效的哈希表查询操作，节点常采用位掩码用于标识其 8 个子节点的存在状态。下列代码显示了以线性树的方式访问当前节点，并演示了子节点的计算过程以及位掩码的使用方法。

```
void VisitLinearOctree(Node *pTree)
{
    // For all eight possible children
    for (int i = 0; i < 8; i++) {
        // See if the ith child exist
        if (pTree->hasChildK & (1 << i)) {
            // Compute new Morton key for the child
            int key = (pTree->key << 3) + i;
            // Using key, look child up in hash table and recursively visit subtree
            Node *pChild = HashTableLookup(gHashTable, key);
            VisitLinearOctree(pChild);
        }
    }
}
```

类似地，[Warren93]中描述了另一种哈希 8 叉树的实现方案。

7.3.5　计算 Morton 键

这里定义 8 叉树叶节点的（整数）坐标为 x、y、z，其对应的 Morton 键如下列函数所示：

```
// Takes three 10-bit numbers and bit-interleaves them into one number
uint32 Morton3(uint32 x, uint32 y, uint32 z)
{
    // z--z--z--z--z--z--z--z--z--z-- : Part1By2(z) << 2
    // -y--y--y--y--y--y--y--y--y--y- : Part1By2(y) << 1
    // --x--x--x--x--x--x--x--x--x--x : Part1By2(x)
    // zyxzyxzyxzyxzyxzyxzyxzyxzyxzyx : Final result
    return (Part1By2(z)  <<   2) + (Part1By2(y) << 1) + Part1By2(x);
}
```

辅助函数 **Part1By2()**用于对坐标值中的各个位实现分离操作，并在每一个原始数据位间插入两个 0 位。这里，通过移位和加法运算可实现中间结果的交替操作。函数 **Part1By2()**的代码如下所示：

```
// Separates low 10 bits of input by two bits
uint32 Part1By2(uint32 n)
{
    // n = ----------------------9876543210 : Bits initially
    // n = ------98----------------76543210 : After (1)
    // n = ------98--------7654--------3210 : After (2)
    // n = ------98----76----54----32----10 : After (3)
    // n = ----9--8--7--6--5--4--3--2--1--0 : After (4)
    n = (n^(n << 16)) & 0xff0000ff;          // (1)
```

```
    n = (n^(n << 8)) & 0x0300f00f;      // (2)
    n = (n^(n << 4)) & 0x030c30c3;      // (3)
    n = (n^(n << 2)) & 0x09249249;      // (4)
    return n;
}
```

在 2D 环境下（对应于 4 叉树），其等价代码如下所示：

```
// Takes two 16-bit numbers and bit-interleaves them into one number
uint32 Morton2(uint32 x, uint32 y)
{
    return (Part1By1(y) << 1) + Part1By1(x);
}

// Separates low 16 bits of input by one bit
uint32 Part1By1(uint32 n)
{
    // n = ----------fedcba9876543210 : Bits initially
    // n = --------fedcba98--------76543210 : After (1)
    // n = ----fedc----ba98----7654----3210 : After (2)
    // n = --fe--dc--ba--98--76--54--32--10 : After (3)
    // n = -f-e-d-c-b-a-9-8-7-6-5-4-3-2-1-0 : After (4)
    n = (n^(n << 8)) & 0x00ff00ff;      // (1)
    n = (n^(n << 4)) & 0x0f0f0f0f;      // (2)
    n = (n^(n << 2)) & 0x33333333;      // (3)
    n = (n^(n << 1)) & 0x55555555;      // (4)
    return n;
}
```

如果 CPU 的寄存器中包含足够的位，上述辅助函数可以实现并行处理。例如，若使用改进后的 **Part1By1()**函数针对 32 位数值进行计算，且最终产生一个 64 位的结果，前述 **Morton2()**可改写为：

```
uint32 Morton2(uint32 x, uint32 y)
{
    // Merge the two 16-bit inputs into one 32-bit value
    uint32 xy = (y << 16) + x;
    // Separate bits of 32-bit value by one, giving 64-bit value
    uint64 t = Part1By1_64BitOutput(xy);
    // Interleave the top bits (y) with the bottom bits (x)
    return (uint32)((t >> 31) + (t & 0x0fffffff));
}
```

综上所述，在 2D 环境下，处理 n 位输入数据时其中间结果需提供包含 $4n$ 位的相关数据。

7.3.6 松散 8 叉树

这里将再次考查动态 8 叉树中由于对象跨越分割面所产生的“阻塞”现象。一种有效的方案是适量地扩展节点体空间并使其部分相交（如图 7.13 所示）。因此，这种不严格的 8 叉树形式可称为松散

8 叉树（参见[Ulrich00]）。一般情况下，松散节点在 6 个方向上按其尺寸的一半进行扩展（但也可按任意范围值进行扩展）。这将使其空间较原尺寸增至 8 倍。

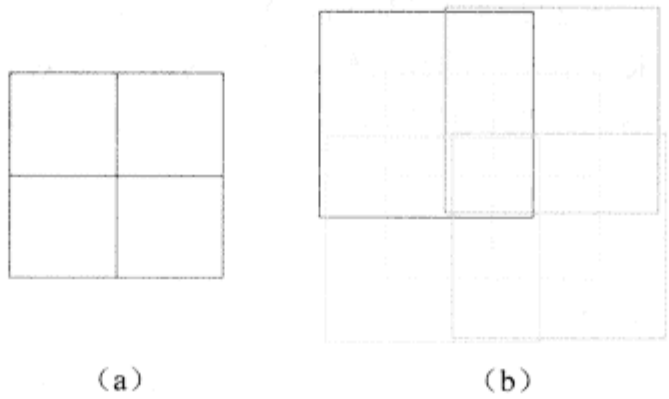


图 7.13　（a）标准 8 叉树的横截面，显示为 4 叉树。（b）在各方向上根据节点 1/2 宽度扩展 8 叉树节点，从而将原 8 叉树转换为松散 8 叉树（其中，松散节点的位置稍加偏移并用灰色加以标识，以更好地显示其边界。原 8 叉树节点则采用虚线表示）

这里，定义一些球体对象且对象被传递至子节点处并包含其球心位置。仅当该球体扩展至松散网格单元的外部，才可使其继续保持于当前层中。

相对于较大的节点，对象将在树型结构中降至较深的层次，使得划分过程具有较强的识别能力。但树中较高层中处于“阻塞”状态下的且经定位后的间距对象，却难以与其他对象进行测试。松散节点的另一个优点是对象的插入层次（深度）可依据其尺寸值计算得到，且插入操作的复杂度为 $O(1)$。

然而，上述特性也需付出一定的代价。较大的节点通常会与更多的邻居节点相交，继而节点的计算数量也会相应增加，以确定任意两组合对象间是否需要执行测试。在多数情况下，松散 8 叉树会降低碰撞检测过程中组合测试的计算量，这也是该方案较具吸引力的地方。

松散 8 叉树在结构上与层次网格较为相似，其主要差别在于松散 8 叉树具有根节点。通常情况下，对象远小于场景世界的尺寸，这也意味着，在获取对象所处的底层节点之前，大量时间都将花费在遍历树型结构的顶部空节点之上。相比较而言，层次网格则是一类浅层次的、无根节点的数据结构，并可实现对象节点的快速访问。

另外，层次网格可实现动态管理，该方案不会处理“空层”数据，即若当前层次中不包含相关对象，则计算即刻停止。基于线性或数组格式的 8 叉树可采用类似的优化方案。而对于指针 8 叉树，将维护相应子树每一个节点中对象的占有数量值。更新该数量值的计算开销将正比于树型结构的高度，其中，层次网格某层中占有量计数器更新操作的时间复杂度为 $O(1)$。由于哈希映射过程中采用了自动无限拼接技术，因此哈希层次网格无须已知场景世界的真实尺寸。然而 8 叉树并不适用于无穷大的场景世界。综上所述，针对碰撞检测应用情景，层次网格的表现要优于松散 8 叉树。

7.3.7　*k*-d 树

8 叉树和 4 叉树可一同视为 k 维树，即 k-d 树（参见[Bentley75]和[Friedman77]）。其中，k 代表子划分空间的数量，且无须匹配当前空间的维度。

与 4 叉树的二维、8 叉树的三维同步空间划分方案不同，*k*-d 树在一次操作过程中只沿一个方向进行划分。传统的方式一般是先沿 *x*、*y*、*z* 方向划分，然后再次在 *x* 向上划分，依次循环。但在 *k* 维空间中，往往可以任意选择划分轴——这使得划分轴和相关划分数据值可存储于 *k*-d 树的节点中。一种有效的方法是利用两个二进制位来显示相应的划分轴，详细内容可参见 13.4.1 节。8 叉树的一层可视为 3 层 *k*-d 树分别沿 *x*、*y*、*z* 向划分后的结果。图 7.14 显示了一棵 2D 空间 *k*-d 树的分解过程以及对应的 *k*-d 树布局。

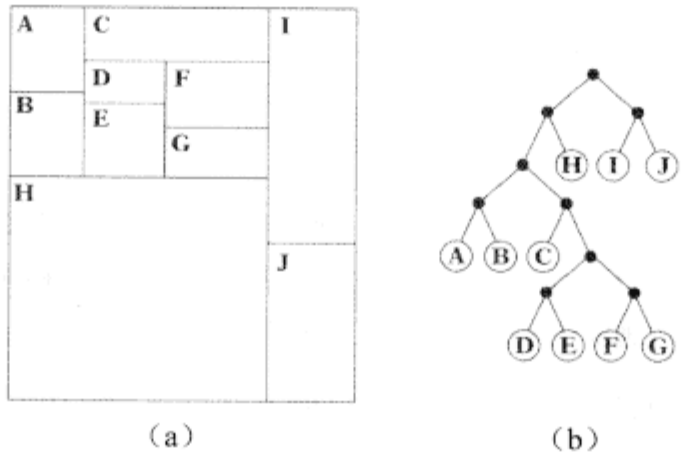


图 7.14　2D 环境下的 *k*-d 树型结构。（a）空间分解过程。（b）*k*-d 树的最终布局

每次沿单一轴向划分通常可产生比较简单的树型操作代码，其中只需考查单一平面的相交结果。进一步来讲，由于该平面与轴对齐，因此相对于该平面的测试也将具有较好的数值健壮性。

针对碰撞检测过程，4 叉树或 8 叉树适用的场合亦同样适用于 *k*-d 树。*k*-d 树的其他应用包括点定位计算（即给定一点，确定其所属区域）、最近邻接点计算（针对某一查询点，在点集中获取其最近点）以及范围搜索（在给定区域内计算其全部点）。

下列代码显示了与给定球体相交的 *k*-d 树中全部节点的访问方法。一个值得注意的问题是如何剔除位于球体外部的子树空间体，即使球体跨越多个体空间中的半空间支撑超平面。这里所采用的方法是在遍历过程中维护体空间内与球心最近的一点。其中，初始状态下，变量 volNearPt 定义为球心坐标。在递归遍历过程中，相对于分割面的较远一侧（即不包含球体的一侧），相关顶点将被分割面限定从而可确定体空间边界内的最近点。最后，volNearPt 与查询后的球心坐标之间的距离可用于子树的剔除操作。相关代码如下所示：

```
struct KDNode {
    KDNode *child[2];          // 0 = near, 1 = far
    int splitType;             // Which axis split is along (0, 1, 2, ...)
    float splitValue;          // Position of split along axis
    ...
};

// Visit k-d tree nodes overlapped by sphere. Call with volNearPt = s->c
void VisitOverlappedNodes(KDNode *pNode, Sphere *s, Point &volNearPt)
{
    if (pNode == NULL) return;
```

```
    // Visiting current node, perform work here
    ...

    // Figure out which child to recurse into first (0 = near, 1 = far)
    int first = s->c[pNode->splitType] > pNode->splitValue;

    // Always recurse into the subtree the sphere center is in
    VisitOverlappedNodes(pNode->child[first], s, volNearPt);

    // Update (by clamping) nearest point on volume when traversing far side.
    // Keep old value on the local stack so it can be restored later
    float oldValue = volNearPt[pNode->splitType];
    volNearPt[pNode->splitType] = pNode->splitValue;

    // If sphere overlaps the volume of the far node, recurse that subtree too
    if (SqDistPointPoint(volNearPt, s->c) < s->r * s->r)
        VisitOverlappedNodes(pNode->child[first∧      1], s, volNearPt);

    // Restore component of nearest pt on volume when returning
    volNearPt[pNode->splitType] = oldValue;
}
```

上述代码框架可以很容易地扩展至定位球体的最近邻接对象这一测试操作，该计算始于一个大尺寸的球体，且由于对象间在球体内部接触，因此需要对球体进行适当的收缩。需要注意的是，*k*-d 树中相关区域的形状将会影响到被访问节点的数量。例如，如果多个狭长区域彼此邻接，某些条件相对宽泛的查询测试很可能就会与该区域全部交叠。为了限制被访问节点的数量，最好在维度空间中避免使用这一类狭长的区域。另外，与这一类狭长区域相对的则是一类大尺寸区域。

关于 *k*-d 树的效率问题(针对于光线跟踪测试环境)将在[MacDonald90]中加以讨论。另外，[Samet90a]和[Samet90b]则对 8 叉树、*k*-d 树以及类似的空间数据结构进行了详细的分析。

7.3.8 混合方案

上述算法构成了各类混合策略的基本内容。例如，将对象或几何图元置入均匀网格后，各非空网格单元中的相关数据可进一步组织为某一树型结构。此时，则可通过网格快速定位某一（树林中）对应的树型结构以执行相关的后续测试（如图 7.15（a）所示）。

在单一的树型层次结构中，网格还可作为一类“加速结构”加以使用。例如，考查一个包含树节点指针的均匀网格，其中各指针均指向紧密包围对应网格单元的树节点（如图 7.15（b）所示）。因此，该方案在定位对象所处的相应节点时，无须在根节点处开始遍历——可先定位对象所处的网格单元，其指针将指向对应的树节点。在查询过程中采用这种方式将有助于避开大多数无关子树。

对于位于网格单元边界且与多个网格单元交叠的对象，可适当调整网格单元以使其相对稀疏，这一过程类似于松散 8 叉树中的节点。综上所述，包含对象的网格单元最终将被用作树型结构的查询入口点。

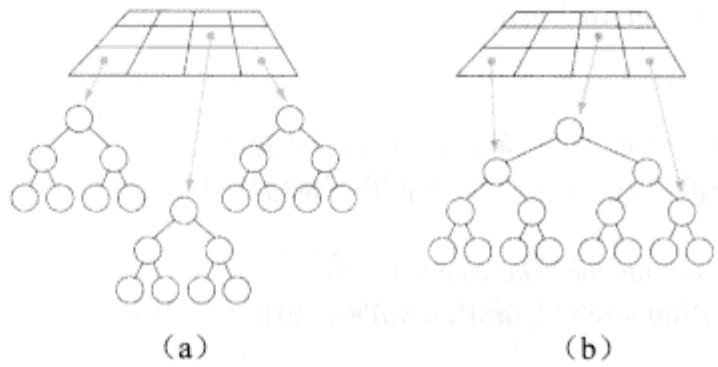


图 7.15 （a）树型网格，各网格单元均包含一棵树。（b）网格单元可索引至某一树型结构，每一网格单元均指向树型结构的起始遍历处

7.4 光线和有向线段的遍历

在多种应用程序特别是游戏程序中，直线拾取测试是一类十分常见的计算。通常，这一类查询将涉及光线或有向线段的计算。典型的应用场合包括视线测试以及子弹的碰撞测试。同时，该测试还可用于初步的碰撞查询计算。例如，可对车轮执行直线拾取测试，以将其定位在地面上。

为了加速这一类查询计算，可采用层次结构以减少光线或线段与对象间的相交测试数量。其核心理念即是将空间场景世界划分为多个网格单元。随后，光线通过其穿越的网格单元而被跟踪，并与该网格单元内部所包含的对象执行相交测试。下面将详细分析光线-*k*-d 树、光线-均匀网格间的查询测试。

7.4.1 *k*-d 树相交测试

针对光线或有向线段与 *k*-d 树之间的相交测试，其后的基本思想还是比较直观的。即执行线段 $S(t) = A + t\,\mathbf{d}$ 与节点分割面之间的相交测试，并计算相交后的 t 值。若 t 值位于线段区间 $0 \leqslant t < tmax$ 内部，则线段跨越该平面并对两个子节点进行递归测试；否则，只递归访问线段原点所处的子节点。

首先访问线段原点所处的子节点，相交的 *k*-d 树节点可保证按由近至远的顺序加以遍历。因此，若已获取了最近相交点，则测试过程即可提前结束，而无须再对其他节点进行测试。相关代码如下所示：

```
// Visit all k-d tree nodes intersected by segment S = a + t * d, 0 <= t < tmax
void VisitNodes(KDNode *pNode, Point a, Vector d, float tmax)
{
    if (pNode == NULL) return;

    // Visiting current node, perform actual work here
    ...

    // Figure out which child to recurse into first (0 = near, 1 = far)
    int dim = pNode->splitType;
    int first = a[dim] > pNode->splitValue;
```

```
    if (d[dim] == 0.0f) {
        // Segment parallel to splitting plane, visit near side only
        VisitNodes(pNode->child[first], a, d, tmax);
    } else {
        // Find t value for intersection between segment and split plane
        float t = (pNode->splitValue - a[dim]) / d[dim];

        // Test if line segment straddles splitting plane
        if (0.0f <= t && t < tmax) {
            // Yes, traverse near side first, then far side
            VisitNodes(pNode->child[first], a, d, t);
            VisitNodes(pNode->child[first^1], a + t * d, d, tmax - t);
        } else {
            // No, so just traverse near side
            VisitNodes(pNode->child[first], a, d, tmax);
        }
    }
}
```

如第 6 章中所述，上述函数可改写为非递归版本。另外，8 叉树与 BSP 树之间的遍历方式也基本上大同小异。同时，还可以将相关面直接链接至其所属的叶节点中，从而实现叶节点间的直接遍历。由于一个面可能位于多个叶节点中，与该面关联的 4 叉树或 8 叉树可对此面进行区域划分，并将各区域分别链接至对应的邻居节点中。

7.4.2　均匀网格的相交测试

均匀网格内的光线投射计算与 Bresenham 直线绘制算法中的光栅化过程十分相似（参见[Rogers98]）。然而，这其中仍存在一些重要的差别。在光栅化过程中，直线可与未被填充的网格单元相交（如图 7.16（a）所示）。而对于碰撞检测过程，这将产生严重的错误，全部跨越的网格单元都需要加以考查，否则在测试过程中将会遗漏某些对象。这里，需要沿直线方向对 6 个邻接的网格单元体进行遍历。在 3D 环境下，若网格单元体间存在某些共享面，则其连接度为 6（在 2D 环境下，连接度为 4）。如果考虑网格单元体间的共享边，则其连接度为 16。另外，如果两个网格单元体间存在共享面、共享边以及共享顶点，则其连接度为 26（在 2D 环境下为 8）。

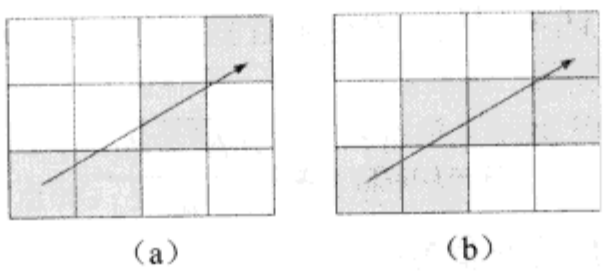


图 7.16　2D 直线的网格单元连接特征。（a）连接度为 8 的直线。（b）连接度为 4 的直线。在 3D 环境下，相应的连接度分别为 26 和 6

Bresenham 类算法可归纳为：沿长轴方向上步进（step），并在满足某一累积误差项时实现短轴上的步进。同时，该类算法可扩展为：在短轴上实现步进时，该操作可穿越某些遗漏的网格单元。相关

内容的详细解释可参见[Joy86]。

依据[Amanatides87]和[Cleary88]中的内容，本节中的相关算法略显不同。该算法呈现某种对称性且并不采取前述长轴步进方案，同时，此处的网格单元全部设定为正方形。在下述内容中，线段的含义为有向直线线段且在特定环境下光线也将采取这一定义。最后，为了简化表达，算法全部采取 2D 网格。

算法的核心思想是：光线在离开某一网格单元时，将穿越其边界。另外，在 x 轴或 y 轴上执行步进将取决于光线最终穿越了垂直边界还是水平边界。该方案可通过维护两个变量 tx 和 ty 加以实现，代表沿该光线、光线原点至邻接垂直边界或水平边界之间的距离。如果 tx 小于 ty，则光线将穿越垂直边界，且随后将与邻接的水平边界相遇。最终，算法将沿 x 轴左向（或右向）步进至邻接网格单元。

在步进过程中，tx 值将被更新以反映邻接垂直边界间的距离。对于均尺寸网格单元，邻接垂直边界间、沿光线上的距离值为一常数，因此 tx 上的变化值可通过预计算得到，记为 Δtx（如图 7.17（a）所示）。同理，邻接水平边界步进值记为常数 Δty（如图 7.17（b）所示）。

在图 7.17（c）中，当前网格单元记为(i,j)，由于 tx（光线原点至邻接垂直边界之间的距离）小于 ty，则下一个被访问的邻接网格单元为$(i+1,j)$。

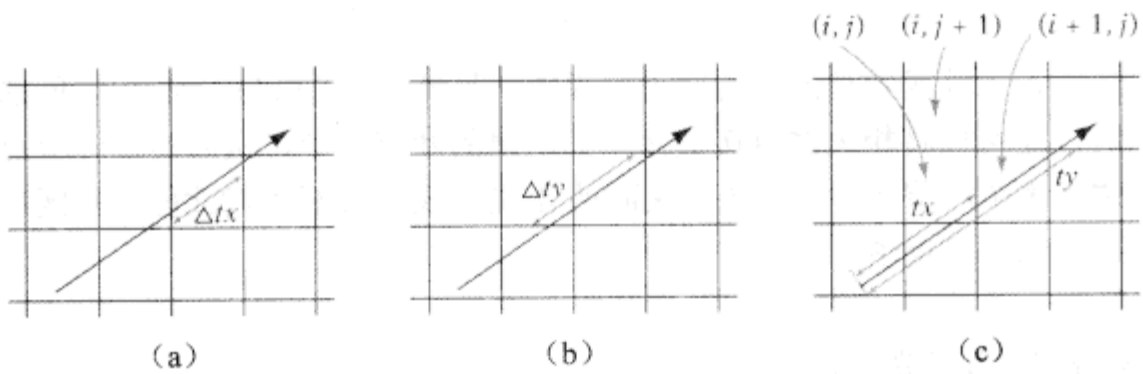


图 7.17　tx、ty、Δtx 以及 Δty 的计算方式。（a）Δtx 表示两垂直边界之间的距离。（b）Δty 表示两水平边界之间的距离。（c）针对网格单元(i,j)，水平边界间的距离 tx 小于垂直边界间的距离 ty，因而下一个受访网格单元为$(i+1,j)$

假定网格单元的边为 M 个单位宽度值，则针对线段$(x_1, y_1)-(x_2, y_2)$，$dx=|x_2-x_1|$，$dy=|y_2-y_1|$，Δtx 为：

$$\Delta tx = M\frac{\sqrt{dx^2+dy^2}}{dx}$$

这里，根据线段的方向 tx 和 ty 的计算过程稍有不同。针对光线的原点(x_1, y_1)及给定的 x 坐标（x_1），其左侧最近垂直边界值为 $minx=M\lfloor x_1/M\rfloor$；右侧最近垂直边界值为 $maxx=minx+M$。若光线方向指向右侧，则 tx 值按如下方式计算：

$$tx=(maxx-x_1)\frac{\sqrt{dx^2+dy^2}}{dx}$$

若光线方向指向左侧，则对应的 tx 值为：

$$tx=(x_1-minx)\frac{\sqrt{dx^2+dy^2}}{dx}$$

图 7.18 显示了该计算的操作过程。另外，初始 ty 值也可按类似的方式加以计算。

为了记录当前被访问的网格单元，首先定位光线原点所处的网格单元(i, j)，随后计算光线的方向

(*di*,*dj*)。针对右向光线，方向定义为(1,1)，左向光线则为(−1,0)。当光线穿越网格单元的边界时，*x* 轴或 *y* 轴的步进 *di* 或 *dj* 将取决于是垂直边界还是水平边界。对于一条线段，提前计算其终点网格单元的坐标将有助于降低遍历开销。

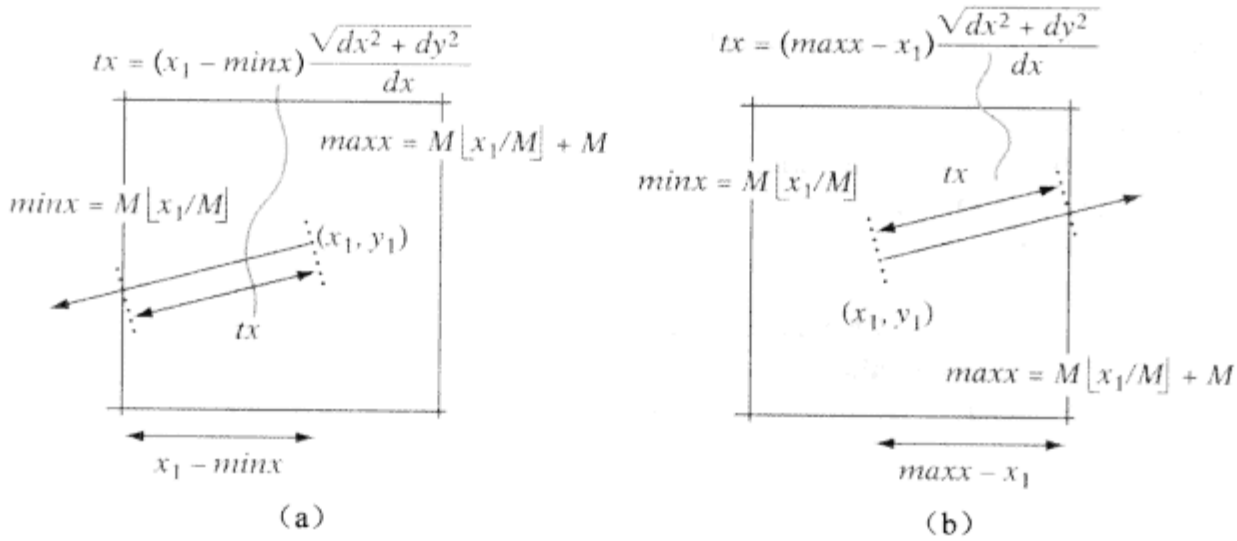


图 7.18　计算 *tx* 的初始值（*ty* 同理）。（a）左向光线。（b）右向光线

下列代码实现了全网格遍历操作。注意，由于 $\sqrt{dx^2 + dy^2}$ 项为常数且分别出现在 *tx*、*ty*、Δ*tx* 和 Δ*ty* 中，因而可推迟计算其结果，从而极大地简化了初始化计算过程。

```
void VisitCellsOverlapped(float x1, float y1, float x2, float y2)
{
    // Side dimensions of the square cell
    const float CELL_SIDE = 30.0f;

    // Determine start grid cell coordinates (i, j)
    int i = (int)(x1 / CELL_SIDE);
    int j = (int)(y1 / CELL_SIDE);
    // Determine end grid cell coordinates (iend, jend)
    int iend = (int)(x2 / CELL_SIDE);
    int jend = (int)(y2 / CELL_SIDE);

    // Determine in which primary direction to step
    int di = ((x1 < x2) ? 1 : ((x1 > x2) ? -1 : 0));
    int dj = ((y1 < y2) ? 1 : ((y1 > y2) ? -1 : 0));

    // Determine tx and ty, the values of t at which the directed segment
    // (x1,y1)-(x2,y2) crosses the first horizontal and vertical cell
    // boundaries, respectively. Min(tx, ty) indicates how far one can
    // travel along the segment and still remain in the current cell
    float minx = CELL_SIDE * floorf(x1/CELL_SIDE), maxx = minx + CELL_SIDE;
    float tx = ((x1 < x2) ? (x1 - minx) : (maxx - x1)) / Abs(x2 - x1);
    float miny = CELL_SIDE * floorf(y1/CELL_SIDE), maxy = miny + CELL_SIDE;
    float ty = ((y1 < y2) ? (y1 - miny) : (maxy - y1)) / Abs(y2 - y1);

    // Determine deltax/deltay, how far (in units of t) one must step
```

```
    // along the directed line segment for the horizontal/vertical
    // movement (respectively) to equal the width/height of a cell
    float deltatx = CELL_SIDE / Abs(x2 - x1);
    float deltaty = CELL_SIDE / Abs(y2 - y1);

    // Main loop. Visits cells until last cell reached
    for (;;) {
        VisitCell(i, j);
        if (tx <= ty) {          // tx smallest, step in x
            if (i == iend) break;
            tx += deltatx;
            i += di;
        } else {                 // ty smallest, step in y
            if (j == jend) break;
            ty += deltaty;
            j += dj;
        }
    }
}
```

这里可以不采用 i、j 索引定位 **grid[i][j]**，而是维护指向当前网格单元的一个指针，且增加该指针值以实现指针的步进操作（而非将相应索引增加 di、dj）。

所述算法可用于 3D 网格的遍历计算，此时，主循环结构则相应地变为：

```
for (;;) {
    VisitCell(i, j, k);
    if (tx <= ty && tx <= tz) {          // tx smallest, step in x
        if (i == iend) break;
        tx += deltatx;
        i += di;
    } else if (ty <= tx && ty <= tz) {   // ty smallest, step in y
        if (j == jend) break;
        ty += deltaty;
        j += dj;
    } else {                             // tz smallest, step in z
        if (k == kend) break;
        tz += deltatz;
        k += dk;
    }
}
```

这里需要注意的是，由于 tx 和 ty 为 float 类型变量，因而舍入误差将使得遍历过程中某些网格单元不同于采用精确计算所获取的访问结果。如果这一问题的确值得关注，则对象在置入网格之前可考虑增加一个较小的 ε 值。另外，算法也可全部采用整型变量，即仅使用整型输入数据并将 tx、ty、Δtx、Δty 乘以初始化代码中的相关分母项。[Barerra03]中描述了一种全整型的、连接度为 4 的直线绘制算法。

若存储于网格内的对象为凹对象，则应确保交点位于当前网格单元内部。另外，若凹对象跨越多个网格单元，则该对象将与多个网格单元碰撞，且应与某一网格单元内部中的对象发生碰撞。7.7 节中

提供了相关算法，以避免针对同一对象执行多次相交测试。

对于均匀网格的光线查询计算，存在多种可能的优化方案。例如，以一定角度入射的光线将比轴对齐的光线穿越更多的网格单元。因此，一类可行的优化方案是令多个网格呈现不同的方向，并选取最具对齐特征的某一网格以实现光线追踪。另一类优化方案是针对某一网格单元内的对象采取前移预排序启发策略。由于当前相交对象很可能在下一次查询中仍然相交，因此针对最近相交对象设定一个距离上界值将有助于对同一网格单元内部的较远对象实施基于距离测试的快速剔除操作。其他类型的优化方案可参考[Glassner89]、[Shirley00]等与光线跟踪技术相关的参考文献。

7.5 排序扫掠算法

在将对象插入至网格或 8 叉树形式的既定子空间划分区域时，跨越多个划分区域的相关对象往往会导致计算量的增加。一类可行的方案是在排序后的空间排列中维护相应对象。这里，一种与之对应的空间排序对象方案称为排序扫掠算法[Baraff92]（在[Cohen95]中记作扫掠剪除算法）。

首先考查在 1D 包围盒中的相交检测状况。其中，盒体对应于坐标轴上的某一区间[b, e]。给定 n 个此类区间，全部 b、e 值可插入至一个排序表中（如图 7.19 所示）。通过扫掠该表，可记录有效区间：遇到 b 值时，将其加入到一个新表中；遇到 e 值时，则将其从中移除。当一个新区间加入表中时，可知其将与表中已有有效区间交叠。

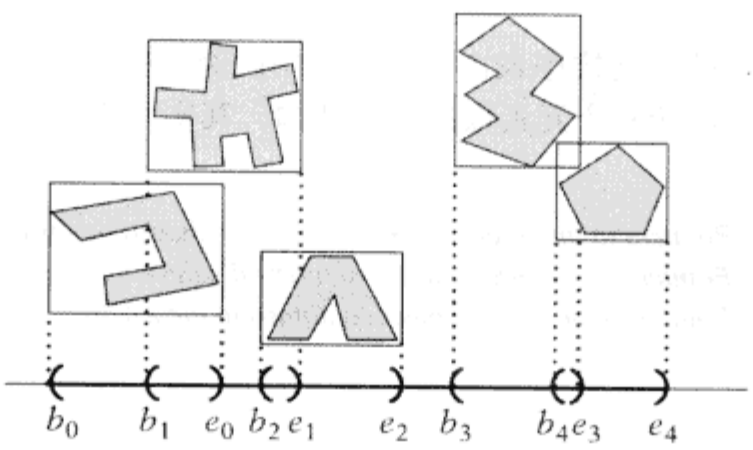


图 7.19 x 轴上的 AABB 的投影区间

这里，通过考查 x、y、z 轴上的投影区间，3D 环境下的 AABB 可由 3 个彼此独立的此类区间加以确定。因此，针对每一个坐标轴，都可采取一个排序链表用以维护全部对象对应 AABB 投影区间的首尾值。通过查询邻接对象，某一单独对象碰撞查询的时间复杂度将接近常量。同时，只有位于被测对象投影区间内的相关对象才需执行检测。最后，此类链表还适用于对象移动后的动态更新操作。

当相关表排序后，且通常情况下对象在帧间的移动范围并不大，因此表中相关内容也将在帧间保持有序。为了充分发挥时间一致性，通常在帧间对相关表加以维护，并采用简单的线性插入排序法使其有序。

为了在排序过程中能够实现快速更新操作，需要使用双向链表。表头和表尾的哨兵节点将有助于简化代码的实现并加速其执行过程（[Knuth98]）。

总体而言，更新包含 n 个对象的表其预期时间复杂度为 $O(n)$。然而，对象间的聚类效应将会破坏上述一致性特征。特别地，对象在世界场景内的较小位移量将导致表内元素大范围的位置移动，排序操作使得聚类轴上的时间复杂度变为 $O(n^2)$。这一类情况十分常见。例如，将全部对象皆置于地面上——这将导致垂直轴向上产生过于紧密的聚类效应（如图 7.20 所示）。

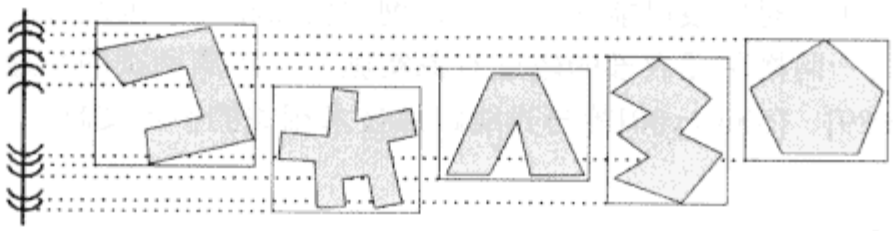


图 7.20　y 轴上的聚类对象（下落对象置于地面上）。较小的对象位移将导致聚类轴向上表内元素的巨大变化

针对上述聚类对象，一类解决方案是：y 轴只负责考查 x 轴和 z 轴向上的各表。多数情况下，仅需在两个轴向上实施排序即可，同时，该方案还可以适当地减少内存开销。在某些情况下，甚至可以选择更具意义的轴向实施排序，而非常规状态下的 x、y、z 轴。例如，若对象在 x 轴或 y 轴上趋于聚集时，可考虑使用 $x=y$ 轴。针对某些特定应用，这虽然可以在一定程度上排除聚类效应所产生的问题，但仍会存在某些遗留问题有待处理。7.5.2 节将会给出一类替代方案以避免复杂度为 $O(n^2)$这一最坏情况出现。

7.5.1　排序链表实现方案

为了维护投影区间范围内的轴向排序表，可考虑采用两类数据结构：对应最小区间值和最大区间值的链表元素结构，以及分别链接两个元素项的另一结构。其实现方式可简单地描述如下：

```
struct AABB {
    Elem *pMin[3];      // Pointers to the three minimum interval values (one for each axis)
    Elem *pMax[3];      // Pointers to the three maximum interval values (one for each axis)
    Object *pObj;       // Pointer to the actual object contained in the AABB
};

struct Elem {
    AABB *pAABB;        // Back pointer to AABB object (to find matching max/min element)
    Elem *pLeft;        // Pointer to the previous linked list element
    Elem *pRight;       // Pointer to the next linked list element
    float value;        // The actual min or max coordinate value
    int   minmax:1;     // A min value or a max value?
};
```

这里，采用 3 个表头指针对当前表加以定义：

```
Elem *gListHead[3];
```

可将区间最小值和最大值合并为一个元素以节省内存空间（仅当需要一个指向 AABB 对象的回调指针时）。另外，合并后的结构还需要一个标志位，以标明其包含的坐标值是最小值还是最大值。该结构如下所示：

```
struct AABB {
    Elem *pMin;     // Pointer to element containing the three minimum interval values
    Elem *pMax;     // Pointer to element containing the three maximum interval values
    Object*pObj;    // Pointer to the actual object contained in the AABB
};

struct Elem {
    AABB *pAABB;        // Back pointer to AABB object (to find matching max/min element)
    Elem *pLeft[3];     // Pointers to the previous linked list element (one for each axis)
    Elem *pRight[3];    // Pointers to the next linked list element (one for each axis)
    float value[3];     // All min or all max coordinate values (one for each axis)
    int   minmax:1;     // All min values or all max values?
};
```

由于 **Elem** 结构自身固定（只是其所含指针重链接而已），因此该结构可连续地置入内存区域，从而使其得到进一步优化。当给定一个指向 **Elem** 结构的指针后，即可得到相应的 min/max 值。

由于 AABB 结构只需链接 **Elem** 结构，因而其中的 **Elem** 结构对可视为新的 **AABB** 结构。而之前的 **AABB** 结构和 **Elem** 结构都需要单独分配内存空间，目前，**AABB** 结构包含了所有必需的数据，所以只需对单一结构分配内存空间。

```
struct AABB {
    Elem min;           // Element containing the three minimum interval values
    Elem max;           // Element containing the three maximum interval values
    Object *pObj;       // Pointer to the actual object contained in the AABB
};

struct Elem {
    Elem *pLeft[3];     // Pointers to the previous linked list element (one for each axis)
    Elem *pRight[3];    // Pointers to the next linked list element (one for each axis)
    float value[3];     // All min or all max coordinate values (one for each axis)
    int   minmax:1;     // All min values or all max values?
};
```

当给定一个指向 **Elem** 结构的指针后，其对应的 AABB 可按如下方式分配空间：

```
AABB *GetAABB(Elem *pElem)
{
    return (AABB *)(pElem->minmax ? (pElem - 1) : pElem);
}
```

与前述表达方式相比，该方案将节省 4 个指针变量（每一个指针 4 个字节，共 16 个字节）。通过将相应结构内嵌于对象内部，还可除去指向对象的指针变量并再次节省 4 个字节。另外，依靠相关技巧，**minmax** 位域还可从 **Elem** 结构中移至 **AABB** 结构中，从而进一步节省内存空间。实际上，这一类数据皆可内嵌于某一结构域中，并且不会占据额外的内存空间。

可以注意到，若每一个对象占用 76～84 个字节，这仍将是一类开销较大的数据结构类型。这里可将 32 位的指针类型变量转化为 16 位的索引值。可以肯定的是，对于任何类型的应用而言，相关对象的数量都不会超过 65536 个。

下列代码假定 3 个表已分别初始化完毕，并包含首、尾哨兵数据。其中，哨兵数据可按下列方式建立：

```
enum {
    MIN_ELEM = 0,                // Indicates AABB minx, miny, or minz element
    MAX_ELEM = 1                 // Indicates AABB maxx, maxy, or maxz element
};

// Initialize the lists, with start and end sentinels
AABB *pSentinel = new AABB;
for (int i = 0; i < 3; i++) {
    pSentinel->min.pLeft[i] = NULL;      // not strictly needed
    pSentinel->min.pRight[i] = &pSentinel->max;
    pSentinel->max.pLeft[i] = &pSentinel->min;
    pSentinel->max.pRight[i] = NULL;     // not strictly needed
    pSentinel->min.value[i] = -FLT_MAX;
    pSentinel->max.value[i] = FLT_MAX;
    gListHead[i] = &pSentinel->min;
}
// Note backwardness of initializing these two
pSentinel->min.minmax = MAX_ELEM;
pSentinel->max.minmax = MIN_ELEM;
```

其中，可通过扫描各表并获取相应的插入位置，从而实现 AABB 相对于排序表的插入操作。为了简化计算，组合相交状态将在 AABB 插入完毕后再执行测试计算。另外，组合相交状态更新的同时对象可通过交错执行例程完成相应的插入操作，这将使得代码的运行更加高效（仅凭在 min、max 元素间检测 **Elem** 结构不足以确定全部组合相交状态，可能会遗漏某些较大 AABB 间的测试，并完全包含该 AABB）。

为了记录碰撞测试的活动状态，下列代码使用了 **AddCollisionPair()**函数和 **DeleteCollisionPair()**函数，分别用以注册/注销组合对象间的碰撞状态。

```
void InsertAABBIntoList(AABB *pAABB)
{
    // For all three axes
    for (int i = 0; i < 3; i++) {
        // Search from start of list
        Elem *pElem = gListHead[i];

        // Insert min cell at position where pElem points to first larger element.
        // Assumes large sentinel value guards from falling off end of list
        while (pElem->value[i] < pAABB->min.value[i])
            pElem = pElem->pRight[i];
        pAABB->min.pLeft[i] = pElem->pLeft[i];
        pAABB->min.pRight[i] = pElem;
        pElem->pLeft[i]->pRight[i] = &pAABB->min;
        pElem->pLeft[i] = &pAABB->min;

        // Insert max cell in the same way. Can continue searching from last
```

```
        // position as list is sorted. Also assumes sentinel value present
        while (pElem->value[i] < pAABB->max.value[i])
            pElem = pElem->pRight[i];
        pAABB->max.pLeft[i] = pElem->pLeft[i];
        pAABB->max.pRight[i] = pElem;
        pElem->pLeft[i]->pRight[i] = &pAABB->max;
        pElem->pLeft[i] = &pAABB->max;
    }

    // Now scan through list and add overlap pairs for all objects that
    // this AABB intersects. This pair tracking could be incorporated into
    // the loops above, but is not done here to simplify the code
    for (Elem *pElem = gListHead[0]; ; ) {
        if (pElem->minmax == MIN_ELEM) {
            if (pElem->value[0] > pAABB->max.value[0])
                break;
            if (AABBOverlap(pAABB, GetAABB(pElem)))
                AddCollisionPair(pAABB, GetAABB(pElem));
        } else if (pElem->value[0] < pAABB->min.value[0])
            break;
    }
}
```

当 AABB 的位置发生变化时，应该对各相关表进行更新：

```
// This updating code assumes all other elements of list are sorted
void UpdateAABBPosition(AABB *pAABB)
{
    // For all three axes
    for (int i = 0; i < 3; i++) {
        Elem *pMin = &pAABB->min, *pMax = &pAABB->max, *t;

        // Try to move min element to the left. Move the roaming pointer t left
        // for as long as it points to elem with value larger than pMin's. While
        // doing so, keep track of the update status of any AABBs passed over
        for (t = pMin->pLeft[i]; pMin->value[i] < t->value[i]; t = t->pLeft[i])
            if (t->minmax == MAX_ELEM)
                if (AABBOverlap(pAABB, GetAABB(t)))
                    if (!HasCollisionPair(pAABB, GetAABB(t)))
                        AddCollisionPair(pAABB, GetAABB(t));
        // If t moves from its original position, move pMin into new place
        if (t != pMin->pLeft[i])
            MoveElement(i, pMin, t);
        // Similarly to above, try to move max element to the right
        for (t = pMax->pRight[i]; pMax->value[i] > t->value[i]; t = t->pRight[i])
            if (t->minmax == MIN_ELEM)
                if (AABBOverlap(pAABB, GetAABB(t)))
                    if (!HasCollisionPair(pAABB, GetAABB(t)))
                        AddCollisionPair(pAABB, GetAABB(t));
        if (t != pMax->pRight[i])
```

```
            MoveElement(i, pMax, t->pLeft[i]);

        // Similarly to above, try to move min element to the right
        for (t = pMin->pRight[i]; pMin->value[i] > t->value[i]; t = t->pRight[i])
            if (t->minmax == MAX_ELEM)
                if (HasCollisionPair(pAABB, GetAABB(t)))
                    DeleteCollisionPair(pAABB, GetAABB(t));
        if (t != pMin->pRight[i])
            MoveElement(i, pMin, t->pLeft[i]);

        // Similarly to above, try to move max element to the left
        for (t = pMax->pLeft[i]; pMax->value[i] < t->value[i]; t = t->pLeft[i])
            if (t->minmax == MIN_ELEM)
                if (HasCollisionPair(pAABB, GetAABB(t)))
                    DeleteCollisionPair(pAABB, GetAABB(t));
        if (t != pMax->pLeft[i])
        MoveElement(i, pMax, t);
    }
}
```

下列函数显示了如何在表内移动 AABB 元素：

```
void MoveElement(int i, Elem *pElem, Elem *pDest)
{
    // Unlink element...
    pElem->pLeft[i]->pRight[i] = pElem->pRight[i];
    pElem->pRight[i]->pLeft[i] = pElem->pLeft[i];
    // ...and relink it _after_ the destination element
    pElem->pLeft[i] = pDest;
    pElem->pRight[i] = pDest->pRight[i];
    pDest->pRight[i]->pLeft[i] = pElem;
    pDest->pRight[i] = pElem;
}
```

针对排序扫掠算法的链表格式实现方案，上述代码给出了其操作的主框架。

7.5.2 基于数组的排序

链表方案的最终效果通常具有两面性。首先，若内存状况允许，该方案可处理大量的对象。其次，当对象固定或少量对象移动时，几乎不需要在帧间维护相关的数据结构。但各对象 AABB 的较大内存开销，以及聚类效应导致的 $O(n^2)$时间复杂度，都可视为该方案的致命缺陷。

一类替代算法是使用数组实现轴向排序。当需要动态地处理多个对象时，数组往往更具灵活性且占据较少的内存空间。采用数组时，即使只是某一对象产生移动也需要更新全部数组内容，因而更需要对其实现全排序操作。同时，聚类问题可得到彻底的根除——计算性能将取决于所采用的排序算法（如复杂度为 $O(n\log n)$的快速排序算法，或复杂度为 $O(kn)$的基数排序算法）。不同于表类型，即使全部对象产生移动，抑或对象移动了较远的距离，相应的更新操作复杂度也将保持一个常量。采用数组

格式将会简化代码的实现，并提供一类缓存友好的内存访问方案。下列代码将针对指向对象 AABB 的指针数组进行操作：

```
struct AABB {
    Point min;
    Point max;
...
};
AABB *gAABBArray[MAX_OBJECTS];
```

针对排序扫掠算法中的每一步，都必须选择某一坐标轴作为排序轴向（具体选择方法稍后描述），并通过一个全局标量加以标识：

```
int gSortAxis = 0;    // Specifies axis (0/1/2) to sort on (here arbitrarily initialized)
```

当确定了排序轴之后，基于所引 AABB 的最小值，相应指针将以升序加以排序。为了简化表达，下列代码使用了 stdlib 库中的 **qsort()**函数。**qsort()**函数中所需的用于排序 AABB 的比较函数如下所示：

```
// Comparison function for qsort. Given two arguments A and B must return a
// value of less than zero if A < B, zero if A = B, and greater than zero if A > B
int cmpAABBs(const void *a, const void *b)
{
    // Sort on minimum value along either x, y, or z (specified in gSortAxis)
    float minA = (*(AABB **)a)->min[gSortAxis];
    float minB = (*(AABB **)b)->min[gSortAxis];
    if (minA < minB) return -1;
    if (minA > minB) return 1;
    return 0;
}
```

待数组排序完毕后，对其加以扫描以考查相交盒体。在数组扫描过程中，相交检测位置在当前盒体位置后处开始。代码如下所示：

```
void SortAndSweepAABBArray(void)
{
    // Sort the array on currently selected sorting axis (gSortAxis)
    qsort(gAABBArray, MAX_OBJECTS, sizeof(AABB *), cmpAABBs);

    // Sweep the array for collisions
    float s[3] = { 0.0f, 0.0f, 0.0f }, s2[3] = { 0.0f, 0.0f, 0.0f }, v[3];
    for (int i = 0; i < MAX_OBJECTS; i++) {
        // Determine AABB center point
        Point p = 0.5f * (gAABBArray[i]->min + gAABBArray[i]->max);
        // Update sum and sum2 for computing variance of AABB centers
        for (int c = 0; c < 3; c++) {
            s[c] += p[c];
            s2[c] += p[c] * p[c];
        }
        // Test collisions against all possible overlapping AABBs following current one
```

```
        for (int j = i + 1; j < MAX_OBJECTS; j++) {
            // Stop when tested AABBs are beyond the end of current AABB
            if (gAABBArray[j]->min[gSortAxis] > gAABBArray[i]->max[gSortAxis])
                break;
            if (AABBOverlap(gAABBArray[i], gAABBArray[j]))
                TestCollision(gAABBArray[i], gAABBArray[j]);
        }
    }

    // Compute variance (less a, for comparison unnecessary, constant factor)
    for (int c = 0; c < 3; c++)
        v[c] = s2[c] - s[c] * s[c] / MAX_OBJECTS;

    // Update axis sorted to be the one with greatest AABB variance
    gSortAxis = 0;
    if (v[1] > v[0]) gSortAxis = 1;
    if (v[2] > v[gSortAxis]) gSortAxis = 2;
}
```

在扫掠过程中，将计算沿各轴向上的 AABB 中心坐标的方差值。待扫掠步骤完毕后，即得到具有最大方差值的某一轴向以备例程下一次调用时使用。该算法仍无法完全避免某些特殊情况，例如，多个维度上的聚类效应仍将使算法的复杂度降至 $O(n^2)$。

这里，在数组结尾位置处添加新的 AABB 数据，插入操作的复杂度将为 $O(1)$。同时，相关排序步骤亦将会缓解定位 AABB 正确位置时的计算开销。另外，当移除 AABB 时，其对应指针也将从数组中移除。当给定一个 AABB 后，针对数组的二分搜索将使得定位其指针的时间复杂度为 $O(\log n)$。数组中尾指针将会取代被删除的指针。如果删除操作较为频繁，可在扫掠步骤中写入 AABB 结构内的指向数组指针的回调指针，这也将使得删除操作的时间复杂度为 $O(1)$。

7.6 网格单元和伪入口

碰撞检测过程不必一定要采取空间划分策略。有时，针对用于渲染世界场景几何体的层次组织结构加以分析即可，如场景图。组织结构与碰撞检测结合得较好的一类实例是网格单元-伪入口算法（[Teller92]和[Luebke95]）。

算法源于建筑结构漫游系统的渲染。该系统可视为一类极具复杂度的、包含大量遮挡物的几何环境。然而，与视锥体范围内的全部几何体相比，一般建筑场景中可视几何体的数量仍占少数。网格单元-伪入口算法通过分析场景间的这种遮挡特征，从而将可渲染几何体的数量降至最低。算法将场景世界划分为不同的区域（网格单元），并确定其彼此连接的边界（伪入口）。例如，场景中的房间可视为网格单元，而门口、窗户则定义为伪入口。总之，伪入口定义了网格单元之间的连接性质，并直接或间接地确定了某一给定网格单元所能触及的其他可视网格单元（如图 7.21 所示）。

在渲染已划分为网格单元和伪入口的场景时，需要首先绘制包含相机的网格单元中的几何体，待该网格单元渲染完毕后，再递归调用渲染函数以绘制伪入口-相机可见的邻接网格单元。在递归调用过

程中，对于新出现的伪入口，其视见量将被当前伪入口剪裁，并不断地缩小场景中的视线范围。最终，当某一新增伪入口相对于相机不可见，或邻接网格单元全部访问完毕时，将结束递归调用过程。

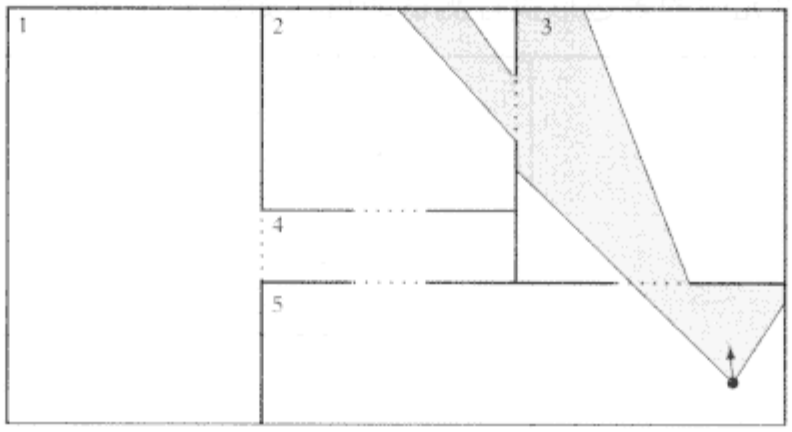


图 7.21　由 5 个网格单元（标记为 1～5）构成的简单场景世界包含 5 个伪入口（虚线）。阴影区域表示为视角可见范围。因此，只需渲染网格单元 2、3 和 5

伪入口一般采用非自身相交的多边形。通常情况下，网格单元多采用凸多面体加以定义。基于凸体特征，网格单元中的多边形可按任何顺序加以渲染，且彼此间不会产生交叠，因而也省略了多边形排序这一步骤（目前的硬件设备一般都具备 Z-缓冲能力，因而网格单元凸体特性将不再是必备条件）。下列伪代码显示了渲染操作的实现过程：

```
RenderCell(ClipRegion r, Cell *c)
{
    // If the cell has not already been visited this frame...
    if (c->lastFrameVisited != currentFrameNumber) {
        // ...timestamp it to make sure it is not visited several
        // times due to multiple traversal paths through the cells
        c->lastFrameVisited = currentFrameNumber;
        // Recursively visit all connected cells with visible portals
        for (Portal *pl = c->pPortalList; pl != NULL; pl = pl->pNextPortal) {
            // Clip the portal region against the current clipping region
            ClipRegion visiblePart = ProjectAndIntersectRegion(r, pl->boundary);
            // If portal is not completely clipped its contents must be partially
            // visible, so recursively render other side through the reduced portal
            if (!EmptyRegion(visiblePart))
                RenderCell(visiblePart, pl->pAdjoiningCell);
        }
        // Now render all polygons (done last, for back-to-front rendering)
        for (Polygon *p = c.pPolygonList; p != NULL; p = p->pNextPolygon)
            RenderPolygon(p);
    }
}
```

该网格单元-伪入口算法可用于碰撞检测查询的优化操作。初始状态下，对象将与包含其中心点坐标的网格单元进行关联。当对象产生移动时，光线测试将用于确定该对象是否已超出当前网格单元的范围。若是，则对连接伪入口执行测试，以计算对象将移至哪一个邻接网格单元中。

针对对象-对象间的查询操作，给定对象 A 后，仅需在 A 所处网格单元中的对象与 A 相交的伪入口所属的邻接网格单元中的对象间执行测试；而对于对象-场景间的碰撞查询，仅需将当前网格单元内的多边形与对象相交的网格单元中的多边形执行测试即可（如图 7.22 所示）。

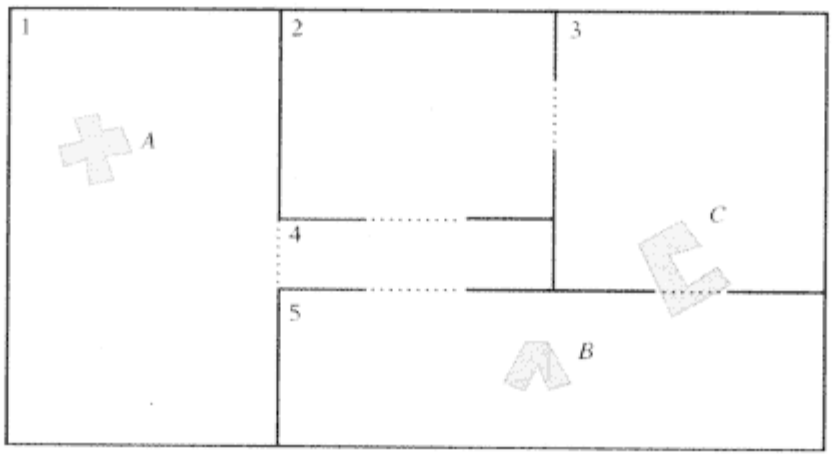


图 7.22　与对象 A 相交的网格单元中不存在任何对象，因而对象 A 无须执行对象间的测试。这里，由于对象 C 跨越网格单元 3 和 5 之间的伪入口并部分位于对象 B 的网格单元中，因此对象 B 和 C 彼此间须进行测试

在第 8 章中，将采用 BSP 树将场景世界划分为相应的凸体区域，且最终的划分结果也可用于碰撞检测操作——这类似于网格单元-伪入口算法。实际上，常采用 BSP 树自动生成网格单元与伪入口。

7.7　避免重复测试

考查下列划分方案，其中单一对象与多个划分区域相交，这将产生下列潜在问题：相同的两个对象可能会彼此测试多次。当组合测试的计算开销较大时，这一类问题若不加以正确地处理，相应的碰撞检测代码将对相关对象运行多次。类似的问题也会出现在光线跟踪计算中：穿越多个网格单元的入射光线将会与跨越多个网格单元的某一对象执行多次测试。从中可以看出，对同一对象多次执行相同的测试将会浪费大量的计算时间。下面将针对具体情况提供相应的解决方案。

7.7.1　位标志

在处理重复测试这一类问题时，一类较为直接的方法是采用位标志，用以标识已执行完毕的相交测试。例如，当入射光线穿越某一网格时，可将各对象均分配一个标志位；对于对象-对象间的相交测试，可将各组合对象关联一个标志位。如果该标志位被设定，则表明相关对象（或组合对象）均已测试完毕，且无须执行进一步检测；若该标志位未被设定，则须执行全测试操作，且待测试完毕后设置该标志位以表明测试完成。

在对象-对象碰撞测试中，位标志可存储于数组中并由该组合对象的索引加以定位。其中，数组元素项(a,b)、(b,a)将视为同一组合。另外，为了节省内存空间，只需存储对角线上的三角形元素即可。因此，针对 n 个对象间的最大组合值，数组仅需存储 $n(n-1)/2$ 个位，而非 n^2。这里，给定两个对象索

引 a 和 b 且 $a < b$，则相应位的索引值可按下列方式计算：

$$bitindex = a(2n - a - 3)/2 + b - 1$$

而非下式：

$$bitindex = an + b$$

下列代码显示了对象-对象间相交测试时位标志数组的实现方式：

```
// Allocate enough words to hold a bit flag for each object pair
const int32 MAX_OBJECTS = 1000;
const int32 MAX_OBJECT_PAIRS = MAX_OBJECTS * (MAX_OBJECTS – 1) / 2;
int32 bitflags[(MAX_OBJECT_PAIRS + 31) / 32];
...
void TestObjectPair(int32 index0, int32 index1)
{
    assert(index0 != index1);
    // Find which object index is smaller and which is larger
    int32 min = index0, max = index1;
    if (index1 < index0) {
        min = index1;
        max = index0;
    }
    // Compute index of bit representing the object pair in the array
    int32 bitindex = min * (2 * MAX_OBJECTS - min - 3) / 2 + max - 1;
    // Look up the corresponding bit in the appropriate word
    int32 mask = 1L << (bitindex & 31);
    if ((bitflags[bitindex >> 5] & mask) == 0) {
        // Bit not set, so pair has not already been processed;
        // process object pair for intersection now
        ...
        // Finally mark object pair as processed
        bitflags[bitindex >> 5] |= mask;
    }
}
```

对于对象-对象间的测试，该算法运行良好；但对于光线投射计算仍不够理想。在对象-对象测试方案中，标志位数组每帧都需要重置一次——这尚可接受。然而对于光线投射计算，标志位数组则需要对各测试光线进行重置。在典型的游戏程序中，每帧可能需要执行数百条光线或类光线测试。即使对象的数量在可接受的数量范围内，这一类测试的计算开销也将变得十分巨大。

7.7.2　时间戳

针对光线投射计算，一类较好的、记录已测试对象的替代方案是维护一个计数器，且在每次光线查询计算过程中增加计数值，并将该计数值（记作时间戳）待对象测试完毕后赋予其中。随后，在相关对象测试之前，都需要检测其时间戳以查看该对象在当前帧中的测试状态，以决定其最终的取舍。

图 7.23 显示了时间戳的工作方式。首先，考查光线与对象 A 之间的交互操作。在网格单元 1 中，光线将针对对象 A 执行测试。如图所示，由于二者间不相交，对象 A 仅标记为该光线的时间戳，且光

线继续处理邻接网格单元。在网格单元 2 中，光线再次与对象 *A* 进行测试，由于对象的时间戳已标记为当前时间，因此可忽略其测试。

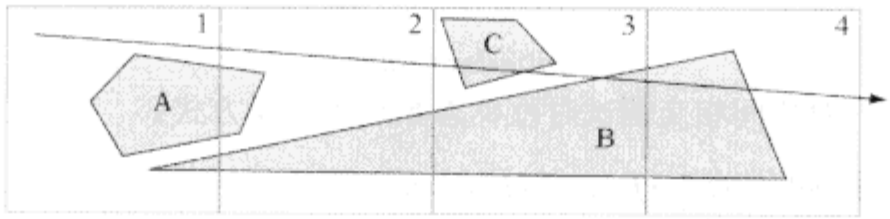


图 7.23　采用时间戳以避免光线与网格单元 1 和 2 中的对象 *A* 相交两次。这里，将通过均匀网格对光线实现跟踪计算。可将与对象 *B* 的相交数据存储至邮箱中，并在网格单元 3 中加以复用且无须进行重复计算

下面考查光线与对象 *B* 之间的测试，且再次从网格单元 1 处开始。如图所示，光线与对象 *B* 在网格单元 3 中相交。由于交点不在当前网格单元中，因此很可能还存在其他交点（网格单元 3 中的对象 *C*）。因此，光线处理过程需要继续考查后续网格单元。在移至邻接网格单元之前，对象 *B* 的相交状态需要与当前已存储的“最优”相交数据进行比较，以确定其是否为最近交点。这里，对象 *B* 恰好为第一个相交对象，因此，一个指向对象 *B* 的指针以及二者当前的碰撞时刻记为“最优”相交状态。在对象 *B* 最终采用时间戳进行标记之前，还应对光线查询过程中所遍历的网格单元数量加以更新。如图所示，由于已知在网格单元 3 中存在交点，则无须再处理后续网格单元 4。在网格单元 2 中，对象 *A* 和 *B* 再次执行测试，由于二者皆以时间戳加以标记，因此可省略当前测试且继续处理网格单元 3。在网格单元 3 中，再次对对象 *B* 测试：由于对象 *B* 已由时间戳标记，该测试可省略。同时，光线还将与对象 *C* 进行测试且二者相交。由于对象 *C* 的交点近于当前已存储的对象 *B* 的交点，则“最优”相交状态更新为对象 *C* 的交点。此时，网格单元 3 中已无其他测试对象，同时，光线测试数据业已设定完毕且测试止于该网格单元处。在对各对象均执行了一次相交测试之后，查询过程最终返回与对象 *C* 的相交数据。

此处，仍有一些细节问题有待考查。若不考虑时间戳的尺寸，则其最终结果将会出现上溢。如果计数器分配到足够的位数（一般在 64 位左右），则该问题即可迎刃而解。假设当前使用 8 位计数器，则每隔 256 个时钟，该时间戳计数器将会出现上溢现象，且全部已存储的时间戳数据都将被重置。否则，在 256 个时钟内，某一未被时间戳标记的突发测试对象将在当前时间片内被错误地报告为“已测试”。

这里所阐述的时间戳技术在光线跟踪计算中则称为“邮箱”算法（[Amanatides87]），而另一种消除重复相交测试以及冗余计算的相应方案则称为“驻留掩码”（residency mask）算法（[Cychosz92]）。

在多数应用中，时间戳的重置问题往往被忽略。但对于包含大量对象的应用场合而言，该操作通常比较耗时。针对这一类应用，无须一次重置全部时间戳数据值，具体内容将在 7.7.3 节中描述。

7.7.3　分时清除时间戳

当采用分时更新时间戳方法时，将会显著降低单次重置时间戳数据的计算开销。这里，当全局计数器出现上溢时，并不一次性地清除计数器中的全部内容，全部时间戳计数器集合将被划分为多个较小的数据块，且每帧中仅对某单一数据块进行重置。

下面通过一个简单的实例对分时算法加以演示（参见表 7.2）。考查 n（$n = 10$）个对象且对象被划分为 3 个数据块，分别包含 3、3、4 个对象。初始状态下，全部对象的时间戳皆为 0。在第一帧中，全局时间戳计数器增至 1。待全部对象更新完毕后，针对每一个数据块其包含对象的时间戳数据为 0 或 1。在结束第一帧之前，数据块 0 被重置为全局时间戳计数器的当前值。

该过程在后续帧中重复执行，且对象的下一个数据块更新至对应帧的全局计数器的当前值。考查表 7.2，需要注意的是，在第 4 帧的结尾处将不再出现值为 1 的时间戳对象，其值定为全局计数器在第 5 帧中的相应值。类似地，在第 5 帧的结尾处也将不再出现值为 2 的时间戳对象。对于后续帧，依此类推。

表 7.2　3 个数据块中的对象在不同时刻的时间戳数据。对象清零时的时间戳其值均不等于当前全局时间戳。因此，即使各帧中包含多个清零对象，也不会出现时间戳冲突这一现象

帧	计　数　器	数据块 0	数据块 1	数据块 2	说　　明
0	0	0	0	0	初始状态下，清零全部对象的时间戳数据
1	1	0/1	0/1	0/1	增加全局计数器以更新对象
		1	0/1	0/1	清除（对象时间戳所在的）数据块 0～1
2	2	1/2	0/1/2	0/1/2	增加全局计数器以更新对象
		1/2	2	0/1/2	清除数据块 1～2
3	3	1/2/3	2/3	0/1/2/3	增加全局计数器以更新对象
		1/2/3	2/3	3	清除数据块 2～3
4	0	0/1/2/3	0/2/3	0/3	增加全局计数器以更新对象
		0	0/2/3	0/3	清零数据块 0；当前数据块 1 中无时间戳对象
5	1	0/1	0/1/2/3	0/1/3	增加全局计数器以更新对象
		0/1	1	0/1/3	将数据块 1 设定为 1；当前数据块 2 中无时间戳对象
6	2	0/1/2	1/2	0/1/2/3	增加全局计数器以更新对象
		0/1/2	1/2	2	将数据块 2 设定为 2；当前数据块 3 中无时间戳对象

为了进一步演示分时清除算法，下列代码显示了该算法实现的一个框架。首先，定义下列常量值：

```
// Use 8 bits for the time stamp counter [0..255]
#define MAX_COUNTER_VALUE 255

// Divide up all time-stamped objects into as many blocks
#define NUM_BLOCKS MAX_COUNTER_VALUE
```

初始状态下，将全部对象的时间戳计数器清零：

```
blockToClear = 0;
tickCounter = 0;
for (i = 0; i < MAX_OBJECTS; i++)
    object[i].timeStamp = tickCounter;
```

每一帧中，时间戳计数器的正确使用方法如下所示：

```
// Increment the global time stamp counter
tickCounter++;
```

```
// Do any and all object testing required for the frame
for (i = 0; i < MAX_OBJECTS; i++) {
    if (object[i].timeStamp == tickCounter) {
        // Already processed this object this frame, do nothing
        continue;
    } else {
        // Process object for intersection here
        ...
        // Mark object as processed
        object[i].timeStamp = tickCounter;
    }
}
```

在每一帧的结尾处（或光线投射计算中的各光线间）全部存储后，时间戳数据的子集将更新为当前时间戳数据值。

```
// Reset the time stamp for all objects in the current block to be cleared
from = blockToClear * MAX_OBJECTS / MAX_COUNTER_VALUE;
to = (blockToClear + 1) * MAX_OBJECTS / MAX_COUNTER_VALUE;
for (i = from; i < to; i++)
    object[i].timeStamp = tickCounter;
// Indicate that the next block should be cleared the next frame
if (++blockToClear >= NUM_BLOCKS)
    blockToClear = 0;
// Wrap the global time stamp counter when it exceeds its maximum value
if (tickCounter >= MAX_COUNTER_VALUE - 1)
    tickCounter = 0;
```

同一技术也可应用于其他缓冲区的分时清除操作。例如，通过预留时间戳数据中的若干位，可在帧间对（软件）Z-缓冲区实现分布式重置操作。

7.8 小结

空间划分是一类可加速粗略测试过程的计算方法，空间划分为邻接区域并将场景对象与其交叠的全部区域关联。随后，仅对位于同一区域的对象执行组合测试。在某些情况下，相关对象只是占据某单一区域，但仍需考查其邻接区域以确定潜在的碰撞可能性。

本章探讨了 3 种空间划分算法：网格、树以及空间排序。其中，网格通常定义为均匀网格，即包含均尺寸的网格单元。另外，大型网格将消耗大量的内存空间。一类降低网格内存空间的有效方案是采用哈希存储，如 7.1.3 节所述，该方案可使得网格的范围增至无穷大。网格也可隐式地加以表达，特别地，隐式位网格可视为一类高效的空间划分算法。由于尺寸差异较大的对象使得均匀网格中的网格单元难以设定，本章还专门讨论了层次网格结构。

一类树型表达方式是 8 叉树。8 叉树将空间划分为 8 个均尺寸的网格单元，并按同一方式针对上述网格单元执行递归划分。考虑到其规则的结构特征，8 叉树可采用线性存储并避免使用指针表示树型节点的链接操作，进而节省了相应的内存空间。松散 8 叉树则针对下列问题加以讨论：对象因跨越分割

面而无法正确地实现对树型结构的向下遍历（只可从某一8叉树网格单元体引用）。

k-d 树则是8叉树等树型结构的通用表达方式，其中，*k*-d 树具有与8叉树等同的空间划分结果，反之则不成立。*k*-d 树在任意位置并每次沿某一轴向将空间划分为两部分。*k*-d 树的结构简洁性使得其上的查询操作变得简单有效。

空间排序策略在一个或多个轴向上的有序结构中对全部对象进行维护。当相关对象产生移动时，将对该有序结构进行逐步更新。由于该结构呈有序状态，因此通过检测邻接对象且无须进行搜索计算即可实现对象的碰撞检测操作。

本章中还对树型结构进行了相应的补充，即通用的 *k*-d 树表达方式。该方案也可作为相应的空间划分算法加以使用。

第 8 章　BSP 树层次结构

在众多的空间划分算法中，BSP 树可视为最为通用的一类算法。该算法可达到与 k-d 树、4 叉树以及 8 叉树异曲同工的效果，反之则不成立。除了空间划分操作，BSP 树还可用于任意多面体场景的边界或实体表达。另外，BSP 树还是一类执行碰撞检测的强有力工具。

8.1　BSP 树

二分空间划分树（简称 BSP 树）是一类二叉树结构，它采用任意位置、方向的分割面递归地将空间划分为多个子空间对。对于 n 维空间，其分割面则为$(n-1)$维超平面。在具体应用过程中，工作空间通常为 3D 空间，有时为 2D 空间。对于 3D 空间，分割超平面体现为平面；在 2D 空间中，分割超平面则为一条直线。相应地，空间划分为两个半空间：正半空间和负半空间，且正半空间位于分割超平面的正面；负半空间则位于分割超平面的背面（参见 3.6 节）。

下面考查一个 BSP 树的示例，如图 8.1 所示。由灰线边界定义的正方形为当前空间（需注意的是，该空间无须使用边界线加以限定，正方形很容易扩展为一个平面）。首先，超平面 A 将原空间划分为两个区域，如图 8.1（a）所示。正方形下方为其对应的 BSP 树，其中，内部节点采用圆形表示，叶节点采用方形表示。图 8.1（b）显示了第 2 个超平面 B 与节点 A 负空间之间的划分过程。最终，第 3 个超平面 C 针对节点 A 的正半空间实施划分，如图 8.1（c）所示。这里需要注意的是，与叶节点对应的空间区域皆为凸体。

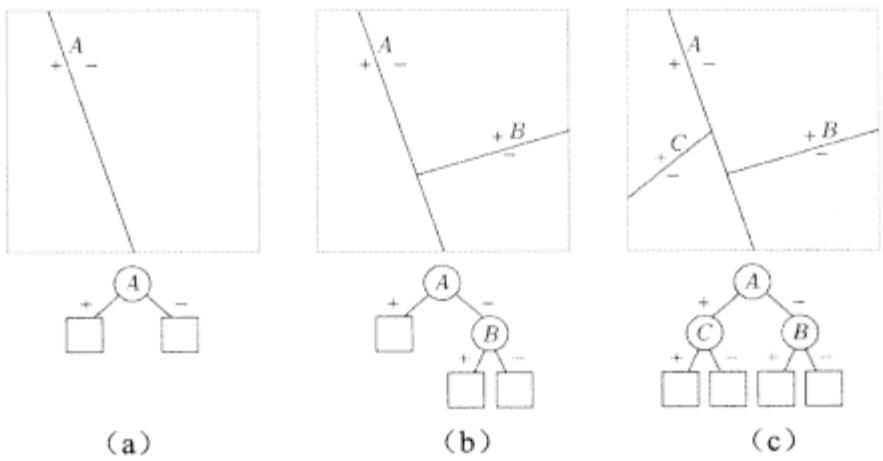


图 8.1　将正方形分割为 4 个凸体子空间及其 BSP 的构造过程。（a）初始分割。（b）针对树型结构，执行第 2 层分割。（c）继续执行第 2 层分割

最初，BSP 树用于解决与隐藏表面相关的某些问题（[Fuchs80]）。该方案在预处理阶段运用独立视角将场景世界加以分解，并在运行期内对结果树实施遍历，从而可从任意视点获取正确（前序或后

序）的对象或多边形顺序。

BSP 树的应用场合包括：光线跟踪技术、几何实体的构造（CSG）以及机器人运动和路径规划。在处理碰撞检测问题时，BSP 树的通用性还表现在：既可作为一类空间划分方案（无边界表达方式），又可作为一种体空间的表达方式（实体对象的有界表达方式）。作为一类空间划分方案，BSP 树与 4 叉树、8 叉树以及 *k*-d 树十分类似，但 BSP 树对空间数据结构的良好模拟性，使其应用范围变得更加广泛。另外，作为一种体空间表达方式，BSP 树还可用作多边形和多面体内部空间的表达、判别方法。

图 8.2 显示了 BSP 树空间划分和体空间表达的具体应用。在图 8.2（a）中，空间被相应地划分并可用于加速空间对象间的碰撞查询计算。基于 BSP 树的层次结构特性，针对树型层次结构中的 n 个对象，一般仅须测试 $O(\log n)$个对象——相对于分割面，可剔除一半测试对象（但某些退化情况将导致全部 n 个对象都被测试。例如，查询对象的尺寸较大或 BSP 树缺乏良好的结构特征）。

图 8.2（b）采用了对象表面轮廓线的分割面选择方式，进而区分其内、外范围。最终，与树型结构各叶节点关联的空间区域则位于轮廓形状的内部或外部。

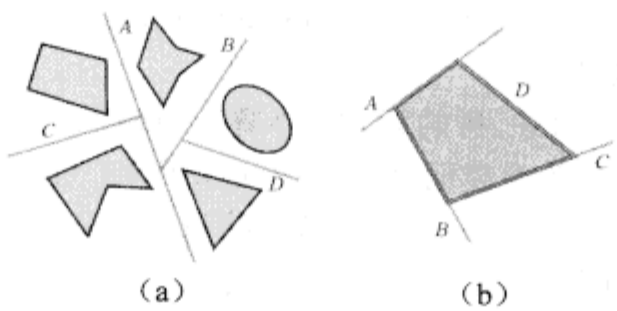


图 8.2　（a）针对多个对象的空间递归二分方案。（b）体空间或对象边界的空间递归二分方案

当采取输入几何体表面轮廓线作为分割面时，一般称作“自动划分”（有时也记作多边形对齐划分）；而与 *xy*、*xz* 或 *yz* 平面对齐的分割面划分方式常称为轴对齐划分。这里，仅采用轴对齐分割面的 BSP 树一般也视为 *k*-d 树（参见第 7 章）。通常，若对分割面（及其划分结果）不采取某种限制，则可称之为任意分割面或通用分割面。

BSP 树的构建或部分更新过程通常都比较耗时，因而很少在运行期内对其进行构造或更新。基于此，BSP 树一般用于存储静态场景几何体数据，而动态对象的碰撞检测操作则不在此列。

8.2　BSP 树的类型

BSP 树可按下列几种方式进行分类：分割面的定位方式、几何体的存储方式以及树型结构的应用范围（空间划分或体空间表达）。特别地，如下几何体的存储方式对应 3 类 BSP 树：节点存储、叶节点存储以及实体叶节点存储。

8.2.1　采用节点存储的 BSP 树

采用节点存储（基于节点）的 BSP 树属于“自动划分”这一范畴，因而可选择几何体面上的相应支撑平面作为构建过程中的分割面。顾名思义，当某一面的支撑平面被选作分割面时，该面和与其共

面的全部平面都存储于节点中。其余面则置入分割面相应的某一侧，以供递归构造该节点的子树使用。这里，任何跨越面都将相对于分割面进行分割计算，随后，其分割结果将再次作为输入数据并向下传递。采用节点存储的树型结构，其叶节点可以为空（全部多边形已存储于节点中）或仅包含一个面（共面的平面）。由于相关数据面已存储于节点中，因而无须再将分割面也存储于节点中（且分割面也源于上述已存储的数据面）。

一般情况下，采用节点存储的树型结构主要用于 3D 环境下的软件渲染——将面置入节点中可降低其分割次数。随着 3D 图形硬件的发展，采用节点存储的树型结构不再具有优势，该结构在几何体划分为独立多边形的过程中常导致过度分割。当前，高效的 3D 渲染硬件负责处理大型的三角带或面片，从而减少了图元的状态变化（该变化将导致较大的计算开销），并有效地降低了发往图形卡的顶点数量。上述变化是无法在采用节点存储的 BSP 树型结构中体现出来的。

另外，采用节点存储的树型结构也不适用于碰撞检测。针对某一查询对象的具体位置，存储于节点中的各面并不存在相应的空间位置关系，因而很可能未曾与当前查询对象相交。然而，由于查询对象将遍历与上述各面关联的分割面，对这一类面仍需要执行测试。通常，某些结构场景（具备显著的共面特征）将饱受这一类过度测试之苦。

图 8.3 显示了构造节点存储 BSP 树的第一个步骤。原输入几何体数据由 12 个多边形构成，如图 8.3（a）所示。最初的分割面选为穿越面 *A* 的平面（如图 8.3（b）所示）。由于 *G* 也位于该分割面之上，因此 *A*、*G* 存储于根节点中。另外，*D* 和 *J* 分别跨越当前分割面，因而需对其进行分割操作。此后，各面皆位于当前分割面的相应一侧，并可将其单独置入相关子树中。针对树型结构下一层中的左子树，穿越 *B* 的平面被选作分割面；而对于右子树，穿越 *H* 的平面作为分割面。图 8.3（c）显示了分割完毕后的 BSP 树。后续构造方式以类似方式执行，直至满足某一结束条件。

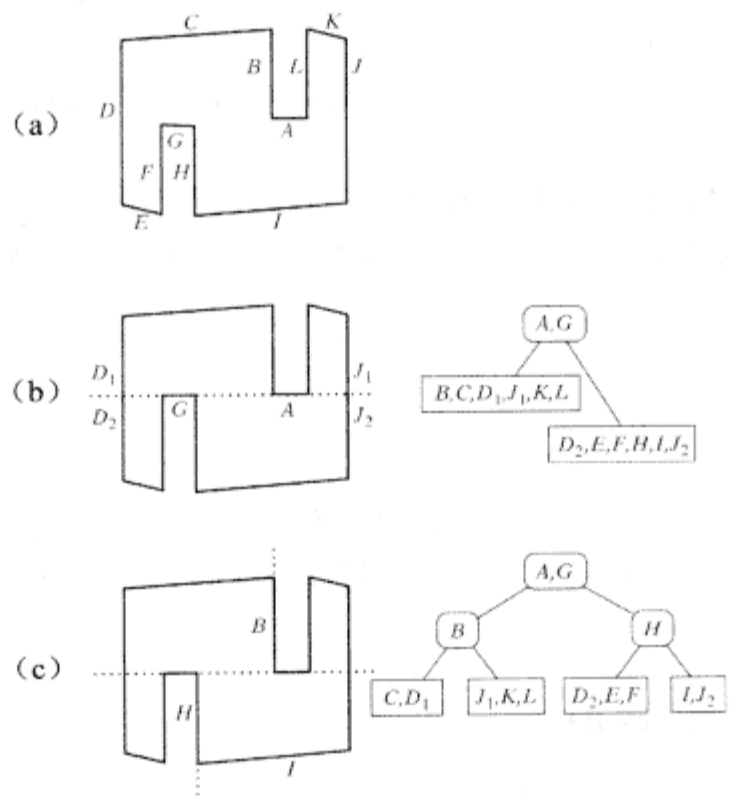


图 8.3 （a）由 12 个多边形构成的输入几何体。（b）穿越数据面 *A*（以及数据面 *G*）的初始分割面。（c）针对树型结构的下一层，分割面将穿越数据面 *B* 和 *H*

[Fuchs83]中最初采用了节点存储的 BSP 树，其中仅包含分割面的多边形才会被存储于节点中。与该分割面共面的多边形则可置于分割面的任意一侧。

8.2.2　采用叶节点存储的 BSP 树

术语叶节点存储（基于叶节点）的 BSP 树是指：几何体存储于叶节点而非内部节点的 BSP 树结构。其中，树型结构的内部节点只包含分割面数据以及节点子树的索引。因为当前内部节点不包含多边形数据，所以采用叶节点存储方式的树查询操作在遍历分割面时无须对多边形进行测试。另外，树型结构上的碰撞查询将在与对象相交的叶节点中的全部多边形之间进行，并且一旦检测到碰撞发生，测试过程即刻退出。

对于采用叶节点存储的树型结构，其分割面可采用通用分割面或自动划分分割面。同时，与分割面共面的相关面数据则不存储于节点中，而是置入左子树或右子树中。当采用自动划分方案时，应对与分割面共面的数据面加以标记：在后续分割面的选择过程中，对于这一类已标记的面将不予考虑。若未有效排除此类面，从理论上讲，在后续子树操作过程中将会出现分割面重复选择这一毫无意义的现象。

采用叶节点存储树型结构的构建过程如图 8.4 所示。当采用某一任意分割面进行划分时，初始分割面将数据集合一分为二。

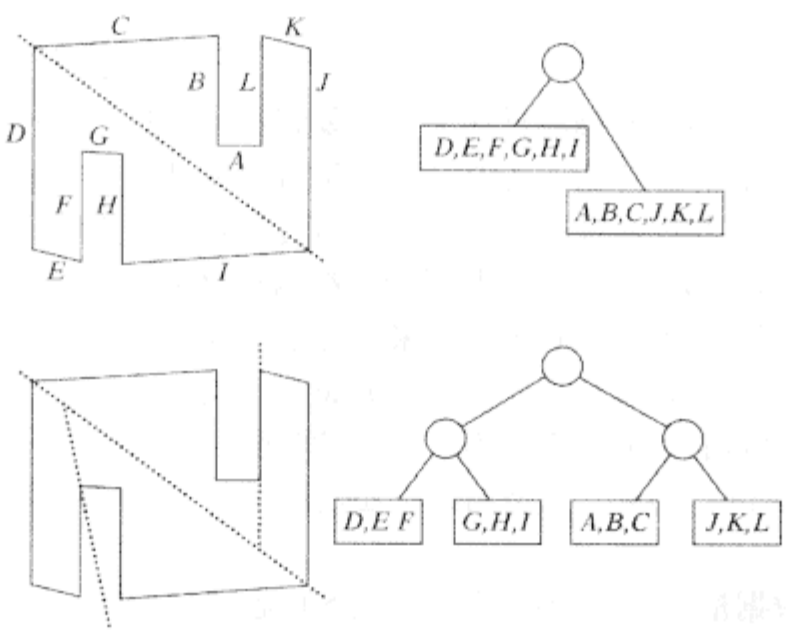


图 8.4　步骤一：构造叶节点存储的 BSP 树型结构

随后，再次对各子区域数据实施划分，直至获取最终的合理划分结果。特别地，当碰撞几何体定义为多边形汤时，采用叶节点存储的 BSP 树（以及 *k*-d 树、4 叉树和 8 叉树）则是一类较为常见且具有针对性的几何数据结构。

8.2.3　实体叶节点 BSP 树

不同于前述两种 BSP 树，实体叶节点 BSP 树可用于表示输入几何数据所占据的实体空间，亦即分

割面独立于对象并对实体空间进行划分。图 8.5 显示了一个 2D 实例，其中实体叶节点 BSP 树用于表达某一凹多边形的内部数据。

需要注意的是，应选择输入几何体中各多边形的支撑面作为分割面，否则，对象体空间将无法正确显示。特别地，与叶节点紧邻的内部节点中的分割面通常对应着某一支撑面，其他内部节点则一般采用任意分割面，这使得树型结构具有较好的平衡性。实际上，针对图 8.5 所示的对象，较好的初始分割面则是由凹顶点及其相对的顶点所定义的平面。

还需要注意的一点是，树型结构中并不存储几何体。叶节点仅表明位于分割面正面或背面的余留区域为空抑或实体。亦即，实体叶节点 BSP 树可视为将输入体空间划分为凸体集，并计算多个半空间的交集。同时，BSP 树的分割面对应着其划分平面以及凸多面体的边界平面。如图 8.5 所示，初始分割面将凹多边形划分为两个（凸）三角形，其他分割面则采用了相关三角形的边界。

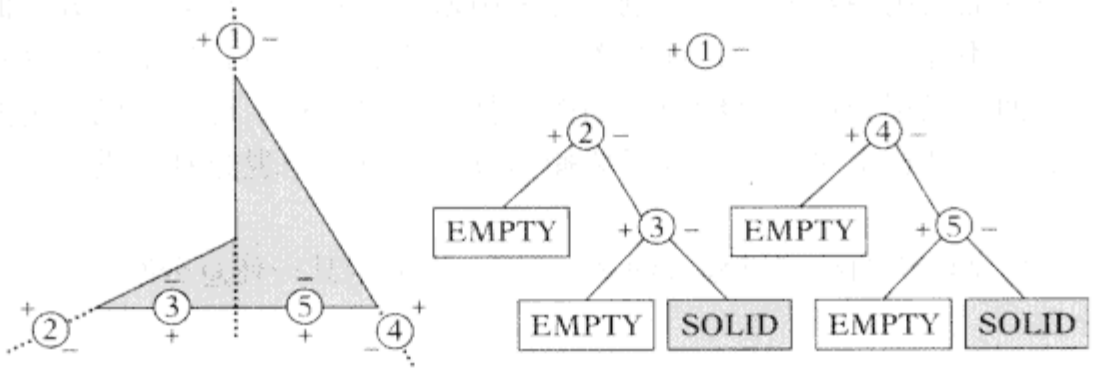


图 8.5　多个分割面的实体对象划分过程，并生成实体叶节点 BSP 树

实体叶节点 BSP 树包含许多与碰撞测试相关的重要特性。该树型结构对于某些健壮性问题（如穿越实体表面）的处理，其手法十分简练。其中，相关碰撞测试可直接反馈对象位于实体内部还是无效外围空间中。这一类树型表达方式可对输入几何体中的边界多边形实现高效的编码操作，因而无须执行特定的多边形测试（另外，多边形也从未显式地存储于树型结构中），并可生成优雅、均一的代码。

当然，完全可以结合实体叶节点 BSP 树以及采用叶节点存储的 BSP 树，以生成一类混合方案。关于混合树的实例，可参考 id Software 公司的作品 Quake II 和 Quake III，其中广泛使用了 brush 存储的树型结构。这一类树型结构针对叶节点实体空间（半空间交集）中的各个平面加以显式存储（在 Quake 术语中称该平面集合为 brush）。另外，树型结构中还使用了削角面并可实现高效的 AABB 查询（参见 8.4.3 节）。

Quake 中使用的各类数据格式其文档在网络上随处可见。例如，[McGuire00]中即描述了基于 Quake II 的 BSP 文件格式；[idSoftware01]中也通过源代码的方式对其进行了有效的阐述。关于 Quake III 的碰撞检测分析，读者可参考[Ostgard03]。

8.3　构造 BSP 树

构造 BSP 树包含下列 3 个步骤：

1．选择分割面。

2．依据分割面将输入几何体划分为两个集合：位于分割面正面的几何体和位于分割面背面的几何

体。而对于跨越分割面的几何体，在划分之前首先对其进行分割计算（分割面上的多边形则需要依据树的构造类型以区别对待）。

3. 通过连接新的树节点以实现树型结构的构造。对划分后的数据集合递归地调用相应的构造算法，以完成对两棵子树的构建操作。

上述步骤的实现代码如下所示（这里采用了叶节点存储的树型结构）：

```
// Constructs BSP tree from an input vector of polygons. Pass 'depth' as 0 on entry
BSPNode *BuildBSPTree(std::vector<Polygon *> &polygons, int depth)
{
    // Return NULL tree if there are no polygons
    if (polygons.empty()) return NULL;

    // Get number of polygons in the input vector
    int numPolygons = polygons.size();

    // If criterion for a leaf is matched, create a leaf node from remaining polygons
    if (depth >= MAX_DEPTH || numPolygons <= MIN_LEAF_SIZE) || ...etc...)
        return new BSPNode(polygons);

    // Select best possible partitioning plane based on the input geometry
    Plane splitPlane = PickSplittingPlane(polygons);

    std::vector<Polygon *> frontList, backList;

    // Test each polygon against the dividing plane, adding them
    // to the front list, back list, or both, as appropriate
    for (int i = 0; i < numPolygons; i++) {
        Polygon *poly = polygons[i], *frontPart, *backPart;
        switch (ClassifyPolygonToPlane(poly, splitPlane)) {
        case COPLANAR_WITH_PLANE:
            // What's done in this case depends on what type of tree is being
            // built. For a node-storing tree, the polygon is stored inside
            // the node at this level (along with all other polygons coplanar
            // with the plane). Here, for a leaf-storing tree, coplanar polygons
            // are sent to either side of the plane. In this case, to the front
            // side, by falling through to the next case
        case IN_FRONT_OF_PLANE:
            frontList.push_back(poly);
            break;
        case BEHIND_PLANE:
            backList.push_back(poly);
            break;
        case STRADDLING_PLANE:
            // Split polygon to plane and send a part to each side of the plane
            SplitPolygon(*poly, splitPlane, &frontPart, &backPart);
            frontList.push_back(frontPart);
            backList.push_back(backPart);
            break;
```

```
        }
    }

    // Recursively build child subtrees and return new tree root combining them
    BSPNode *frontTree = BuildBSPTree(frontList, depth + 1);
    BSPNode *backTree = BuildBSPTree(backList, depth + 1);
    return new BSPNode(frontTree, backTree);
}
```

针对叶节点存储的树型结构或叶节点实体类型的 BSP 树，位于分割面上的多边形可置于其任意一侧（此处，相关多边形置入分割面的正面一侧）。而对于采用节点存储的树型结构，共面多边形则存储于节点之中。

虽然上述代码结构简单，但其中的帮助函数仍存在某些问题以待处理。特别地，令函数 **PickSplittingPlane()**选择一个较好的分割面并非轻而易举。在 8.3.1 节和 8.3.2 节中将详细讨论这一问题。另外，函数 **ClassifyPolygonToPlane()**和 **SplitPolygon()**中所涉及的多边形分类操作以及分割计算等健壮性问题，也将在 8.3.3 节和 8.3.5 节中加以讨论。

下面将考查何时结束树型结构的构造过程。当然，这取决于 BSP 树的构造类型。对于节点存储的树型结构，当输入几何体集合最终为空时，递归构造过程结束。相同方式也可用于实体叶节点类型的 BSP 树。而采用叶节点存储的 BSP 树，当出现下列情况之一时，构造过程结束：

- ☑ 叶节点中的数据小于某一预设多边形数量。
- ☑ 到达某一分割深度。
- ☑ 不存在较优的分割面。

针对树型结构的构造以及动态查询，本章中所显示的示例均采用同一树型表达方式。这里需要强调的是，基于计算性能方面的考虑，一个良好的系统很可能采用两种完全不同的结构方式。在构造阶段，BSP 树的设计思路应着重于操作的简洁性，因而可使用指针对子节点加以链接，并采用全尺寸非压缩字段对树型结构加以表达。相比较而言，在运行阶段，BSP 树则应占用较少的内存空间（采用指针和压缩节点）并以缓存友好的方式输出节点（参见第 6 章和第 13 章）。

8.3.1 分割面的选择

在树型结构的构造过程中，分割面的选择顺序将极大地影响最终树型结构的尺寸和形状，进而决定其查询效率以及内存空间的占有量。考虑到分割面顺序的多样性，计算最优 BSP 树将是一个十分棘手的问题。但在实际应用中，可采用一类近似最优算法对此类问题加以解决。例如，可在树型结构的各层中尝试一定数量的候选分割面，并根据相应的评估函数获取最佳分割面（参见 8.3.2 节）。

由于分割面数量上的不确定性，可通过一类算法对该问题加以限定。例如，可利用自动划分方案将分割面的数量限制在一个有限范围内，亦即针对输入几何体中各个面的支撑面，该策略对其搜索过程加以约束。

自动划分方案易于实现，且该算法也是采用节点存储方式的树型结构的自然选择方式。然而，仅就自动划分方案而言，它在某些情况下表现得并不理想。例如，对于采用叶节点存储方式的树型结构，

若仅依据自动划分则会导致分割次数的增加。图 8.6（a）显示了相关的几何数据，其中，采用自动划分方案的分割面将不可避免地对数据面进行分割。图 8.6（b）则显示了采用通用分割面的划分过程。其结果为：多边形组彼此分离且未被分割（需要注意的是，若 3 组多边形彼此过于靠近，则很难获取相应的分割面对其实行划分）。

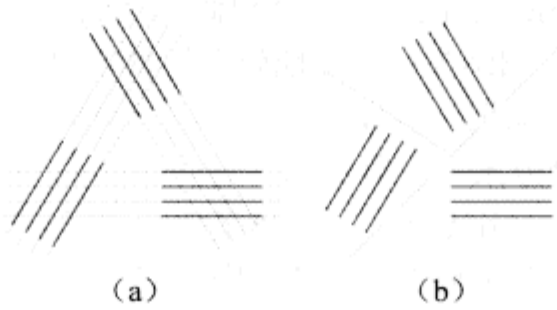


图 8.6　（a）针对 12 个数据面，其全部自动划分分割面最终将分割 4 个数据面。
（b）采用通用划分方案可消除或减少相关问题的出现

自动划分的另一个缺陷还会出现在下列情形中：基于多边形球体的 BSP 树。当选取某一数据面作为分割面时，凸形球体的全部数据面可能都位于分割面的一侧——这将导致树型结构严重失衡，其深度将等于球体数据面的数量（如图 8.7（a）所示）。在最坏情况下，点包含测试的复杂度将为 $O(n)$。相比较而言，若分割面可穿越球体，则相对于分割面的每一次测试，都将剔除约一半数量的数据面，则最坏情况下，查询的时间复杂度降为 $O(\log n)$（如图 8.7（b）所示）。

如前所述，如果 BSP 树采用球体边界面这一形式加以计算，则可选择球体数据面对应的支撑面作为分割面。同时，在此基础上还可采取通用分割方案，并对几何体实现二分计算直至叶节点中仅包含一个多边形。

该方案的缺点是：全部叶节点的深度值皆为 $O(\log n)$，因而对那些完全位于球体之外的节点查询，该方案无法保证“提前退出”。

[Naylor93]提供了一类较好的解决方法。Naylor 采取了一种混合方案，即综合使用自动划分方案和通用划分方案，如此，既维持了树型结构的深度为 $O(\log n)$，又保证了查询操作的“提前退出”。算法首先选择一组自动划分平面并将当前球体与周围空间分离开来，最终形成一个包围球体的四面体（如图 8.7（c）所示，A、B 和 C 构成了一个包围三角形）。

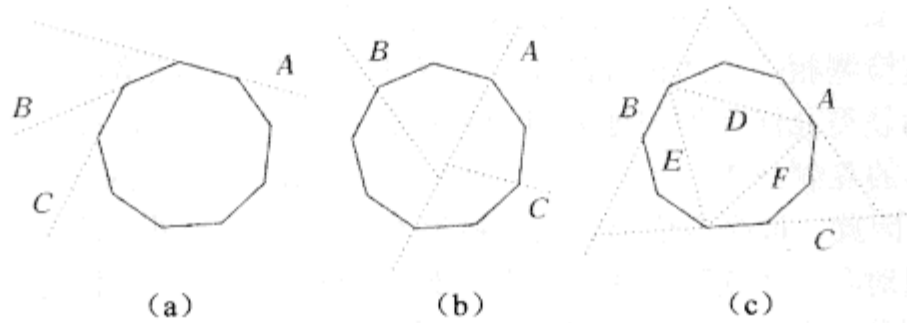


图 8.7　（a）最坏情况下，基于多边形球体的自动划分的 BSP 树型结构，其高度为 $O(n)$。（b）当球体采用通用划分时，树型结构的高度可降至 $O(\log n)$。（c）采用自动划分和通用划分的 Naylor 混合方案同样支持球体边界划分方式，其树型结构高度为 $O(\log n)$并可实现提前退出

最终，空间体被球体内部的 4 个平面分割为 5 个独立空间（如图 8.7（c）中的 D、E 和 F 所示）。同时，可得到与某一空间对应的、球体内部的一个四面体，而剩余的 4 个空间（在 2D 环境下显示为 3 个空间）可递归地执行上述分割过程，直至全部数据面处理完毕。最终的树型结构可视为经过不断修正的、球体的近似构造序列，且随着树型结构层次的不断增加，划分结果愈加精确。另外，该算法与第 9 章将要讨论的 Dobkin-Kirkpatrick 层次结构十分相似。

因此，全场景的 Naylor 算法可归纳为：高效地处理场景内的细节对象，同时，生成该对象的近似包围体并利用包围体数据面进行自动划分，最后针对包围体内的几何体实现分割计算。其中，除了自动分割面，其他候选分割面的选择方案如下所示：

- ☑ 某些预设方向上的（如轴对齐方向）、穿越多边形顶点的平面。
- ☑ 某些预设方向上的均匀分布于输入几何体范围内的（网格）平面。
- ☑ 穿越多边形某一边及其他顶点的平面。

除了分割面的自动选择方案，还可以自定义提示平面（hinting plane）以控制分割面的选择。这一类平面一般对应着具有某种特定标记的多边形，且不作为渲染几何体加以输出，其存在的目的只为定义分割面。关于提示平面的制定方案，包括以下几点：

- ☑ 若提示面已知，则首选该平面。
- ☑ 提示面的选择范围并没有特定的限制，但需要强调的一点是：应禁止非提示面对相关提示面实施分割计算。

例如，提示面适用于下列情形：在某一多层建筑物中，可将该面与地面对齐放置，并可快速定位地面位置，进而继续执行相关搜索。

最后，可采用简单的爬山算法对其他平面进行选择，以确定最终的分割面。亦即，待获取初始分割面后（可采用任意方案），对其稍作偏置以获取多个相似面，并在此邻域空间内进行评估。若存在一个较优的分割面，则将其保留。重复执行该爬山算法，直至获取某一（局部）最大值或超时。

8.3.2 分割面的评估

目前，存在多种分割面评估方案（参见[James99]）。这一类方案假设使用节点存储的树型结构，且支撑面可依据下列方式加以选择：具有最大共面值的数据面，或具有最大面积的数据面。将这一类数据面-存储至节点中，即可移除大量无关数据面（情形 1）或具有较高分割几率的数据面（情形 2）。

下面将讨论与碰撞检测相关的两种分割面评估算法，其中，算法拾取相应的分割面并尽量减少几何体的分割次数，从而使得几何体均匀地分布于分割面的两侧。首先，最少分割策略在划分过程中总是尽可能地避免几何体的复制操作。这里，若初始多边形的数量为 n，则在最坏情况下，一次划分操作即可引入 n 个多边形。因此，最终多边形的数量将为 $O(n^2)$级别。另外，跨越多边形的一类分割面通常位于场景的边界处，以对各场景实现隔离。因此，最少分割策略很可能使得树型结构的平衡性极差。另一方面，虽然平衡划分将会生成具有良好结构的树型，但从场景及其对象的中部实施划分操作常会生成过量的分割几何体。针对上述两种情况，在实际应用中常采用一类加权线性组合计算。与平衡性相比，降低几何体的分割次数往往具有更大的权值。图 8.8 显示了相应的划分操作且遵循下列原则：尽量避免跨越多边形；位于分割面两侧的多边形数量应保持均衡。因而，此划分操作可视为二者之间的一种折中方案。

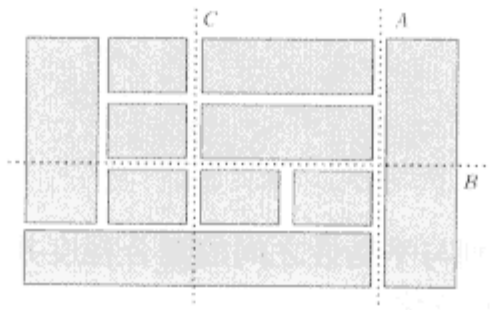


图 8.8　城市网格状规划示意图。这里尽量将穿越多边形（*A*）的数量降至最低，以使分割面（*B*）两侧的多边形数量呈均匀分部状态，并在最小化穿越多边形数量以及多边形（*C*）均匀分部之间获取某种折中方案

下列代码片段显示了较为基础的自动划分分割面选择算法。其中，代码将考查多边形集合中的全部支撑面，以选择最终的分割面。最后，代码会计算一个加权线性组合值。

```
// Given a vector of polygons, attempts to compute a good splitting plane
Plane PickSplittingPlane(std::vector<Polygon *> &polygons)
{
    // Blend factor for optimizing for balance or splits (should be tweaked)
    const float K = 0.8f;
    // Variables for tracking best splitting plane seen so far
    Plane bestPlane;
    float bestScore = FLT_MAX;

    // Try the plane of each polygon as a dividing plane
    for (int i = 0; i < polygons.size(); i++) {
        int numInFront = 0, numBehind = 0, numStraddling = 0;
        Plane plane = GetPlaneFromPolygon(polygons[i]);
        // Test against all other polygons
        for (int j = 0; j < polygons.size(); j++) {
            // Ignore testing against self
            if (i == j) continue;
            // Keep standing count of the various poly-plane relationships
            switch (ClassifyPolygonToPlane(polygons[j], plane)) {
            case POLYGON_COPLANAR_WITH_PLANE:
                /* Coplanar polygons treated as being in front of plane */
            case POLYGON_IN_FRONT_OF_PLANE:
                numInFront++;
                break;
            case POLYGON_BEHIND_PLANE:
                numBehind++;
                break;
            case POLYGON_STRADDLING_PLANE:
                numStraddling++;
                break;
            }
        }
        // Compute score as a weighted combination (based on K, with K in range
        // 0..1) between balance and splits (lower score is better)
        float score = K * numStraddling + (1.0f - K) * Abs(numInFront - numBehind);
        if (score < bestScore) {
            bestScore = score;
```

```
            bestPlane = plane;
        }
    }
    return bestPlane;
}
```

此处，变量 K 用于控制最少分割与平衡性之间的混合程度并设为常量。在实际应用中，该值可在树型构造过程中随当前节点的深度而变化。

多数碰撞查询操作只需通过遍历节点即可访问到与查询对象相交的某一个（或几个）叶节点，因而具有良好平衡性的树型结构则意味着全部叶节点应基本处于同一层次，这也最大限度地避免了最坏情况的发生。但对于平均操作情况而言，分割面的选择不应只侧重于树型结构的平衡性，还需要提供查询过程中的最短遍历路径。在图 8.9（a）中，分割面均分相应空间，且分割面两侧各包含等数量的多边形，从而使分割结果呈现良好的平衡性。假定点查询均匀分布于空间内部，则预计多边形测试数量为 $0.5\times5+0.5\times5=5$。在图 8.9（b）中，空间的分割比例为 2/3，预计多边形测试数量为 $2/3\times2+1/3\times8=4$。另外，若查询操作不再呈现均匀分布状态，则可对真实状态下的查询位置进行概率统计，以计算其精确的概率值。

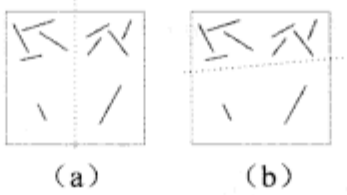


图 8.9 （a）均匀分割。（b）最小化分割

相关研究表明，无须考查过多的分割面即可获得较好的计算结果。例如 Fuchs 等人指出：针对最少分割策略，只需在每层中随机选择 5 个候选多边形并与其他多边形进行测试，即可生成最小树型结构（参见[Fuchs80]）。这一结论在[James99]中也得到了验证。需要注意的是，上述结论皆采用了自动划分方案以及节点存储的树型结构（基于 BSP 树的理论研究结果多半采用自动划分方案和节点存储的树型结构，因而往往与实际应用结果大相径庭）。

除此之外，其他类型的分割面评估算法还包括最小化冲突算法、中和冲突算法等（参见[James00]）。这一类算法一次考查两个或多个连续的分割面，并在树型结构的当前层中选择分割面以尽可能地减少冲突（这里，冲突是指数据面被某一潜在分割面分割）。再次强调，上述算法是针对自动划分方案进行操作的，且不具有一般性。

某些评估算法，例如针对平衡性的估算，通常不会出现分割面的重复选择。但即使如此，仍须引入一类显式机制以避免重复选择的发生。例如，当采用自动划分方案时，可对包含已选支撑面的多边形加以标记，且在后续操作中将不再选取已标记多边形中的支撑面，进而避免分割面的重复选择。有时，针对已选分割面也可采取记录表的方式加以解决。

8.3.3 基于分割面的多边形分类

作为 BSP 树构造过程中的一个环节，待分割面选取完毕后，将按如下方式划分全部多边形：

- ☑ 位于分割面正面的多边形。
- ☑ 位于分割面背面的多边形。

☑　多边形跨越分割面。

☑　多边形与分割面重合。

虽然上述理论较为直观，但在浮点运算过程中，该分类方案需要考虑下列问题（即平面上一点的误差问题）：相对于该平面，所谓“平面上一点”仍存在某些距离上的偏差。当多边形被某一分割面分割时，这类偏差将使得分割结果无法相对于分割面的正、反两面加以正确分类。例如，在计算交点时，即使分割操作在理论上将会生成位于分割面两侧的划分结果，计算误差也将使其跨越分割面。因此，针对健壮的多边形分类操作，正确地处理浮点运算问题是十分必要的。

在实际应用中，算法常采用具有一定“厚度”的平面消除上述误差。亦即，靠近平面的全部顶点皆视为该平面上的点。这里不再使用平面方程 $\mathbf{n}\cdot(X-P)=0$，若顶点满足 $|\mathbf{n}\cdot(X-P)|<\varepsilon$，则该点位于平面之上。下列函数显示了基于厚平面的顶点分类计算：

```
// Classify point p to a plane thickened by a given thickness epsilon
int ClassifyPointToPlane(Point p, Plane plane)
{
    // Compute signed distance of point from plane
    float dist = Dot(plane.n, p) - plane.d;
    // Classify p based on the signed distance
    if (dist > PLANE_THICKNESS_EPSILON)
        return POINT_IN_FRONT_OF_PLANE;
    if (dist < -PLANE_THICKNESS_EPSILON)
        return POINT_BEHIND_PLANE;
    return POINT_ON_PLANE;
}
```

然而，定义合适的 ε 值具有一定难度。该问题暂且搁置并在第 11 章中加以详细讨论。

图 8.10 显示了相对于厚平面的多边形分类操作过程，并考查了 4 个相关三角形。针对三角形 *ABC*，顶点 *A* 位于分割面之上，顶点 *B*、*C* 则位于分割面的背面。由于该三角形无顶点位于分割面的正面，因此三角形 *ABC* 归类于分割面的背面。在三角形 *DEF* 中，顶点 *D* 位于分割面的正面，顶点 *E*、*F* 位于分割面之上。由于该三角形无顶点位于分割面的背面，则三角形 *DEF* 归类于分割面的正面。需要注意的是，即使顶点 *E* 位于平面 $\mathbf{n}\cdot(X-P)=0$ 的负半空间，三角形 *DEF* 仍视为位于分割面的正面——厚平面计算视顶点 *E* 位于分割面之上。

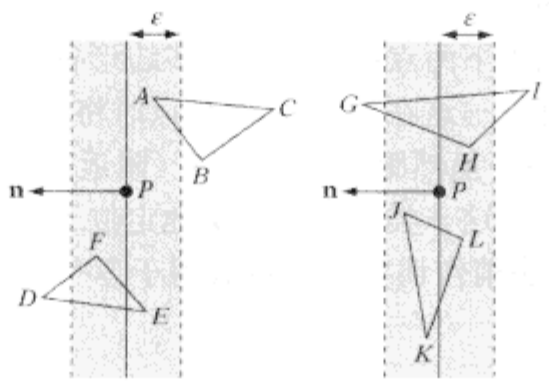


图 8.10　三角形 *ABC* 位于平面的背面，三角形 *DEF* 位于平面的正面，三角形 *GHI* 穿越平面，三角形 *JKL* 则与平面重合

三角形 *GHI* 跨越分割面，其顶点 *G* 和 *I* 分别位于分割面的正面和背面；三角形 *JKL* 则与分割面共

面，其全部顶点皆位于厚分割面之上。下列代码显示了基于分割面的任意多边形分类的扩展算法：

```
// Return value specifying whether the polygon 'poly' lies in front of,
// behind of, on, or straddles the plane 'plane'
int ClassifyPolygonToPlane(Polygon *poly, Plane plane)
{
    // Loop over all polygon vertices and count how many vertices
    // lie in front of and how many lie behind of the thickened plane
    int numInFront = 0, numBehind = 0;
    int numVerts = poly->NumVertices();
    for (int i = 0; i < numVerts; i++) {
        Point p = poly->GetVertex(i);
        switch (ClassifyPointToPlane(p, plane)) {
        case POINT_IN_FRONT_OF_PLANE:
            numInFront++;
            break;
        case POINT_BEHIND_PLANE:
            numBehind++;
            break;
        }
    }
    // If vertices on both sides of the plane, the polygon is straddling
    if (numBehind != 0 && numInFront != 0)
        return POLYGON_STRADDLING_PLANE;
    // If one or more vertices in front of the plane and no vertices behind
    // the plane, the polygon lies in front of the plane
    if (numInFront != 0)
        return POLYGON_IN_FRONT_OF_PLANE;
    // Ditto, the polygon lies behind the plane if no vertices in front of
    // the plane, and one or more vertices behind the plane
    if (numBehind != 0)
        return POLYGON_BEHIND_PLANE;
    // All vertices lie on the plane so the polygon is coplanar with the plane
    return POLYGON_COPLANAR_WITH_PLANE;
}
```

由于引入了 ε 值，即通过判断点与厚平面之间的位置关系，从而改善了前述误差问题，且函数的执行不受任何影响。但仍需要注意以下两点：首先，在多边形各边的端点与分割面之间引入了 ε 值之后，会增加边与分割面之间的角度——这在计算边-面交点时将会提高相应的精确度；其次，较好的 ε 值可确保正确地处理点-面包含关系。虽然距分割面 ε 值（附近）的点仍会产生分类误差，但该误差不会影响多边形分割计算的健壮性——这一类交点将正确地定位于厚平面上。

对传统的多边形-平面分割算法稍作调整即可将其用于厚平面处理，相应的调整方法将在后文中讨论。

8.3.4 多边形分割计算

在 BSP 树构造过程中，当多边形跨越分割面时，需要执行多边形的分割计算，并将相应的分割结

果置于分割面的两侧。如图 8.11 所示，其中执行了两次分割操作并将空间划分为 A、B、C 3 个区域。三角形 T 分别与 A 和 B 相交，因而可在叶节点中被引用。如果不执行三角形 T 的分割计算，则有 T 跨越分割面 1 并分别置于分割面的两侧。T 还将与分割面 2 执行测试，且再次置于该分割面的两侧。此时，全部叶节点构造完毕且无一例外包含三角形 T。

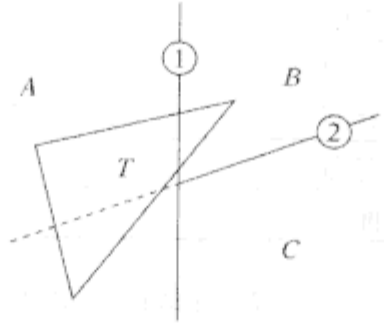


图 8.11　若 T 未被第一个分割面分割，则 T 将穿越第二个分割面，复件最终将位于区域 C 对应的叶节点内；否则该叶节点将为空

当三角形 T 针对平面 1 执行分割计算时，分割面右侧只有一小部分几何体置于右子树中，且完全位于分割面 2 的一侧，此时，与区域 C 对应的叶节点将为空。亦即，树型结构构造过程中的重要一步即是对多边形执行剪裁计算。这一类剪裁操作最终将多边形限制在下列叶节点中：其关联空间体不与多边形相交。

另一种方案是维护与叶节点对应的空间体，并执行多边形-凸体-多面体测试，以最终确定多边形是否应加入某一叶节点中。空间体的维护操作只限于轴对齐的 BSP 树，且其叶节点空间为 AABB。在其他场合中，该构造过程中的多边形分割计算方案则不适用。

分割面的剪裁操作一般使用 Sutherland-Hodgman 剪裁算法（[Sutherland74]）。该算法每次处理一条边，且依据边上两端点与剪裁面的相对位置，算法可分为 4 种情况，如图 8.12 所示。在第一个示例中，当两个端点皆位于剪裁面的正面时，输出尾端点；在第二个示例中，边由裁剪面的正面进入剪裁面的背面，则输出边-面的交点；在第 3 个示例中，当两个端点皆位于剪裁面的背面时，则不输出任何顶点；最后，当边由裁剪面的背面进入剪裁面的正面时，则输出边-面的交点和尾端点。

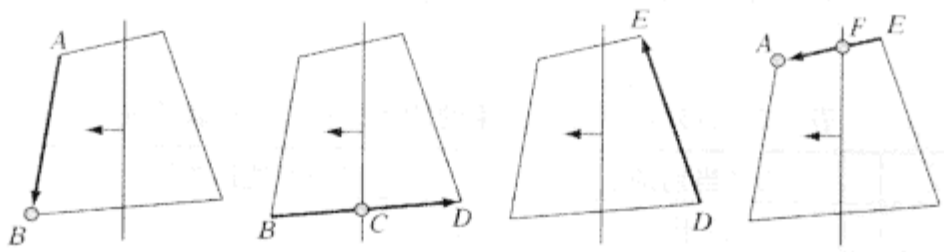


图 8.12　针对多边形 $ABDE$，采用 Sutherland-Hodgman 多边形分割算法时的 4 种情形。其中，输出多边形 $BCFA$ 的各顶点分别采用灰色表示

不难看出，当采用 Sutherland-Hodgman 算法时，仅位于剪裁面正面的多边形被保留。但在 BSP 树构造过程中，则需保留全部分割结果，因此需要对算法进行适当的扩展。另外，由于剪裁面为厚平面，因此需添加相应规则以正确处理面内顶点问题。一类常见的规则集如表 8.1 所示，其中增加了第 5 条规则以处理面内顶点问题。

如前所述，采取厚平面的一个原因是计算交点 I 时将会产生浮点误差，即交点无法准确定位于数

学平面上。

表 8.1　修订后的 Sutherland–Hodgman 剪裁算法并用于处理厚平面问题，该算法将分别保留剪裁后的两个结果多边形。其中，*AB* 表示有向线段，*I* 表示 *AB* 与剪裁面的交点

A（上一个顶点）	*B*（当前顶点）	正面多边形输出	背面多边形输出
位于正面	位于正面	*B*	无
位于背面	位于正面	*I*、*B*	*I*
位于正面	位于背面	*I*	*I*、*B*
位于背面	位于背面	无	*B*
任意位置	位于面上	*B*	*B*

使用厚平面的另一个优点是：仅需针对跨越剪裁面的相应边计算交点。考虑到剪裁面的厚度值，还需要对边-面的平行特征加以限定，这也从另一方面提升了交点计算的准确性。在第 11 章中将进一步探讨浮点运算问题。

对于多数应用而言，表 8.1 中的规则集都将生成正确的结果。但需要强调的一点是，在某些情况下，剪裁后的两个多边形会彼此相交。这里，考查如图 8.13 所示的四边形 *ABCD*，当被平面分割后，输出部分将同时包含 *BCE* 部分。

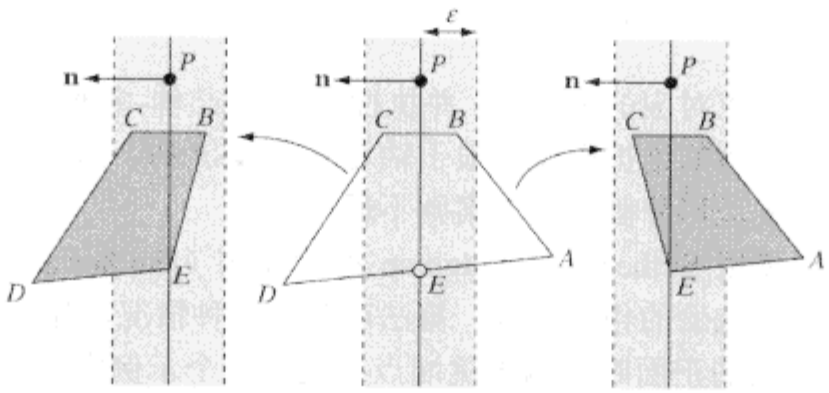


图 8.13　修正后的分割算法将导致潜在问题：分割面片间（以深灰色表示）可能相交

因此，针对健壮的多边形检测算法，分割后的多边形之间不应彼此相交，需要考查全部 9 种情况，如表 8.2 所示。

表 8.2　针对厚平面所采用的健壮的多边形剪裁算法

A（上一个顶点）	*B*（当前顶点）	正面多边形输出	背面多边形输出
位于正面	位于正面	*B*	无
位于面上	位于正面	*B*	无
位于背面	位于正面	*I*、*B*	*I*
位于正面	位于面上	*B*	无
位于面上	位于面上	*B*	无
位于背面	位于面上	*B*	*B*
位于正面	位于背面	*I*	*I*、*B*
位于面上	位于背面	无	*A*、*B*
位于背面	位于背面	无	*B*

相关实现代码如下所示：

```
void SplitPolygon(Polygon &poly, Plane plane, Polygon **frontPoly, Polygon **backPoly)
{
    int numFront = 0, numBack = 0;
    Point frontVerts[MAX_POINTS], backVerts[MAX_POINTS];

    // Test all edges (a, b) starting with edge from last to first vertex
    int numVerts = poly.NumVertices();
    Point a = poly.GetVertex(numVerts – 1);
    int aSide = ClassifyPointToPlane(a, plane);

    // Loop over all edges given by vertex pair (n - 1, n)
    for (int n = 0; n < numVerts; n++) {
        Point b = poly.GetVertex(n);
        int bSide = ClassifyPointToPlane(b, plane);
        if (bSide == POINT_IN_FRONT_OF_PLANE) {
            if (aSide == POINT_BEHIND_PLANE) {
                // Edge (a, b) straddles, output intersection point to both sides
                Point i = IntersectEdgeAgainstPlane(a, b, plane);
                assert(ClassifyPointToPlane(i, plane) == POINT_ON_PLANE);
                frontVerts[numFront++] = backVerts[numBack++] = i;
            }
            // In all three cases, output b to the front side
            frontVerts[numFront++] = b;
        } else if (bSide == POINT_BEHIND_PLANE) {
            if (aSide == POINT_IN_FRONT_OF_PLANE) {
                // Edge (a, b) straddles plane, output intersection point
                Point i = IntersectEdgeAgainstPlane(a, b, plane);
                assert(ClassifyPointToPlane(i, plane) == POINT_ON_PLANE);
                frontVerts[numFront++] = backVerts[numBack++] = i;
            } else if (aSide == POINT_ON_PLANE) {
                // Output a when edge (a, b) goes from 'on' to 'behind' plane
                backVerts[numBack++] = a;
            }
            // In all three cases, output b to the back side
            backVerts[numBack++] = b;
        } else {
            // b is on the plane. In all three cases output b to the front side
            frontVerts[numFront++] = b;
            // In one case, also output b to back side
            if (aSide == POINT_BEHIND_PLANE)
            backVerts[numBack++] = b;
        }

        // Keep b as the starting point of the next edge
        a = b;
        aSide = bSide;
    }
```

```
    // Create (and return) two new polygons from the two vertex lists
    *frontPoly = new Polygon(numFront, frontVerts);
    *backPoly = new Polygon(numBack, backVerts);
}
```

需要注意的是，代码中调用了 **assert()**函数，以判断厚平面上某一交点的真实性。依据当前输入数据的坐标值范围，该函数还可用于对平面厚度值的优劣性实现有效的检测。

图 8.14 显示了平面分割两个多边形后的对应结果。其中，初始多边形采用轮廓线加以表示，分割结果则为内嵌的深灰色多边形。

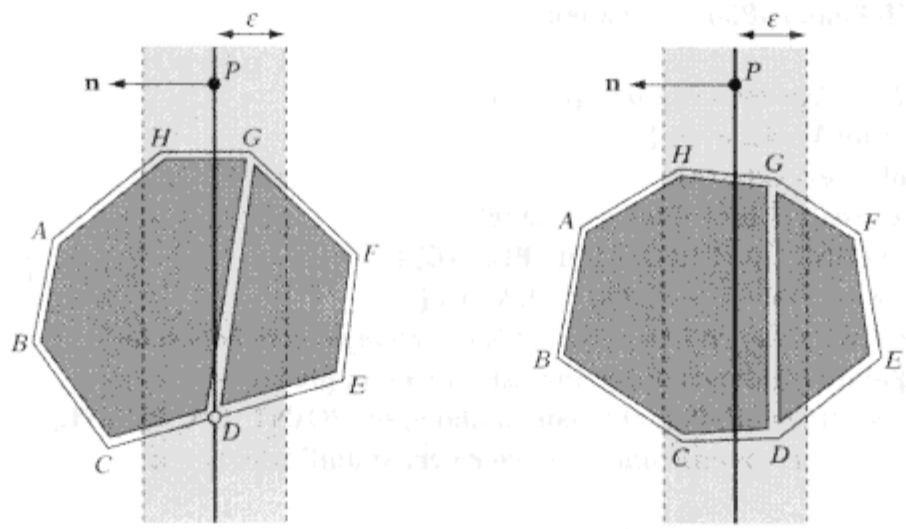


图 8.14　采用较为健壮的剪裁算法并使用厚平面对两个多边形进行分割操作。（内嵌）结果多边形用深灰色表示

虽然多边形在树构造过程中被分割，但这并不意味着叶节点中的多边形片段一定是最终树型结构中的输出多边形。

针对碰撞检测，在对多边形执行分割计算时，最好保存一个原多边形的引用。在叶节点中，一般输出该引用数据而非多边形片段自身。其中，对原多边形的引用加以存储可在平面剪裁计算过程中避免许多健壮性问题，例如狭长多边形和 *T* 型连接。

另外，剪裁计算将生成额外的顶点数据，非剪裁计算似乎更加节省内存空间。但如果不对多边形进行剪裁，则多边形相交测试仅限于叶节点空间内部，且无法保证对其外部的部分多边形执行测试。因此，针对光线相交测试，BSP 树的遍历操作将无法获取最近交点。

多数基于多边形剪裁算法的 BSP 树，其构造方式、资料都未涉及厚平面这一与健壮性紧密相关的话题。然而，[Chin92]却是一个例外，其中详细讨论了健壮的多边形剪裁算法。

8.3.5　更多讨论

虽然在前面已经对健壮的多边形分割计算进行了详细的讨论，但仍存在一个问题尚未解决，即对两个邻接多边形的公共边进行剪裁。考查图 8.15（a）所示的两个三角形 *ABC* 和 *CBD*，其顶点全部以逆时针方式排列。分割计算将剪裁公共边 *BC* 两次：每个三角形各一次。问题的关键之处在于：边 *BC* 属于三角形 *ABC*；边 *CB* 属于三角形 *CBD*。考虑到浮点运算与实数运算间的差异性，*BC* 与分割面的交点以及 *CB* 与分割面的交点并不相同。若对该问题不加处理，代码将会在两个邻接三角形之间生成缝

隙（交点各异）。边 *CB* 与剪裁面交于 *FCB* 且异于 *FBC*（此处放大图中的交点误差以更好地显示这一问题）。如图所示，这将在分割边之间产生一个缝隙。

图 8.15（b）显示了这一问题。对于三角形 *ABC*，边 *BC* 和 *CA* 被分割面剪裁，且边 *BC* 与分割面交于 *FBC*。对于三角形 *CBD*，边 *CB* 和 *BD* 被剪裁。*CB* 与平面相交于点 *FCB* 且异于点 *FBC*（这里放大交点误差以清楚地显示相关顶点的位置）。如图所示，这将在分割面之间产生缝隙。

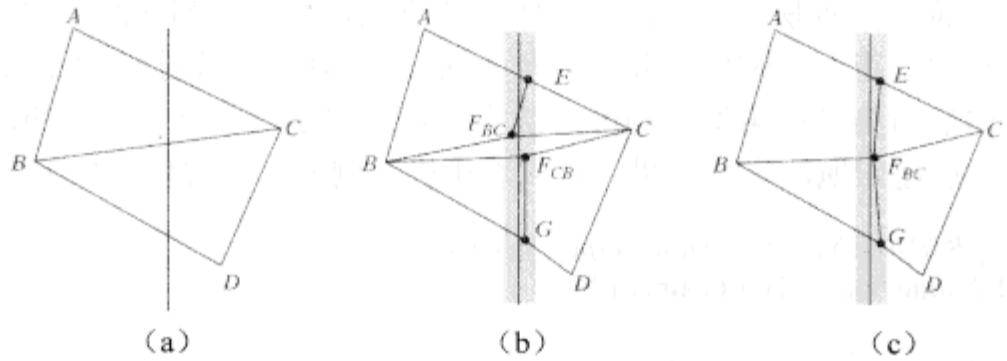


图 8.15　（a）两个三角形与某一平面相交。（b）公共边的交点不一致并产生缝隙。（c）公共边的交点一致

为了有效地解决这一问题，需要一次性地选择剪裁边：边 *BC* 或 *CB*。按照下列方式稍作调整即可：当某边的两个端点分别位于分割面的两侧时，则该边被剪裁。因此，可一致地定义该边由分割面的正空间进入其负空间（反之亦然），进而体现其一致性相交特征。

与 8.3.4 节中的代码的唯一区别在于：当再次调用函数 **IntersectEdgeAgainstPlane()**时，须交换该边的两个端点并使其作为参数传入其中，如下所示：

```
...
if (bSide == POINT_IN_FRONT_OF_PLANE) {
    if (aSide == POINT_BEHIND_PLANE) {
        // Edge (a, b) straddles, output intersection point to both sides.
        // Consistently clip edge as ordered going from in front -> behind
        Point i = IntersectEdgeAgainstPlane(b, a, plane);
        ...
    }
    ...
```

图 8.15（c）显示了正确的检查过程，并消除了三角形间的缝隙。

8.3.6　BSP 树的性能调试

对 BSP 树中的相关参数进行调试以实现其最佳性能并不容易。例如，分割策略 *A* 一定优于 *B*？树型结构的构造过程止于哪一层（深度值）？如何确定函数 **PickSplittingPlane()**中变量 ***K*** 的最佳取值？

一类较好的解决方案是在运行期内建立一个玩家或敌方的参考数据库，这将有助于对各相关参数的设定进行评估。在正常运行过程中，相关数据以及所采用的碰撞几何体将被写入日志文件中。另外，由于该数据库可对碰撞几何体的各类空间划分方案实现统一查询，进而可比较不同参数设置之间的优劣，以实现较好的空间划分方案。如果可能的话，参数的调试过程可通过类爬山算法自动实现（如模拟退火算法或遗传算法）。

8.4 BSP 树的应用

BSP 树可以非常方便地处理某一类特定问题。下面的示例将以从远到近的顺序渲染节点存储的 BSP 树中各节点的数据面。如前所述，通过特定的构造方式，位于分割面两侧的几何体彼此不相交。因此，若相机位于分割面的正面，则位于分割面背面的几何体将不会遮挡其正面的几何体（以及分割面）。相应地，从后向前渲染当前树型结构可归结为：先递归地渲染当前节点的远子树，随后渲染当前节点中的数据面，最后递归地渲染近子树（对应着相机一侧）。相关代码如下所示：

```
// Render node-storing BSP tree back-to-front w/ respect to cameraPos
void RenderBSP(BSPNode *node, Point cameraPos)
{
    // Get index of which child to visit first (0 = front, 1 = back)
    intindex=ClassifyPointToPlane(cameraPos,node->plane)== POINT_IN_FRONT_OF_PLANE;

    // First visit the side the camera is NOT on
    if (node->child[index]) RenderBSP(node->child[index], cameraPos);
    // Render all polygons stored in the node
    DrawFrontfacingPolygons(node->pPolyList);
    // Then visit the other side (the one the camera is on)
    if (node->child[index^1]) RenderBSP(node->child[index^1], cameraPos);
}
```

上述递归算法也可用于 BSP 树查询。另外，采用叶节点存储的 BSP 树仅是空间划分方案诸多形式中的一类，其详细内容可参考第 7 章。8.4.1 节将集中讨论基于实体叶节点 BSP 树的碰撞查询。

8.4.1 点与实体叶节点 BSP 树间的测试

点与实体叶节点 BSP 树间的测试则较为直观。在每一个节点中，需执行点-分割面之间的位置计算。若顶点位于分割面的正面，则访问对应正面空间的子节点，反之亦然。遍历过程重复执行，直至到达叶节点。此时，叶节点对应的实体空间即为所求。

在上述计算过程中，唯一复杂之处在于如何正确处理“点位于实体空间边界之上”这一问题。其中，如点位于分割面上，则需分别遍历两棵子树，如果遍历结果相同（实体空间或空），则该结果即为最终答案；否则，点位于实体空间边界之内。下列代码显示了该测试的实现过程（采用厚平面）：

```
int PointInSolidSpace(BSPNode *node, Point p)
{
    while (!node->IsLeaf()) {
        // Compute distance of point to dividing plane
        float dist = Dot(node->plane.n, p) - node->plane.d;
        if (dist > EPSILON) {
            // Point in front of plane, so traverse front of tree
            node = node->child[0];
        } else if (dist < -EPSILON) {
```

```
            // Point behind of plane, so traverse back of tree
            node = node->child[1];
        } else {
            // Point on dividing plane; must traverse both sides
            int front = PointInSolidSpace(node->child[0], p);
            int back = PointInSolidSpace(node->child[1], p);
            // If results agree, return that, else point is on boundary
            return (front == back) ? front : POINT_ON_BOUNDARY;
        }
    }
    // Now at a leaf, inside/outside status determined by solid flag
    return node->IsSolid() ? POINT_INSIDE : POINT_OUTSIDE;
}
```

若不针对点-边界方式加以分类（即位于边界上的点可视为实体空间内部几何数据时），上述测试可简化为：

```
int PointInSolidSpace(BSPNode *node, Point p)
{
    while (!node->IsLeaf()) {
        // Compute distance of point to dividing plane
        float dist = Dot(node->plane.n, p) - node->plane.d;
        // Traverse front of tree when point in front of plane, else back of tree
        node = node->child[dist <= EPSILON];
    }
    // Now at a leaf, inside/outside status determined by solid flag
    return node->IsSolid() ? POINT_INSIDE : POINT_OUTSIDE;
}
```

8.4.2　光线与实体叶节点 BSP 树间的相交测试

一类基于 BSP 树的典型查询操作是：光线或线段与树型结构之间的相交计算。这里，可改写 7.4.1 节中的 **VisitNodes()**函数，即针对与线段相交的采用节点存储的 *k*-d 树，遍历其全部节点。本节将采用另一种非递归的遍历方案。

VisitNodes()函数与这里所讨论的查询操作有着显著的区别。前者将计算线段与分割面之间的交点，并将线段一分为二。随后，子线段递归划分并逐层构建子树。考虑到误差的累计效应，最终交点很可能偏离原线段，且查询操作所遍历的层次越深，交点与原线段之间的偏离值也就越大。从健壮性的角度来看，该方案难以令人满意。

误差积累效应并非不可避免。这里不再使用计算交点的方式，而是单独计算光线与分割面之间的相交 t 值。另外，在计算相交时刻时，采用厚平面也可以在一定程度上降低计算误差。

下列函数利用光线 $R(t) = P + t\mathbf{d}$，$t_{min} \leqslant t \leqslant t_{max}$ 与实体叶节点 BSP 树进行相交测试。其中，若二者相交，则返回首次相交时的 t_{hit}。

```
// Intersect ray/segment R(t) = p + t*d, tmin <= t <= tmax, against bsp tree
// 'node', returning time thit of first intersection with a solid leaf, if any
```

```
int RayIntersect(BSPNode *node, Point p, Vector d, float tmin, float tmax, float *thit)
{
    std::stack<BSPNode *> nodeStack;
    std::stack<float> timeStack;

    assert(node != NULL);
    while (1) {
        if (!node->IsLeaf()) {
            float denom = Dot(node->plane.n, d);
            float dist = node->plane.d - Dot(node->plane.n, p);
            int nearIndex = dist > 0.0f;
            // If denom is zero, ray runs parallel to plane. In this case,
            // just fall through to visit the near side (the one p lies on)
            if (denom != 0.0f) {
                float t = dist / denom;
                if (0.0f <= t && t <= tmax) {
                    if (t >= tmin) {
                        // Straddling, push far side onto stack, then visit near side
                        nodeStack.push(node->child[1^nearIndex]);
                        timeStack.push(tmax);
                        tmax = t;
                    }else nearIndex = 1^ nearIndex;        // 0 <= t < tmin, visit far side
                }
            }
            node = node->child[nearIndex];
        } else {
            // Now at a leaf. If it is solid, there's a hit at time tmin, so exit
            if (node->IsSolid()) {
                *thit = tmin;
                return 1;
            }
            // Exit if no more subtrees to visit, else pop off a node and continue
            if (nodeStack.empty()) break;
            tmin = tmax;
            node = nodeStack.top(); nodeStack.pop();
            tmax = timeStack.top(); timeStack.pop();
        }
    }
    // No hit
    return 0;
}
```

其中，光线与分割面间的 t 值决定了点 P 相对于分割面的位置关系，可复用该值以确定光线相对于分割面的投射方向。另外，在考查光线-分割面间的有效区间$[t_{min}, t_{max}]$时，应处理 4 种情况，如图 8.16 所示。当 $t<0$ 或 $t>t_{max}$ 时，仅需遍历分割面的近子树（点 P 所属空间）；当 $0\leqslant t<t_{min}$ 时，则遍历分割面的远子树；最后，若 $t_{min}\leqslant t\leqslant t_{max}$，则针对$[t_{min}, t]$遍历分割面的近子树，且针对$[t, t_{max}]$遍历分割面的远子树。

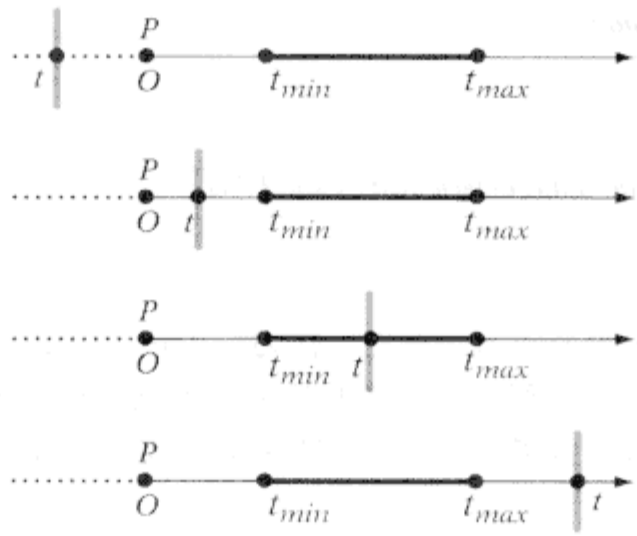


图 8.16　光线活动区域与平面（以灰色表示）相交的 4 种状态

若测试过程采用厚平面，则应添加一个 ε 项，使得光线与厚平面相交时能够正确地访问全部子树。在树型结构的构建过程中，ε 项应大于厚平面的厚度值。

考查下列情形：当光线与采用叶节点存储的树型结构相交，且多边形跨越分割面时，不对其进行剪裁，多边形分别置入分割面的两侧。此时将会产生光线穿越网格时的相同问题：光线可能与多边形交于当前叶节点体空间之外的某一点。然而，任何位于当前叶节点体空间之外的交点都将被删除，以保证处理过程正常进行。

8.4.3　基于实体叶节点 BSP 树的多面体查询

除了点和光线的查询操作之外，相对于 BSP 树的空间体查询也是一类重要的测试，例如 AABB 测试。这里，可采用蛮力算法在 BSP 中逐层构造每一个 AABB 的表面多边形，并对跨越分割面的多边形实施剪裁计算。如果多边形的某一部分位于实体叶节点内，则二者间存在碰撞。该算法的代码实现如下所示：

```
// Intersect polygon 'p' against the solid-leaf BSP tree 'node'
int PolygonInSolidSpace(Polygon *p, BSPNode *node)
{
    Polygon *frontPart, *backPart;
    while (!node->IsLeaf()) {
        switch (ClassifyPolygonToPlane(p, node->plane)) {
            case POLYGON_IN_FRONT_OF_PLANE:
                node = node->child[0];
                break;
            case POLYGON_BEHIND_PLANE:
                node = node->child[1];
                break;
            case POLYGON_STRADDLING_PLANE:
                SplitPolygon(*p, node->plane, &frontPart, &backPart);
                if (PolygonInSolidSpace(frontPart, node->child[0])) return 1;
                if (PolygonInSolidSpace(backPart, node->child[1])) return 1;
```

```
                    // No collision
                    return 0;
            }
        }
        // Now at a leaf, inside/outside status determined by solid flag
        return node->IsSolid();
}
```

然而，该算法（至少）存在两个问题。首先，作为一类边界测试，若 AABB 完全包含实体对象的某一较小且可以自由浮动的局部组成部分，二者间的碰撞测试往往会失效。其次，上述算法的计算开销也不容忽视：至少需要进行 6 次子查询方可确定最终的结果（而对于其他类型的多面体，其计算开销还会相应地增长）。

针对第一个问题，可将查询体表示为实体 BSP 树，并采用基于 BSP 树的集合操作计算两棵树的交集是否为空（参见[Thibault87]）。然而，该算法并不能解决第二个问题。

[Melax00]给出了另外一种算法，针对 AABB 和原 BSP 树形成的 Minkowski 和，该算法将构造与其对应的扩展空间体，并在此基础上形成一棵新的 BSP 树。因此，AABB 与原 BSP 树之间的测试就转换为基于当前扩展树的点查询操作。类似地，动态 AABB 与原 BSP 树之间的相交测试则转换为基于当前扩展树的光线测试。随着该算法的引入，上述问题均已解决。

另外，扩展空间体可按下列方式加以构造：原 BSP 树中的全部面沿其法线方向向外偏移 AABB 的半径 r 个距离（基于面法线）。例如，考查树节点中的某一存储平面，若其平面方程为 $\mathbf{n} \cdot X = d$，则 $\mathbf{n} \cdot X = d + r$ 即为其偏移平面。投影半径 r 按如下方式设定：

$$r = e_0|\mathbf{n}_x| + e_1|\mathbf{n}_y| + e_2|\mathbf{n}_z|$$

其中，e_0、e_1、e_2 为 AABB 范围值的一半，且 $\mathbf{n} = (a, b, c)$为存储平面的法线（参见 5.2.3 节）。

然而，简单地偏移 BSP 树节点中的某一平面还不足以彻底解决碰撞检测中的问题。如图 8.17（b）所示，空间体在顶点处（在 3D 环境下则为边）的扩展量过大，致使该范围内的碰撞检测失效。为了获取正确的结果，需要在扩展树中添加一个“削角面”，以便在平面偏移时，在 BSP 树和 AABB 之间构造正确的 Minkowski 和（如图 8.18 所示）。

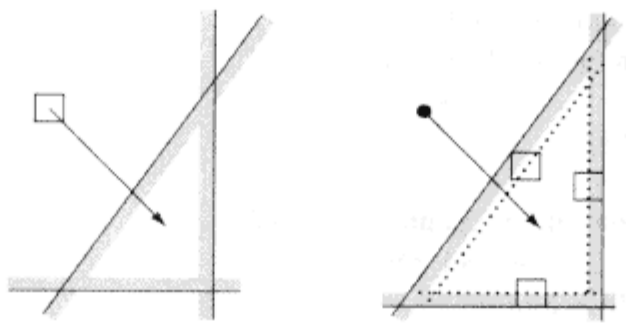


图 8.17　（a）AABB 与原相交体空间之间的相交测试。（b）半空间相交体中的各平面向外偏移 AABB 的半径距离（投影至其平面法线方向）以形成一个扩展体空间，从而将 AABB 查询测试替换为点查询测试。然而，这难以构造正确的 Minkowski 和：其形状往往会在各角处出现过量偏移，从而导致该区域的无效碰撞检测

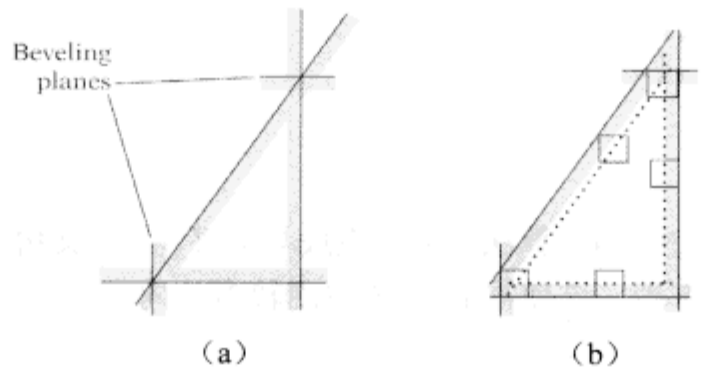


图 8.18　（a）必须额外添加削角面至 BSP 树，以构造相交空间体和 AABB 的 Minkowski 和。（b）平面偏移后，最终形状与下列操作相对应：沿相交体空间边界扫掠 AABB

在确定削角面时，考虑到空间体一般为多个半空间的交集，其最终定义为基于凸多面体的、实体 BSP 树中的实体叶节点。因此，针对 BSP 树的 AABB 查询操作一般可看作 AABB 与存储于实体叶节点中的多面体之间的相交测试。

针对 AABB 和多面体之间的碰撞测试，分离轴测试将计算碰撞产生的具体轴向。其中，存储于 BSP 树多面体中的各个平面（分割面）即可作为多面体数据面的分割轴。另外，AABB 数据面的分割轴，以及 AABB 边与多面体边的叉积向量都可作为削角面的选择依据。

特别地，对于穿越 AABB 数据面的支撑面，其添加位置位于该面法线方向、多面体的某一顶点处（若仍未添加的话）。而与边-边叉积向量对应的支撑面则使得该面穿越叶多面体中的相应边。

由于 AABB 的方向是固定的，若针对实体 BSP 树仅执行 AABB 查询，则可提前计算削角面并存储于树型结构中。如图 8.19 所示，削角面可添加在实体叶节点及其父节点之间的某一位置。

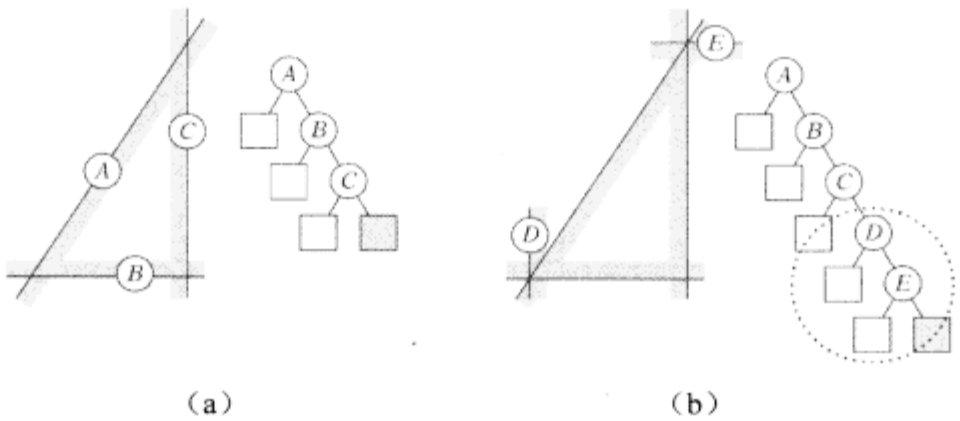


图 8.19　（a）三角形的非削角树型结构。（b）在实体叶节点及其父节点之间插入削角面

若针对 BSP 树执行任意类型的多面体查询（包括 OBB 和球体查询），则无法实现削角面的预计算过程，即相关计算只能在运行期内加以处理。另外，同样的情况还会发生在动态 BSP 树上（如 BSP 树旋转时）。针对采用了分离轴测试的削角面，为了便于其计算，实体 BSP 树应在实体叶节点之上添加相应的显式边界描述。

需要注意的是，当利用扩展树执行点查询或光线查询且在层次结构遍历过程中时，全部分割面都应采取一定程度上的偏移策略，而并非仅是 BSP 树实体边界平面。否则，叶节点与查询体之间将不会得到正确的相交测试结果。

8.5 小结

BSP 树是一类用途广泛的空间划分算法，可以完成与 4 叉树、8 叉树以及 *k*-d 树相同的任务，但需在每一个树型节点处存储平面方程，这使得 BSP 树占用较多的内存空间。同时，BSP 树还可作为一种多面体对象边界或实体的表达方式。

本章描述了 3 种类型的 BSP 树：采用节点存储的 BSP 树、采用叶节点存储的 BSP 树以及实体叶节点 BSP 树，并详细分析了自顶向下方式的、树型结构的递归构造算法，以及用于构建较优树型结构的分割面的选择、评估方案。

对于 BSP 树（或其他类似的数据结构），需要着力解决图元插入操作以及查询操作的健壮性问题，以确保得到正确的相交测试结果。就此问题，本章还阐述了使用厚平面的改进算法，以及厚平面的剪裁、分类算法。

作为一类 BSP 树的基本查询算法，本章还演示了点查询操作以及光线查询操作。最后，通过在查询期间添加削角面、偏移分割面并构造扩展树的方式，还详细讲解了基于 BSP 树的多面体查询操作。扩展树可有效地在多面体和原 BSP 树之间建立 Minkowski 和，并可简化光线查询计算，因而可视为一类高效的多面体查询替代算法。

第9章 凸体算法

本书所描述的包围体具有一个共同的特征：它们都为凸体对象。这一现象并非偶然，凸体对象所具有的某些属性使其非常适用于碰撞检测。例如，对于处于分离状态的凸体对象，总可以找到一个分割面（参见 8.3 节）。另外，凸体对象上两点间的距离既是局部最小值，也是全局最小值（如图 9.1（a）所示）。该属性使得两凸体对象间的全局最小距离可通过简单的爬山算法计算得到。亦即，任意选择两个顶点 A、B，（在对象范围内）逐步选择靠近 B 的各顶点，最终得到两顶点间的全局最小值。

凹体对象则不具备上述特征。例如，两个凹体对象间的局部最小值不一定是全局最小值（如图 9.1（b）所示）。

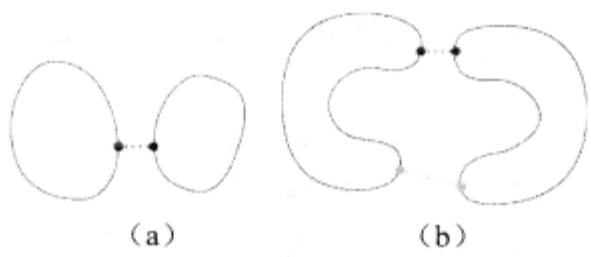


图 9.1 （a）针对两个凸体对象，两点间的局部最小距离即为全局最小值。（b）对于两个凹体对象，两顶点间的局部最小距离（灰色表示）则不必是全局最小值（黑色表示）

详细了解凸体对象的相关属性，将有助于设计出更好的碰撞检测算法。本章将主要分析与凸体类型相关的各种算法，同时，算法将主要围绕（凸）多面体展开。另外，一些算法还将涉及通用类型的凸体对象，如 9.5 节中的 GJK 算法。在本章的后续内容中，如果不加以特别说明，多面体一律视为凸体对象。

9.1 基于边界的碰撞检测

这里，需要构造一个参考坐标系以考查相应的凸体算法，两个多面体对象间的碰撞检测问题可针对其边界展开测试。此处，给定多面体 P 和 Q，且应尽量使其顶点和数据面（以及边）易于访问。

假定多面体位于同一参考坐标系（否则可将 Q 转换至 P 的空间内），其相交状态可按下列方式确定（[Moore88]）：

1. P 中的各边与多面体 Q 进行测试（例如，可采用 5.3.8 节中的相关算法）。若某一边部分或完全位于 Q 内部，则测试即刻停止并返回 “多面体处于相交状态”这一信息。

2. 相应地，Q 中的各边与多面体 P 进行测试。若某一边部分或完全位于 P 内部，则测试即刻停止并返回“P 和 Q 处于相交状态”这一信息。

3. 最后，还需正确地处理退化问题，即包含对齐数据面的同类对象（如立方体）间可能出现的彼

此穿越现象。针对这一情况，可在 Q 中各面质心坐标与 P 中的全部数据面之间进行测试。若 P 包含某一质心坐标，则测试即刻停止并返回“多面体处于相交状态”这一信息。

4．否则，返回“不相交”这一信息。

其中，步骤 3 似乎可归类于步骤 1 和步骤 2 中。但考虑到浮点误差问题，前两步计算很可能导致退化问题，因此质心测试是必不可少的。另外，步骤 1 和步骤 2 中的边测试的计算开销也是相当大的。因此，上述测试过程可按如下方式加以改进：

1．将 Q 中的各顶点与 P 中的所有面进行测试。若存在顶点位于 P 的内部，则测试即刻停止并返回“多面体处于相交状态”这一信息。

2．将 P 中的各顶点与 Q 中的所有面进行测试。

3．将 Q 中的各边与多面体 P 进行相交测试（例如，可采用 5.3.8 节中的相关算法）。若存在边与多面体 P 相交，则测试即刻停止并返回“多面体处于相交状态”这一信息。

4．同样，还需正确地处理退化问题，即包含对齐数据面的同类对象（如立方体）间可能出现的彼此穿越现象。针对这一情况，可在 Q 中各面质心坐标与 P 中的全部数据面之间进行测试。若 P 包含某一质心坐标，则测试即刻停止并返回“多面体处于相交状态”这一信息。

5．否则，返回“不相交”这一信息。

根据顶点数量或测试边的数量，上述算法的复杂度皆为 $O(n^2)$。在实际应用中，可对各多面体添加包围盒，并将相交测试计算限定在顶点、边、数据面与两包围盒交叠实体这一范围内。若多面体之间本不相交，则其包围盒一般情况下也不会交叠。因此，这一类无关的顶点、边以及数据面将被剔除，从而避免了与之相关的复杂测试。相反，当多面体之间相交时，通过计算多面体内部顶点，亦可在测试前期提高最终交点的命中率。其结果为：多数情况下的测试过程可视为一类线性操作，虽然这并非最优操作。如前所述，深入了解对象的凸体特征将有助于获取快速的碰撞检测算法。

9.2 最近特征算法

在实时仿真过程中，对象在帧间的位移量和旋转量通常较小。针对凸体对象，帧间的连贯性表明：通常情况下，两个非相交对象间的最近点基本位于上一帧中二者最近点的附近处。但对多面体而言，上述论点并非总是成立：瞬间的方位变化可能导致最近点位于相关数据面的另一侧，且（依据数据面的尺寸）最近距离将变得不确定。所以，多面体既不记录最近点也不使用来自上一帧的最近点数据作为初始顶点。一类较好的方案是在帧间记录最近特征（顶点、边或者数据面）。对于分别来自两个分离多面体的一组特征，其中若包含了一组最近点，则称之为多面体间的最近特征。

早期比较著名的、基于最近特征的碰撞检测算法应该是 Lin-Canny 算法（[Lin91]）。随后，一种类似的算法——Voronoi-Clip 算法（简称 V-Clip 算法）对 Lin-Canny 进行了修订。关于 V-Clip 算法，将在 9.2.1 节中加以介绍。

9.2.1 V-Clip 算法

V-Clip 算法在两个多面体之间根据最近特征计算其最近点，且有下列定理。

定理： 令 F_A、F_B 为两个分离多面体 A 和 B 之间的一组特征，且 $VR(F_A)$和 $VR(F_B)$记为 Voronoi 域。另外，令 $P_A \in F_A$，$P_B \in F_B$ 为特征间的最近点，若 $P_A \in VR(F_B)$，$P_B \in VR(F_A)$，则 F_A 和 F_B 定义为一组最近特征，P_A 和 P_B 定义为一组最近点（且不唯一）。

图 9.2 显示了满足该定理条件的一组（2D）多面体。

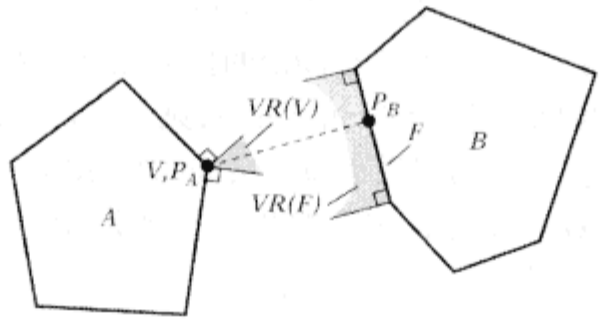


图 9.2 两个分离的 2D 多面体 A 和 B。图中所示为顶点-数据面之间的特征组合 V 和 F，并形成了最近特征组合且包含了对象间的最近顶点组合 P_A 和 P_B

V-Clip 算法始于一组（两个）特征，分别来自于每一个多面体。在算法的每一次迭代过程中，测试当前特征数据并考查是否满足定理条件所规定的最近特征。若满足条件，算法终止并返回“不相交”信息以及一组最近特征；否则，V-Clip 算法将当前某一特征更新为临近特征。临近特征按如下方式定义：

- ☑ 顶点的临近特征定义为包含该顶点的多条边。
- ☑ 面的临近特征定义为包围该面的多条边。
- ☑ 边的临近特征定义为边上的两顶点以及该公共边的两个邻接面。

根据上述列举的临近特征定义，图 9.3 显示了一类可能的特征组合及其合法的转换操作。其中，实箭头表示转换操作后特征间的距离将减少，虚箭头表示特征间的距离不发生任何变化。由于图中的虚箭头并不包含循环操作，因此仅执行有限数量的特征转换操作且特征间的距离不断减小。另外，图中不存在与增加特征距离对应的转换操作，因此可确保算法的最终收敛性且算法最终将返回一组最近特征（以及一组最近点）。

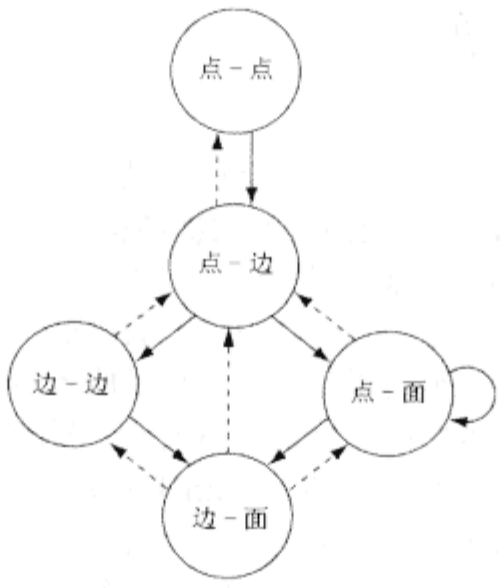


图 9.3 特征组合转换。其中，实箭头表示特征间的距离将减小，虚箭头则表示距离无变化

在图 9.3 中，实箭头指向了“顶点-面”状态自身——这并不符合上述临近特征定义。该转换操作的存在只是因为在“顶点-面”状态下，V-Clip 算法可能会出现局部最小值阻塞现象。该问题如图 9.4 所示，其中顶点 V 位于数据面 F 的支撑面的下方，且同时位于数据面 F 的 Voronoi 域 $VR(F)$ 中全部 Voronoi 平面的内部。在这种情况下，图 9.3 中不存在合法的状态转换操作，任何期望的转换操作都将增加特征间的距离，或增加某一特征间的尺寸——这也是算法所不允许的。当检测到这一问题时，可在顶点 V 与对象 A 中的全部支撑面之间采取一类复杂度为 $O(n)$ 的蛮力测试：若 V 位于全部支撑面的内部，则对象间相交，算法终止并返回“对象间相交”这一信息；否则，当前顶点 V 与对象 A 持有最大正距离，且 F 更新为数据面 G。更新操作将确保局部最小值不会递减至原值，其中，与数据面 F 相比，基于 G 的顶点 V 为当前相对最近点，且 G 至少与 A 中其他数据面处于同一临近水平。

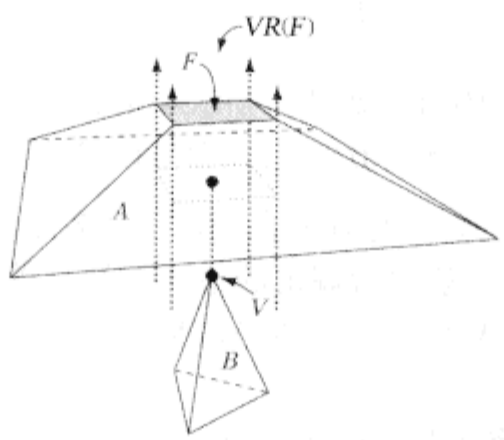


图 9.4　两个对象间的构造方式，其中 V-Clip 算法将产生局部最小值阻塞现象

针对图 9.3 中的 5 种特征组合，相应的最近特征测试也彼此不同，其详细内容过于复杂且不在本书探讨范围之列。另外，与 Lin-Canny 算法不同，在进行最近特征组合测试时，V-Clip 算法并不计算特征间的最近点，这将大大提高 V-Clip 算法的健壮性。

关于 V-Clip 的全部实现细节可参考[Mirtich98]和[Coutinho01]；健壮的算法实现代码也可通过网络得到。需要注意的是，V-Clip 算法是受到专利保护的，其美国专利号为 6054997。

9.3　层次多面体表达形式

如同包围体层次结构可加速多边形组间的查询过程，层次表达形式也可用于加速与多面体相关的查询计算。第 8 章曾采用 BSP 树创建了一类高效的层次结构。相对于层次结构方式，另一种方案是采取多面体序列且多面体之间彼此内嵌，类似于设计模式中的 Russian doll 模式。该方案始于原多面体，序列中的每一个内接多面体都可视为其外界父多面体的简化版本。在具体查询操作过程中，算法始于高度简化的内层多面体并在序列中执行遍历操作，且终止于原多面体。可以看出，序列中多面体之间的拓扑结构差异并不显著，因此，各步骤中的计算量并不大。一般情况下，查询计算的复杂度为 $O(\log n)$。

在这一类内嵌多面体表达形式中，Dobkin-Kirkpatrick 层次结构可算是最为常见的一种，9.3.1 节将对其进行介绍。其中，Dobkin-Kirkpatrick 层次结构可用于下列查询操作：在给定方向上计算极值顶点；在多面体上计算距离已知顶点的最近点，以及确定两个多面体之间的相交状态。同时，通过 Dobkin-Kirkpatrick 算法还衍生出了其他几种碰撞检测算法。例如，Dobkin-Kirkpatrick 层次结构混合算

法，[Guibas99]中的 Lin-Canny 算法，以及[Ehmann00]中的多分辨率方案。

9.3.1　Dobkin-Kirkpatrick 层次结构

d 维多面体 P 的 Dobkin-Kirkpatrick 层次结构 （DK）可看作是一个多面体序列，其中包含了多个与 P 近似的内嵌多面体。这里，$P = P_0$ 且 P_{i+1} 可通过剔除 P_i 中的顶点子集得到。另外，顶点子集将形成一个所谓的最大独立顶点集（即集合中任意两个顶点均不邻接）。最后，层次结构的构造过程终止于某一称为 d 单形体的内层多面体 P_k（在 3D 环境下为四面体）。[Edelsbrunner87]中通过简单的贪心算法给出了最大独立顶点子集的构造方法，其伪代码如下所示：

```
// Given a set s of vertices, compute a maximal set of independent vertices
Set IndependentSet(Set s)
{
    // Initialize i to the empty set
    Set i = EmptySet();
    // Loop over all vertices in the input set
    for (all vertices v in s) {
        // If unmarked and has 8 or fewer neighboring vertices...
        if (!Marked(v) && Degree(v) <= 8) {
            // Add v to the independent set and mark all of v's neighbors
            i.Add(v);
            s.MarkAllVerticesAdjacentToVertex(v);
        }
    }
    return i;
}
```

独立集是当前顶点集合的一个子集，使得 DK 层次结构的高度为 $O(\log n)$。另外，在 DK 层次结构的构造过程中，可使用相关指针信息将被删除的多面体 P_i 特征链接至新增的多面体 P_{i+1} 特征，反之亦然。这一类指针信息将有助于实现高效的层次结构查询。

图 9.5 显示了由顶点 V_0～V_{11} 定义的凸多边形 P 及其 Dobkin-Kirkpatrick 层次结构的构造过程。其中，$P_0 = P$。多边形 P_1 可通过删除顶点 V_1、V_3、V_5、V_7、V_9 以及 V_{11} 得到（亦形成了一个最大独立顶点集）；同理，P_2 也可从 P_1 中得到，即删除顶点 V_2、V_6 以及 V_{10}。由于 P_2 最终结果为 2-单形体（三角形），则当前构造过程结束。

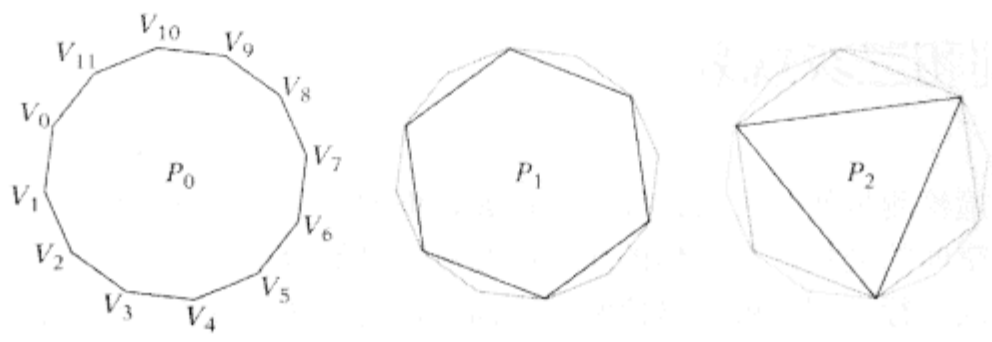


图 9.5　凸多边形 $P = P_0$ 的 Dobkin-Kirkpatrick 结构

构造完 DK 后，即可实现多种不同类型的查询操作。例如，可通过简单的递归调用计算距查询点 S 最近的顶点 P，如图 9.6 所示。

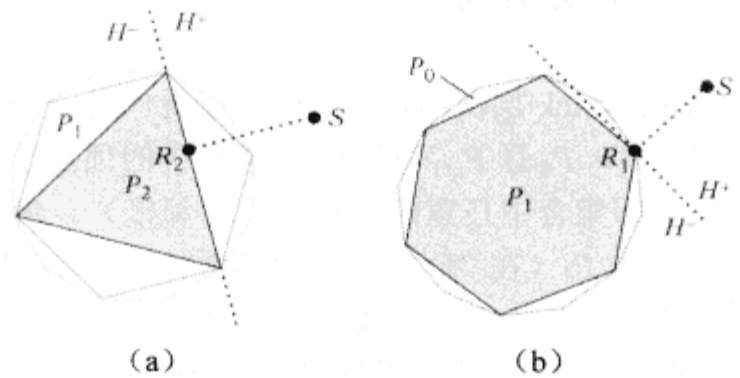


图 9.6　(a) P_2 的支撑面 H。(b) P_1 的支撑面 H，且支撑面分别经过多面体上距查询点 S 的最近点

计算始于内层多边形 P_i（$i=2$），可直接计算 P_i 上距 S 的最近点 R_i（参见 5.1.6 节）。若 $R_i = S$，则 S 位于 P 内部，算法结束；否则，已知 R_i，P_{i-1} 的最近点按如下方式计算——令 H 为 P_i 中穿越 R_i 的支撑面，则 P_{i-1} 由 H 的正、负空间构成：

$$P_{i-1}=(P_{i-1} \cap H^{+}) \cup (P_{i-1} \cap H^{-})$$

因此，最近点 R_{i-1} 位于 $P_{i-1} \cap H^{+}$ 和 S 之间，或位于 $P_{i-1} \cap H^{-}$ 和 S 之间。位于 $P_{i-1} \cap H^{-}$ 和 S 之间的最近点可通过 R_i 得到。由于 $P_{i-1} \cap \mathrm{H}^{+}$（基于层次结构的构造结果）由一个顶点和两条边定义，因此计算位于 $P_{i-1} \cap H^{+}$ 和 S 之间的最近点时，其时间复杂度为常量。结果为：P 距查询点 S 的最近点需 $O(\log n)$ 步且时间复杂度为 $O(\log n)$。

通过较为简单的递归调用，DK 层次结构也可用于计算方向 d 上的极值点。首先，通过考查内存多面体 P_k 中的全部 4 个顶点可确定其极值点 V_k。已知 V_k，仅须考查 P_{k-1} 中的若干顶点（全部为 V_k 的邻接顶点）即可确定 P_k 中的极值点。该步骤重复执行，直至到达 P_0：通过这种方式，原多面体中的极值点需要 $O(\log n)$步。可以看出，各步骤的测试时间复杂度为常量，且获取最终极值点的时间复杂度为 $O(\log n)$。

[O'Rourke98]中也给出了与 DK 层次结构相关的描述。另外，同步遍历两个层次结构可使两个凸多边形以及凸多面体间的相交测试时间复杂度分别达到 $O(\log n)$和 $O(\log^2 n)$级别（[Dobkin90]）。

最后，针对多面体的极值点问题，可考虑使用[Eberly04]中建议的基于 BSP 树的加速结构来替代 DK 层次结构，前者更易于使用且占用更少的内存空间。

9.4　线性规划和二次规划

凸多面体间的碰撞检测虽然可视为计算几何问题，但也可作为数学优化问题加以处理。例如，可将多面体记作多个半空间的交集，多面体的布尔相交测试则等价于线性规划中的可行性问题，而后者将针对一组线性不等式进行求解。就碰撞检测问题而言，问题求解过程转化为计算两个多面体全部半空间中的一点（每一个半空间分别对应一个线性不等式）。另外，不仅是多面体间的相交测试问题，计算分离多面体间的间隔距离以及最近点都可视为一类优化问题，即二次规划问题。综上所述，无论

是线性规划问题还是二次规划问题，它们都是一类与线性不等式约束相关的优化操作。在线性规划问题中，其最终目标是针对相关约束获取一个最优线性函数；而二次规划问题则是得到一个最优的二次函数。后续小节将详细讨论如何将线性规划和二次规划问题表示为碰撞检测问题，并对相关问题的算法加以概述。

9.4.1 线性规划

定义线性函数为 $f(x_1, x_2,\cdots, x_n) = c_1x_1+c_2x_2 +\cdots+ c_n x_n$，其中 c_i 项为实数常量；线性约束可分别表示为 $f(x_1, x_2,\cdots, x_n) = c$，$f(x_1, x_2,\cdots, x_n)\leqslant c$ 或 $f(x_1, x_2,\cdots, x_n)\geqslant c$。因此，线性规划问题将基于线性约束有限集对当前线性函数进行优化（求解最大值或最小值）。该函数亦称作目标函数。不失一般性，这里所讨论的线性规划问题（LP）将计算目标函数的最大值并仅限于形如 $f(x_1, x_2,\cdots, x_n)\leqslant c$ 的约束条件（$f(x_1, x_2,\cdots, x_n)\leqslant c$ 可写作 $-f(x_1, x_2,\cdots, x_n)\leqslant -c$）。另外，如果需要，可通过 $f(x_1, x_2,\cdots, x_n)\leqslant c$ 和 $f(x_1, x_2,\cdots, x_n)\geqslant c$ 来表示等式约束条件。

从几何角度上讲，包含 n 个变量的线性不等式在 $\mathbb{R}^n$ 内定义了一个半空间。因此，由 m 个不等式定义的有限集，其解包含的区域将与 m 个半空间的交集 H_i 相对应，其中 $1\leqslant i\leqslant m$。同时，该半空间的交集即为凸多面体 P，且 $P= H_1 \cap H_2 \cap\cdots\cap H_m$。还需要强调的一点是，这里的目标函数可视为向量 $\mathbf{x} = (x_1, x_2,\cdots, x_n)$ 和 $\mathbf{c} = (c_1, c_2,\cdots, c_n)$ 之间的点积。基于此，LP 问题从几何观点上则等价于：在方向 $\mathbf{c}$ 上计算 P 中的支撑点 $\mathbf{x}$（实际上，由于 P 为凸多面体，支撑点 $\mathbf{x}$ 可选为支撑顶点）。

在其他专业书籍中，LP 问题常被抽象为矩阵-向量格式，而非针对多面体的几何优化操作。在矩阵形式中，将计算目标函数 $\mathbf{c}^T\mathbf{x}$ 的最大值（或最小值），其中，$\mathbf{c}$ 表示包含 d 个常量的列向量；$\mathbf{x}= (x_1, x_2,\cdots, x_d)^T$ 且满足约束条件 $\mathbf{Ax}\leqslant\mathbf{b}$（其中，$\mathbf{A}$ 表示为一个 $n \times d$ 的系数矩阵；$\mathbf{b}$ 为一个包含 n 个常量的列向量）。在本书中，关于 LP 问题将采纳几何解释。

当给定两个凸多面体 A 和 B 后，可分别将其视为半空间的交集。这里，采用 A、B 相交体空间，即凸多面体 C 表示来自 A、B 半空间的交集。当且仅当线性约束条件为 A、B 定义的半空间且该 LP 问题有解时，A、B 间处于碰撞状态。由于两多面体之上的公共点可视为其碰撞的直接依据，因而可随意选择相应的目标函数。例如，$\mathbf{c} = (1, 0,\cdots, 0)$。

下面的示例显示了如何利用 LP 问题解决多面体相交测试。如图 9.7 所示，考查三角形 A 和 B，其中，三角形 A 由半空间 $-x + y\leqslant -1$、$-x - 4y\leqslant -1$ 以及 $4x + y\leqslant 19$ 确定；三角形 B 由 $-4x + y\leqslant 0$、$x - 4y\leqslant 0$ 以及 $x + y\leqslant 5$ 确定。因此，若下列 LP 问题有解，则三角形之间相交：

$$
\begin{aligned}
&\text{最大值：} x \\
&\text{条件：} -x + y\leqslant -1 \\
&\qquad -x - 4y\leqslant -1 \\
&\qquad 4x + y\leqslant 19 \\
&\qquad -4x + y\leqslant 0 \\
&\qquad x - 4y\leqslant 0 \\
&\qquad x + y\leqslant 5
\end{aligned}
$$

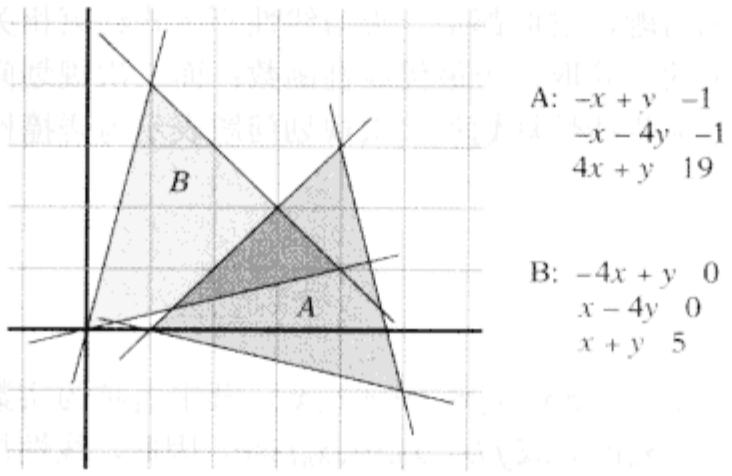


图 9.7　三角形 $A=\{(1,0),(5,-1),(4,3)\}$和 $B=\{(0,0),(4,1),(1,4)\}$分别定义为 3 个半空间的交集

由图 9.7 可知，该 LP 问题有解且当 $x=4$ 时存在最大值。

另外，几何体间的相交问题也可按如下方式表示为 LP 问题：以顶点的格式定义两个多面体。则当前线性规划问题为：确定是否存在一个平面系数以在两个多面体间形成一个分割面。如图 9.7 所示，针对约束条件 $f(A_i)\leqslant 0$，$f(B_i)>0$，该问题可表示为：确定 a、b、c 以获取分割线 $f(x,y)=ax+by+c$。其中，A_i 和 B_i 为多面体上的顶点。特别地，若下列 LP 问题有解，则两个三角形之间不相交：

$$
\begin{aligned}
&\text{最大值：} a\\
&\text{条件：} a+c\leqslant 0\\
&\qquad 5a-b+c\leqslant 0\\
&\qquad 4a+3b+c\leqslant 0\\
&\qquad c>0\\
&\qquad 4a+b+c>0\\
&\qquad a+4b+c>0
\end{aligned}
$$

现已知三角形 A、B 相交，则该 LP 问题无解。

在求解线性规划问题方面，较为著名的算法是 Danzig 单形体算法，参见[Chvatal83]。从前面的例子可以看出，当 LP 问题具有较低的维度以及较少的约束条件时，也可使用一类更为简洁的算法。这里将要讨论的则是一类基于[Seidel90]的随机算法。另外，在介绍 Seidel 算法之前，首先引入 Fourier–Motzkin 算法。该算法针对不等式约束集是否存在有效解这一问题提供了一种简单方案（同时，Fourier-Motzkin 消去法也是求解小型方程组时的一种常用方法）。

9.4.1.1　Fourier–Motzkin 消去法

对于线性不等式方程组的求解问题，一种简单的算法是 Fourier-Motzkin 消去法。该方法在计算过程中将连续改写相关方程组，并在每一次改写时消去方程组中的一个变量。当仅剩一个变量时，若变量的最大下界小于其最小上界，则该方程组有解。这里，令方程组包含 n 个变量、m 个约束条件，如下所示：

$$
\begin{aligned}
&a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n\leqslant b_1\\
&a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n\leqslant b_2\\
&\cdots\\
&a_{m1}x_1+a_{m2}x_2+\cdots+a_{mn}x_n\leqslant b_m
\end{aligned}
$$

这里，选取变量 x_k 用于消元且各不等式针对 x_k “求解”，x_k 上的下界 s 和上界 t 如下所示：

$$L_1(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n) \leqslant x_k \qquad x_k \leqslant U_1(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n)$$
$$L_2(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n) \leqslant x_k \qquad x_k \leqslant U_2(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n)$$
$$\cdots \qquad\qquad \cdots$$
$$L_s(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n) \leqslant x_k \qquad x_k \leqslant U_t(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n)$$

由于 x_k 的下界须小于等于其上界值，因而可采用由 x_k 的上界、下界组合构成的 st 不等式集来替换当前不等式集，从而完全消去 x_k。如下所示：

$$L_i(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n) \leqslant U_j(x_1, x_2, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n)$$

其中，$1 \leqslant i \leqslant s$，$1 \leqslant j \leqslant t$。

该过程重复执行，且在每次循环过程中除去一个变量，直至最终剩下一个变量。为了进一步帮助读者理解该算法，再次考查图 9.7 中两个三角形代表的 6 个约束条件，且针对变量 x，不等式集变为：

$$x \geqslant 1 + y \qquad x \geqslant y/4$$
$$x \geqslant 1 - 4y \qquad x \leqslant 4y$$
$$x \leqslant (19 - y)/4 \qquad x \leqslant 5 - y$$

调整不等式的顺序，则 x 的上界（左侧）、下界（右侧）如下所示：

$$y/4 \leqslant x \qquad x \leqslant 4y$$
$$1 + y \leqslant x \qquad x \leqslant 5 - y$$
$$1 - 4y \leqslant x \qquad x \leqslant (19 - y)/4$$

由于 x 的下界须小于等于其上界，则通过 x 的上界、下界组合构造全部 9 个约束条件，即可完全消除 x 项，如下所示：

$$y/4 \leqslant 4y \qquad 1 + y \leqslant 4y \qquad 1 - 4y \leqslant 4y$$
$$y/4 \leqslant 5 - y \qquad 1 + y \leqslant 5 - y \qquad 1 - 4y \leqslant 5 - y$$
$$y/4 \leqslant (19 - y)/4 \qquad 1 + y \leqslant (19 - y)/4 \qquad 1 - 4y \leqslant (19 - y)/4$$

经过相应的化简后，得到下列与 y 相关的约束条件：

$$-4/3 \leqslant -1 \leqslant 0 \leqslant 1/8 \leqslant 1/3 \leqslant y \leqslant 2 \leqslant 3 \leqslant 4 \leqslant 19/2$$

则 y 的最小范围是：

$$1/3 \leqslant y \leqslant 2$$

上述有效范围使得原方程组有解，因而两个三角形之间存在一个（或多个）公共点。可以发现，该有效范围与图 9.7 中的示例内容相吻合。实际上，它定义了相交体在 y 轴上的一个投影区间。

类似地，可在该区间内选取一个 y 值并代入原不等式集，即可得到 x 的有效区间并选取 x 值。最终，坐标(x,y)表示为相交体内部的一点。此处，令 $y = 2$，则 x 的有效区间如下所示：

$$-7 \leqslant 1/2 \leqslant 3 \leqslant x \leqslant 3 \leqslant 17/4 \leqslant 8$$

最终 $x = 3$。如图 9.7 所示，(3,2)即为相交体的某一点。

虽然 Fourier-Motzkin 消去法易于理解并且实现起来也很方便，但在变量的消去过程中，相应的不等式数量也会急剧增长。在最坏情况下，每次迭代过程中不等式的数量将以平方级增长，并导致指数级的复杂度。当然，这一问题可通过某些启发式规则加以改善，例如，可选择消去生成较少不等式的变量。考虑到多数引入不等式的冗余性，移除这一类不等式将有效地抑制指数增长，具体内容可参考[Imbert93]。尽管存在指数级增长这一缺陷，Fourier-Motzkin 消去法仍不失为一类有意义的算法（对于

小型方程组），且该算法特别适用于并行计算。同为计算多面体的顶点数据，与 Fourier-Motzkin 消去法十分相近的则是双重描述（double description）算法，参见[Motzkin53]。

9.4.1.2 Seidel 算法

针对包含 d 个变量、m 个约束条件的线性规划问题，一种较为简洁的算法是 Seidel 算法。Seidel 算法的运行时间可达到期望的 $O(d!m)$级别。当 d 值较大时，该算法并不实用；而对于较小的 d 值（如 $d \leqslant 3$ 时）——即多数碰撞检测问题所关注的维度——该算法则呈现出其高效的一面。首先，为了方便起见，LP 问题按下列方式重新加以描述：定义 m 个 d 维半空间集合$\{H_1, H_2, \cdots, H_m\}$，半空间交集（也可能是空集）将形成一个多面体 P，且 $P = H_1 \cap H_2 \cap \cdots \cap H_m$。另外，还需在空间 $\mathbb{R}^d$ 内定义一个目标向量 $\mathbf{c}$。则问题变为：计算 P 中方向 $\mathbf{c}$ 上的极值点 $\mathbf{v}$（抑或证明 P 为空集时，该问题无解）。

究其本质，Seidel 算法采取了渐进式增长这一方式，算法针对半空间进行循环操作，且在每一次循环中根据前次循环中多面体 P_{i-1} 的顶点 $\mathbf{v}_{i-1}$ 计算多面体 $P_i = H_1 \cap H_2 \cap \cdots \cap H_i$ 的最优顶点 $\mathbf{v}_i$。在下列两种情况下，循环操作将会引入新的半空间（如图 9.8 所示）。

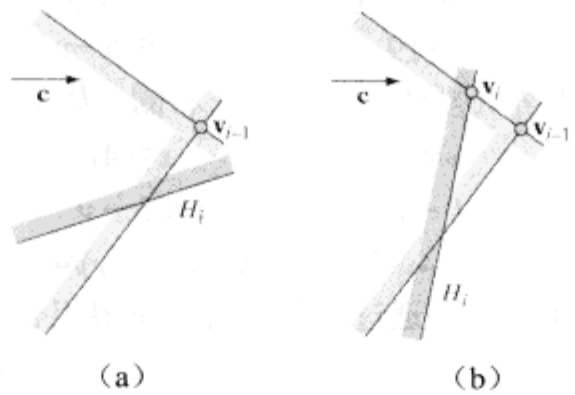


图 9.8 （a）$\mathbf{v}_{i-1}$ 包含于 H_i，因此 $\mathbf{v}_i = \mathbf{v}_{i-1}$。（b）$\mathbf{v}_{i-1}$ 与 H_i 冲突，因而 H_i 必位于 H_i 包围超平面上的某处

首先，当 $\mathbf{v}_{i-1}$ 位于 H_i 内部时，$\mathbf{v}_{i-1}$ 为有效顶点且无须执行其他操作（即 $\mathbf{v}_i = \mathbf{v}_{i-1}$）；另一方面，若 $\mathbf{v}_{i-1}$ 位于 H_i 外部，则 $\mathbf{v}_{i-1}$ 为无效顶点。此时，$\mathbf{v}_i$ 则包含于 H_i 的包围超平面 π 中（否则，与顶点 $\mathbf{v}_i$ 相比，方向 $\mathbf{c}$ 上、线段 $\mathbf{v}_i$-$\mathbf{v}_{i-1}$ 间将存在另一极值点，从而导致矛盾）。如下所示，可将当前 LP 问题降至 $d-1$ 维以计算 $\mathbf{v}_i$。令 $\mathbf{c}'$为 $\mathbf{c}$ 在平面 π 上的投影，且令 G_i 为 H_j（$1 \leqslant j < i$）与平面 π 之间的交集。则求解 $\mathbf{v}_i$ 可归于 G_i 约束条件、目标向量 $\mathbf{c}'$所确定的 $d-1$ 维 LP 问题。$\mathbf{v}_i$ 可采用递归方式加以计算且 $d = 1$ 时递归结束。在 1D 情况下，$\mathbf{v}_i$ 可视为满足全部约束条件的最大值或最小值（取决于方向向量 $\mathbf{c}'$）。同时，根据约束的对立条件，还可以检测到当前 LP 问题的无效解。

在算法开始之前，需要定义一个初始变量 $\mathbf{v}_0$，且该值可通过引入一个相交实体 P 的包围盒 P_0 得到。由于 $P_0 \cap P$ 持有 P 的最优顶点，则在每次循环过程中，该问题可重新表示为计算多面体 $P_i = P_0 \cap H_1 \cap H_2 \cap \cdots \cap H_i$ 的最优顶点 $\mathbf{v}_i$（源于上一次循环中多面体 P_{i-1} 的顶点 $\mathbf{v}_{i-1}$）。因此，初始变量 $\mathbf{v}_0$ 简单地对应于 P_0 中、方向 $\mathbf{c}$ 上的极值点（相关赋值操作基于向量 $\mathbf{c}$ 中的坐标分量）。

[Seidel90]表明，该算法的期望运行时间为 $O(d!m)$。另外，Seidel 算法是一类随机算法，亦即，既定时间复杂度后的具体计算过程将依赖于半空间初始状态下的随机顺序，其随机状态将严重影响算法的实际性能。

针对图 9.7 中的三角形，图 9.9 显示了 Seidel 算法的具体计算过程。第一幅图显示了初始状态下的 6 组半空间数据，其余图像则分别显示了经 $H_1 \sim H_6$ 处理后的 P_0 状态。在每一个循环步骤中，处理后的

半空间将采用黑实线标注，同时还包括对应顶点 $\mathbf{v}_i$。在给定的（随机）半空间顺序中，将出现 3 次 1D 递归调用：H_2、H_5 和 H_6。对应的包围超平面内的 1D 约束条件则采用箭头加以标注。

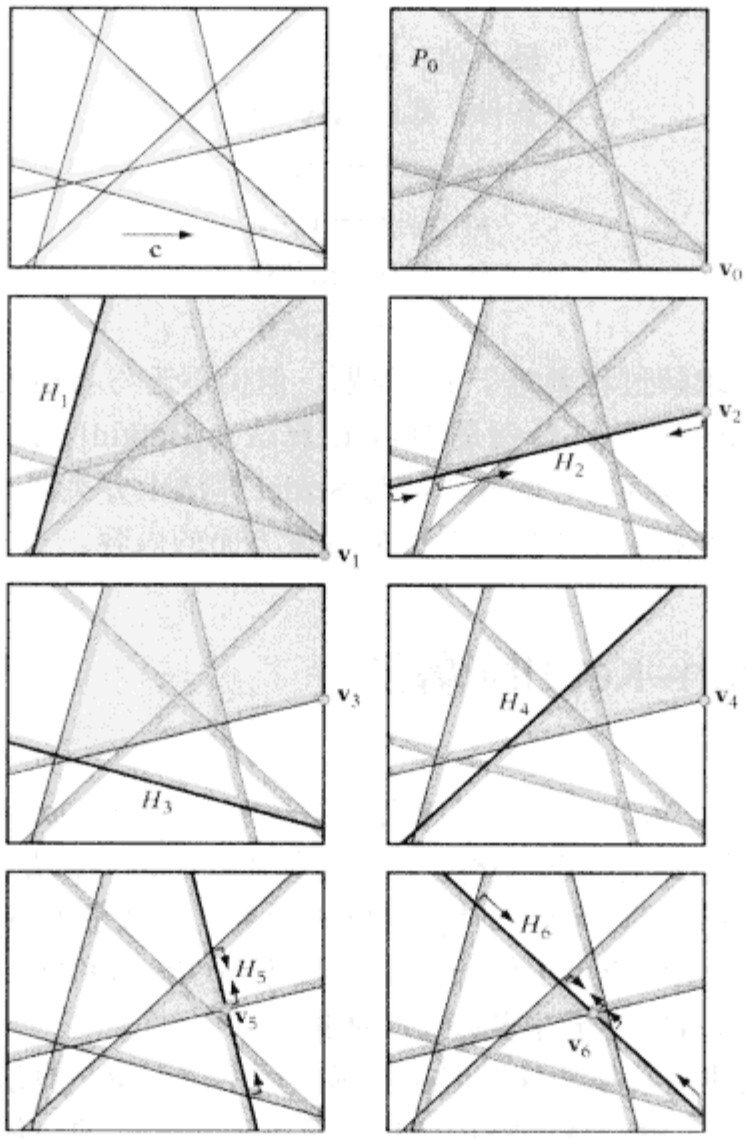


图 9.9　左上图显示了图 9.7 中的 6 个半空间。其余内容则针对这 6 个半空间使用了 Seidel 算法。其中，有效区域显示为浅灰色，当前半空间则使用黑线表示。相应的箭头为 1D 约束条件

9.4.2　二次规划

虽然线性规划方案可有效地确定多面体间的相交状态，但当两个多面体处于分离状态时，该方案并不返回任何相关信息。在某些情况下，常需要获取两多面体间的最近点数据，以计算二者间的距离。这里，多面体间的距离问题可表示为二次规划（QP）问题。类似于 LP 问题，二次规划问题同样包含线性约束条件，只是目标函数呈现二次非线性状态。针对多面体间的距离问题，一类可行的 QP 方案为：令两个分离多面体 A 和 B 分别定义为$\mathbb{R}^d$ 内、对应于顶点有限集的两个凸体。

$$A = CH(S_A), \qquad S_A = \{A_1, A_2, \cdots, A_m\}$$
$$B = CH(S_B), \qquad S_B = \{B_1, B_2, \cdots, B_n\}$$

则当前问题变为：计算 A 中的顶点 P 以及 B 中的顶点 Q，并使 P、Q 间的距离（或（列）向量 $\mathbf{v}=$

$P-Q$ 的长度值）最小化。这里，令 $\mathbf{G}$ 为 $d\times m$ 尺寸的矩阵且（列）包含 S_A 中的顶点；令 $\mathbf{H}$ 为 $d\times n$ 尺寸的矩阵且（列）包含 S_B 中的顶点，多面体间的距离问题则转化为在变量 $\mathbf{x}=(x_1, x_2,\cdots, x_m)^T$ 和 $\mathbf{y}=(y_1, y_2,\cdots, y_n)^T$ 中计算下列 QP 问题：

$$
\begin{aligned}
&\text{最小值：}\ \mathbf{v}^T\mathbf{v}\\
&\text{条件：}\ \mathbf{Gx}-\mathbf{Hy}=\mathbf{v}\\
&x_1+x_2+\cdots+x_m=1\\
&y_1+y_2+\cdots+y_n=1\\
&x_1, x_2,\cdots, x_m\geqslant 0\\
&y_1, y_2,\cdots, y_n\geqslant 0
\end{aligned}
$$

通常情况下，QP 问题的求解过程要难于 LP 问题，但仍不乏与其相关的多种算法。根据碰撞检测问题的规模，两种较具实用价值的算法分别是 Botkin 算法（[Botkin]、[Botkin95]）以及 Gärtner 算法（[Gärtner00]）。就线性规划问题而言，两种算法与 Seidel 算法十分类似。另外，Botkin 算法的期望运行时间为 $O(dd!\,n)$，而 Gärtner 算法则未提及时间复杂度方面的内容。

9.5 Gilbert–Johnson–Keerthi 算法

针对多面体间的相交问题，最为有效的算法之一便是由 Gilbert、Johnson 和 Keerthi 三人共同开发的一类迭代算法，简称 GJK 算法（[Gilbert88]）。如前所述，GJK 算法是一种基于单形体的下降算法，算法定义了两个输入顶点集并计算该集合凸体间的欧式距离（以及最近点）。通常情况下，GJK 算法的应用范围不仅仅是多面体，还可应用于任意凸体顶点集（[Gilbert90]）。本章中的 GJK 示例算法均无特定限制，其通用版本亦可直接用于非多边形凸体集。

虽然 GJK 算法的原文描述颇具技巧性，但在实际应用中，无论是算法自身还是相关实现，其具体操作过程并不十分复杂。深入了解凸体的特征分析以及算法所采用的凸体集理论依据，将有助于更好地理解算法。

需要强调的一点是，GJK 算法本身并不对一组输入对象进行任何实质性的操作，而是在对象间计算 Minkowski 差。相关问题的转化使得两个凸体集间的距离计算减为求解单一凸体集内的距离。GJK 算法在每次循环迭代过程中将搜索 Minkowski 差对象并获取一个子空间体，且该空间体均为单形体。另外，算法并不显式地计算 Minkowski 差，而是根据具体要求并通过一个支撑映射函数对 Minkowski 差采取抽样计算。上述概念（Minkowski 差、单形体以及支撑映射函数）可参考第 3 章。这里假定读者对相关内容已十分了解。

9.5.1 算法概述

如前所述，通过计算两多面体间的欧式距离，GJK 算法可有效地解决二者间的相交问题。该算法基于下列原理：两多面体 A 和 B 的间距等价于其 Minkowski 差 C（$C=A\ominus B$）与原点之间的距离（如图 9.10 所示）。因此，问题简化为求解 C 上距原点的最近点。考虑到显式计算 Minkowski 差的复杂性，算法本身似乎并未获得较大的改观。但 GJK 算法的关键之处在于它并不显式地计算 Minkowski 差，算

法只是通过支撑映射函数 $C = A \ominus B$ 抽样选取 Minkowski 差顶点集。特别地，支撑函数 $s_A(\mathbf{d})$将给定方向 $\mathbf{d}$ 映射至该方向上、凸体对象 A 中的某一支撑点。由于支持函数反映了线性函数的最大值，则针对 C 的支持映射 $s_{A\ominus B}(\mathbf{d})$，其与 A、B 相关的支持映射可表示为 $s_{A\ominus B}(\mathbf{d}) = s_A(\mathbf{d}) - s_B(-\mathbf{d})$。因此，可根据具体要求从多面体 A 和 B 中计算相应的 Minkowski 差顶点。

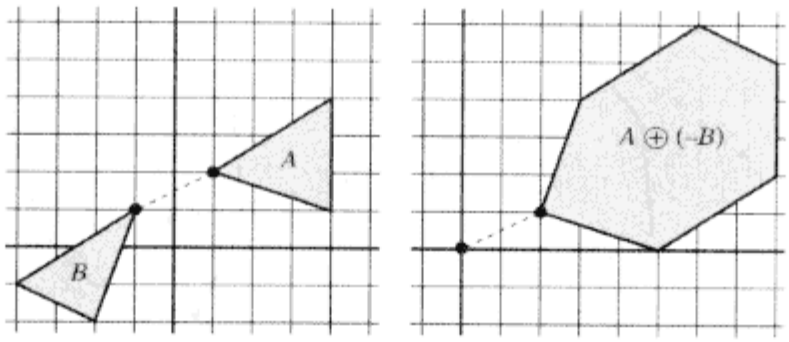


图 9.10　A 和 B 之间的距离等价于 Minkowski 差与原点之间的距离

针对距原点最近点的 Minkowski 差搜索操作，GJK 算法采用了 Carathéodory 定理（[Rockafellar96]）。该定理表明，针对$\mathbb{R}^d$空间内的凸体 H，H 中的各顶点可表示为至多 $d+1$ 个顶点的凸组合：在每次循环迭代过程中，通过维护 Minkowski 差 C 中的、由（最多）$d+1$ 个顶点构成的集合 Q，将使 GJK 算法对 C 实施有效的搜索。最终，Q 中的凸体在 C 中形成了一个单形体。若原点包含于当前单形体中，A 和 B 之间处于相交状态且算法结束；否则，更新集合 Q 以形成一个新的单形体，并确保其所含顶点比当前单形体更接近于原点。该处理过程终止于某一集合 Q，且该集合包含距原点的最近点。若 A 和 B 之间处于分离状态，二者的最小间距则可通过 $CH(Q)$（即 Q 中的凸体）中的最小范数顶点得到。下列 GJK 算法步骤显示了集合 Q 的更新方式：

1．将集合 Q 初始化为 A 和 B 的 Minkowski 差中的一个或多个顶点（最多为 $d+1$ 个顶点，其中 d 表示维度）。

2．计算 $CH(Q)$中具有最小范数的顶点 P。

3．若点 P 为原点，则原点必定位于 A 和 B 的 Minkowski 差中。计算终止并返回“A、B 相交”这一信息。

4．将 Q 减至其最小子集 Q'，以使 $P \in CH(Q')$。亦即，针对 Q 中不包含顶点 P 的子单形体，移除其顶点。

5．令 $V = s_{A\ominus B}(-P) = s_A(-P) - s_B(P)$为$-P$ 方向上的支撑点。

6．与顶点 P 相比较，若点 V 在方向$-P$ 上并非极值点，则计算结束并返回“A 和 B 不相交”这一信息。针对由原点指向顶点 P 的向量，其长度值即为 A、B 之间的距离。

7．否则，将点 V 添加至集合 Q 中并返回步骤 2。

图 9.11 显示了 2D 环境下 GJK 算法针对多边形上距原点 O 的最近点计算过程。算法始于顶点 A 且初始状态下的单形体集合 $Q = \{A\}$。对于只包含一个顶点的单形体，该顶点即是距原点的最近点。随后，在$-A$ 方向上搜索支撑点且得到顶点 B，将顶点 B 添加至单形体集中：$Q = \{A, B\}$。在 $CH(Q)$上距原点的最近点为顶点 C。由于需要将 A、B 作为凸组合用以表示 C，则二者皆须置于 Q 中。此处，顶点 D 为$-C$ 方向上的支撑点并添加至集合 Q 中，且有 $Q = \{A, B, D\}$。当前，$CH(Q)$上距原点的最近点为顶点 E。由于在 Q 中需要将 B、D 作为顶点凸组合用以表示 E，因此 Q 更新为 $Q = \{B, D\}$。在$-E$ 方向上的支

撑点为 F，且添加至 Q 中。随后，$CH(Q)$上距原点的最近点为 G。同样，D、F 为 Q 中最小顶点集且需要将其作为凸组合表示 G，Q 更新为 $Q=\{D,F\}$。此处，在方向$-G$ 上不存在比 G 距原点更近的顶点，因此算法结束。

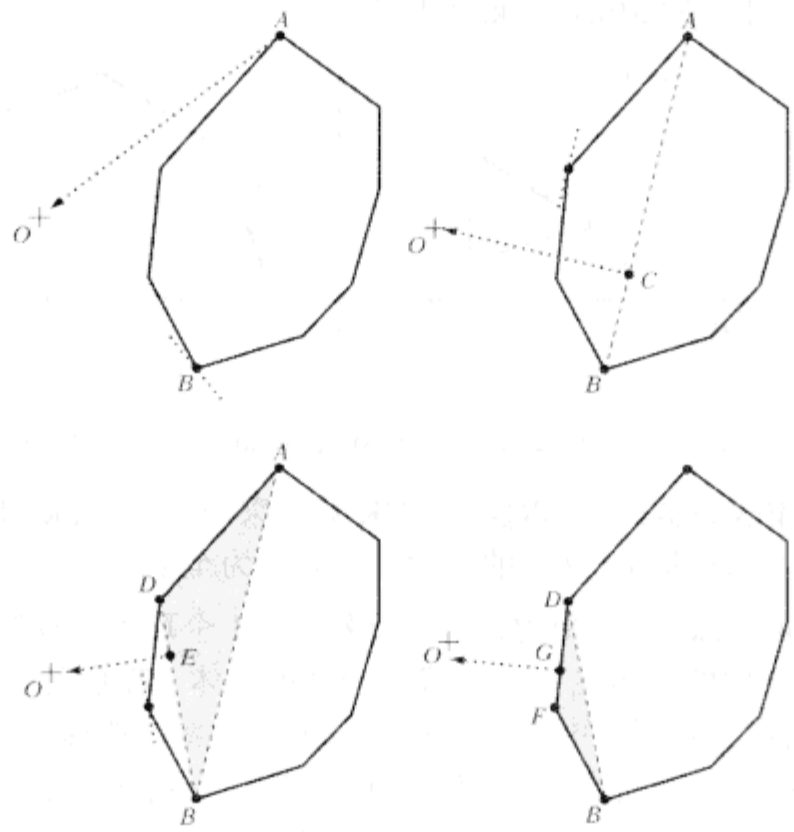


图 9.11　采用 GJK 算法计算某一多边形上距原点最近的顶点

通过将内部多面体间的距离与其球体扩展半径和进行比较，GJK 算法还可用于快速处理球体扩展多面体。还需强调的一点是，如果采用 GJK 算法对非凸多面体顶点集进行计算，则需在非凸多面体中的凸体间计算最小间距。

虽然上述算法分析仅演示了多面体的计算过程，但 GJK 算法适用于任意凸体类型（[Gilbert90]）。由于通常情况下输入凸体将通过其支撑映射进行采样计算，因此采用通用凸体对象的 GJK 算法时，需要提供适当的通用映射函数（参见 3.8 节）。

针对多面体操作，GJK 算法在若干步骤中即可结束并返回间距数据。然而，对于任意凸体类型，该算法一般以渐进方式收敛于某一间距值。因此，应在算法结束条件上添加相应的容值，以在计算非多面体对象时使算法能够正常结束。

当对象处于分离状态时，GJK 算法除了要返回间距值，可能还需要计算输入对象间的最近点：可从 A、B 对象顶点内、最后一次单形体集 Q 的顶点数据中选择并计算。其详细计算步骤可参见 9.5.3 节。

读者如果想详细了解 GJK 算法，可参考[Bergen03]以及[Coutinho01]中的相关内容。另外，GJK 算法最初是采用 FORTRAN 语言加以实现的，读者也可通过网络获取其 C 语言版本的代码实现。

9.5.2　计算单形体内的最小范数顶点

下面针对单形体集合 $Q=\{Q_1,Q_2,\cdots,Q_k\}$（$1\leqslant k\leqslant 4$），计算 $CH(Q)$中的最小范数顶点 P。在 GJK 算

法的原型中，这一过程是通过一个称作“距离子算法（或 Johnson 算法）”的函数加以实现的，距离子算法将单独考查 Q 中的全部子集。例如，当 $k=4$ 时，共有 15 个子集：4 个顶点(Q_1，Q_2，Q_3，Q_4)；6 条边(Q_1Q_2，Q_1Q_3，Q_1Q_4，Q_2Q_3，Q_2Q_4，Q_3Q_4)；4 个数据面($Q_1Q_2Q_3$，$Q_1Q_2Q_4$，$Q_1Q_3Q_4$，$Q_2Q_3Q_4$)以及单形体($Q_1Q_2Q_3Q_4$)内部区域。这里将以尺寸升序的方式逐一对子集进行搜索。当原点位于当前被测子集相应特征（顶点、边或数据面）的 Voronoi 域时（或者当前子集对应于单形体的内部区域时，原点位于单形体内部），搜索过程结束。一旦确定了相关特征，则该特征上的最小范数顶点可通过原点在该特征上的正交投影得到。

考查下面的例子：$Q=\{Q_1, Q_2, Q_3, Q_4\}$。针对顶点 Q_1，任意顶点 P 位于 Voronoi 域，当且仅当满足下列不等式条件：

$$(P-Q_1)\cdot(Q_2-Q_1)\leqslant 0$$
$$(P-Q_1)\cdot(Q_3-Q_1)\leqslant 0$$
$$(P-Q_1)\cdot(Q_4-Q_1)\leqslant 0$$

当且仅当满足下列不等式条件时，顶点 P 位于与边 Q_1Q_2 关联的 Voronoi 域：

$$(P-Q_1)\cdot(Q_2-Q_1)\geqslant 0$$
$$(P-Q_2)\cdot(Q_1-Q_2)\geqslant 0$$
$$(P-Q_1)\cdot((Q_2-Q_1)\times\mathbf{n}_{123})\geqslant 0$$
$$(P-Q_1)\cdot(\mathbf{n}_{142}\times(Q_2-Q_1))\geqslant 0$$

其中：

$$\mathbf{n}_{123}=(Q_2-Q_1)\times(Q_3-Q_1)$$
$$\mathbf{n}_{142}=(Q_4-Q_1)\times(Q_2-Q_1)$$

若点 P 并不处于顶点或边的 Voronoi 域，则需要查看点 P 以及 Q 中的其余顶点是否位于 Q 中 3 个顶点所构成的数据面平面的另一侧，从而对数据面的 Voronoi 域进行测试。例如，当且仅当满足下列不等式条件时，点 P 位于 $Q_1Q_2Q_3$ 的 Voronoi 域中：

$$((P-Q_1)\cdot\mathbf{n}_{123})((Q_4-Q_1)\cdot\mathbf{n}_{123})<0$$

其中：

$$\mathbf{n}_{123}=(Q_2-Q_1)\times(Q_3-Q_1)$$

可定义类似的不等式集以测试其他顶点、边以及数据面的 Voronoi 域包含关系。多数不等式结果可在其他 Voronoi 域之间实现共享且无须重复计算，这将使整体测试过程更加高效。另外，上述方法可较好地处理小于 4 个顶点的单形体集合。

在 5.1.2 节、5.1.5 节以及 5.1.6 节中，当计算线段、三角形以及四面体上距给定点的最近点时，也采用了同样的算法，这也使得本节中的内容似曾相识。实际上，GJK 算法的最初实现即采用了这一类基于特例的优化算法计算距离子算法（[Ruspini]）。

9.5.3 GJK 算法、最近点以及接触流形

GJK 算法不仅可以计算间隔距离，还能够返回对象间的一组最近点数据。当对象间处于分离状态时，集合 Q 中最多包含 3 个顶点以构成 Minkowski 差 $C=A\Theta B$ 所表示的单形体。这里，令 k 表示为 Q

中的顶点数量，P 记为 $CH(Q)$中的最小范数顶点。由于 Q 中包含了作为凸组合表示 P 的最小顶点数，因此顶点 P 如下所示：

$$P=\sum_{1\leqslant i\leqslant k} c_i Q_i, \sum_{1\leqslant i\leqslant k} c_i=1，\ c_i>0$$

其中，$(c_1, c_2,\cdots, c_k)$定义为基于顶点 Q_i 的 P 的质心坐标。现已知顶点 Q_i 为输入对象 A、B 中顶点 A_i 与 B_i 之差，则 P 可改写为：

$$P=\sum_{1\leqslant i\leqslant k} c_i(A_i-B_i)=G-H，其中 G=\sum_{1\leqslant i\leqslant k} c_i A_i，\ H=\sum_{1\leqslant i\leqslant k} c_i B_i$$

由于可将 P 视为 A、B 间的最小间距，因此 A、B 边界上必存在顶点（并不唯一），其差为 P。这里，令其为 G 和 H。因此，通过维护顶点 A_i 和 B_i 可直接获取 A、B 间的一组最近点。P 与 Q 中凸体间的位置关系（P 是否位于凸体之上）可视为计算基于顶点 Q_i 的 P 的质心坐标，而质心坐标的计算则并非难事。

针对两个碰撞（或者准碰撞）多面体间的 GJK 算法，除了最终返回一组最近点数据，算法还可派生出单形体间的接触流形计算，其详细内容可参见[Zhang95]和[Nagle02]。

9.5.4 利用爬山法计算极值顶点

在 GJK 算法中，较为耗时的步骤之一即是在给定方向上确定极值点。一般来讲，通过遍历全部 n 个顶点，可在 $O(n)$时间内获取极值点。一类较好的算法是定义一个对应的数据结构，并存储各顶点的全部邻接顶点。随后，即可通过简单的爬山法尽可能地访问多个候选极值点，以获取最终的极值点，例如，对于顶点 V 而言，算法将针对其邻接顶点 V_i 在给定方向上计算相对于 V 的最终极值点。此时，当前顶点更新为 V_j 且重复执行上述搜索过程。考虑到多面体的凸体特性，若顶点的邻接顶点中未发现极值点，则当前顶点即为最终的极值点。该算法的高效之处在于：在极值点的搜索过程中，将生成较短的顶点路径。

图 9.12（a）显示了一个针对十二面体的爬山算法示例。其中，当前顶点 V 包含 3 个邻接顶点 V_0，V_1 和 V_2。与顶点 V 相比，由于$(V_0-V)\cdot\mathbf{d}>0$，则在搜索方向 $\mathbf{d}$ 上，V_0 为极值点且更新为当前顶点。可采取类似的方式（对应的 3 个步骤在图中以黑箭头加以标注）直至确定最终的顶点 E。相对于顶点 E，由于其邻接顶点中未发现新的极值点，则在方向 $\mathbf{d}$ 上，顶点 E 为最终的极值点。

对于大型多面体，可在顶点的邻接表中添加一个或多个邻接点，以加速爬山法的计算过程。通过将路径链接至距当前顶点较远的顶点（沿两顶点的构造边搜索，以获取最少的计算步骤），爬山法在搜索过程途经这一类添加顶点时，即可加速极值点的路径处理过程。一类较为简单的处理方案是随机添加邻接点，而更为智能的方法是使添加点与当前顶点之间的距离值至少为 k。为了保证添加节点确实被爬山算法所访问，较好的方法是将这一类节点在邻接顶点表中的位置前置。另外，可采用一种最速下降算法，即算法不再移至首个候选极值点，而是检测全部邻接点并直接将当前顶点更新为邻接极值点。如图 9.12（b）所示，算法针对初始顶点 V 添加了两个邻接点并使用了最速下降算法。其中，只需两步即可到达顶点 E。即便如此，算法的计算量并未得到显著改观：算法在每一步中都需要检测大量的邻接顶点。

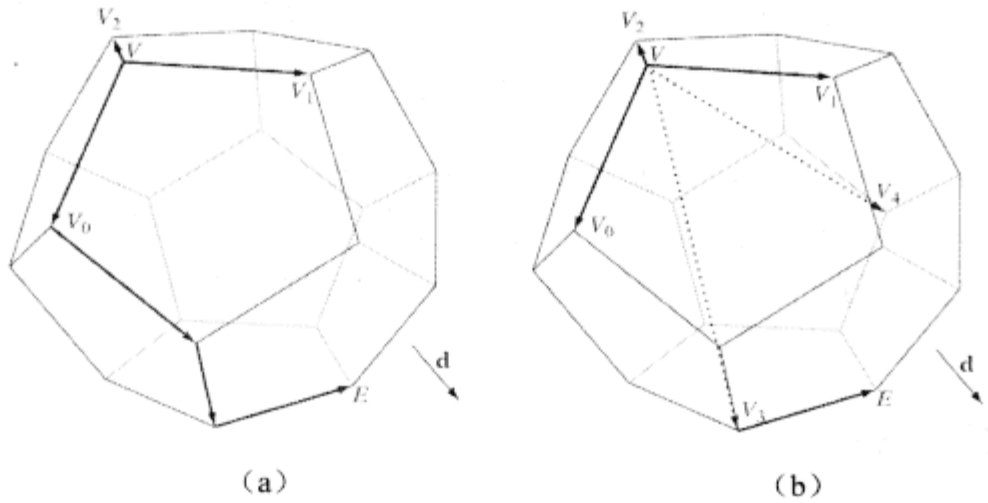


图 9.12 （a）仅采用邻接顶点时，从顶点 V 至极值顶点 E 的爬山算法。
（b）采用辅助信息（添加邻接数据）后的加速爬山算法

当爬山法引入邻接顶点时，还将衍生出另一个问题，即共面问题。图 9.13 显示了共面问题（此处，顶点位于同一条边上）所导致的爬山法局部最小值阻塞问题。在图 9.13（a）中，假设顶点 A 具有局部极小值，而爬山法无论是移向顶点 B 还是达到 C，都无法靠近极值点 E。而若采取随机路径移至两个邻接点中的某一个，则在下一步操作中可能还会回溯至顶点 A，从而造成死循环。在 2D 环境下，禁止对先前已访问顶点的回溯将消除这一类死循环操作，但这在 3D 环境下并不奏效（顶点可能不包含可访问的邻接顶点）。

通过将各顶点分别添加 N 个邻接顶点，可在爬山法搜索过程中有效地避免局部最小值阻塞问题，其中，顶点 N 即为顶点 V 的非共面邻接点。如图 9.13（b）所示，为顶点 A 添加顶点 D 将有助于避免局部最小值问题。另外，需要补充的一点是，也可使用层次结构在算法时间内确定某一极值点，例如 Dobkin-Kirkpatrick 层次结构。

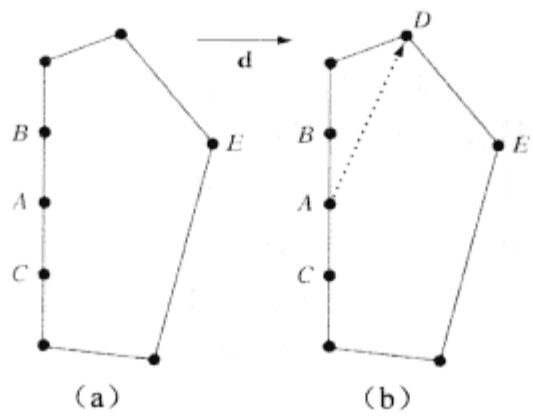


图 9.13 （a）对顶点 A 的邻接顶点 B 或 C 应用爬山法并不能接近极值点 E，因而顶点 A 具有局部极小值。
（b）添加非共面邻接顶点（这里为顶点 D）可有效地避免局部最小值阻塞问题

9.5.5 与顶点缓存相关的一致性问题

爬山法求解极值点在很大程度上将依赖于初始顶点 P 的选择。在最佳情况下，顶点 P 自身即是极

值点，可使搜索过程即刻结束；而在最坏情况下，顶点 P 可能位于极值点的相反方向。另外，由于对象在帧间的相对位移量一般都较小，因此，将上一帧中的、对象间的某一最近点作为初始顶点 P 可视为一种较好的解决方案。

另一种方法则是保留上一帧中的单形体数据。无论采用哪一种方案，都可以使用数组或哈希表加载帧间的各对象组信息。当无法保证帧间一致性特征时，可继而计算对象质心坐标间的对应向量，并根据该向量计算两对象上的支撑点。最终，可选择这一类支撑点作为初始顶点。该方法也可用于首次初始化操作中。

9.5.6 旋转对象的优化

两对象间的碰撞通常都发生在同一参考坐标系内，例如，场景空间所表示的世界坐标系，或某一相对坐标系（后者则更为简洁，因为只需转换一个对象）。然而，GJK 算法前期并不将对象的全部顶点转换至另一个对象空间内，较好的转换方案是仅将传递至支撑映射函数的方向向量转换至另一参考坐标系中。对于前者，对象的全部 n 个顶点都必须执行一次转换（适用于包含较少顶点的对象）；对于后者，则仅需转换某一向量，但向量须转换 k 次——每次循环都将执行一次转换。一般情况下，k 值远小于 n，因而后一种转换方案将更加高效。另外，前者还需要额外的内存空间以存储转换后的顶点数据，而后者无须存储这一类附加顶点数据。但在上述两种转换方式中，待获取最近点后，仍需两步额外的转换操作以便在世界空间内表示这些顶点。

针对第二种转换方案，它并不适用于下列情形：利用二分搜索计算两个对象的碰撞时刻 t，即此时经过线性变换的一个移动对象与另一个对象相交。第一种转换方案仍需要 n 次转换（其中，平移转换并不改变对象的旋转状态），而第二种转换方案则需要 km 次转换，其中 m 为二分搜索的步数。

9.5.7 移动对象的 GJK 算法

虽然 GJK 算法经常出现于两个凸多面体间的计算场合，但这远不能体现其通用性。例如，给定两个顶点集，算法可在点集凸体间计算最小距离向量（例如，支撑映射函数从不返回凸体内部的顶点）。这一显著特征使得算法可以一种直接方式计算线性平移凸体对象间的碰撞检测。

[Xavier97]中给出了一种计算移动多面体的解决方法。考查两个多面体 P 和 Q，其位移向量分别表示为 $\mathbf{t}_1$ 和 $\mathbf{t}_2$（这里为了简化碰撞测试过程，将使多面体 Q 处于静止状态），因此，多面体 P 的相对位移向量（基于多面体 Q）定义为 $\mathbf{t}=\mathbf{t}_1-\mathbf{t}_2$，且有初始位置处的顶点 V_i，则 $V_i+\mathbf{t}$ 表示原顶点平移后的位置。

不难看出，在多面体 P 的全程移动范围内，P 所形成的扫掠体将对应于首、尾顶点 V_i 和 $V_i+\mathbf{t}$ 构成的凸体（如图 9.14 所示）。因而，处理平移多面体 P 与 Q 之间的碰撞问题则简化为向 GJK 算法传递 P 位移前后的首、尾顶点（其中，给定两个点集后，GJK 算法负责计算对应的凸体）。但该方案并非完美无缺：当多面体 P 的顶点数量加倍后，其支撑顶点的计算时间也会相应地增加。为了避免这一问题，可考查向量 $\mathbf{t}$、$\mathbf{d}$ 之间的位置关系。从极值点的相关定义不难看出，当 $\mathbf{t}$ 背离向量 $\mathbf{d}$ 时，顶点 $V_i+\mathbf{t}$ 中则不存在超越 V_i 的极值点。因此，当 $\mathbf{d}\cdot\mathbf{t}\leqslant 0$ 时，可确保在顶点 V_i 范围内计算极值点（如图 9.15（a）所示）。类似地，若 $\mathbf{d}\cdot\mathbf{t}>0$，则只需在 $V_i+\mathbf{t}$ 范围内考查极值点（如图 9.15（b）所示）。

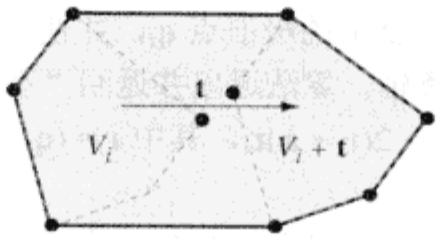


图 9.14　针对平移 **t** 后的凸多面体，起始顶点 V_i 和最终顶点 V_i+**t** 形成了该多面体的扫掠体

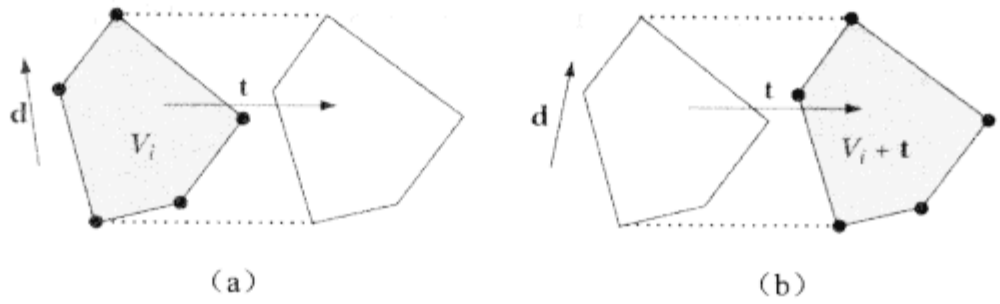


图 9.15　（a）若 $\mathbf{d} \cdot \mathbf{t} \leqslant 0$，则可在原顶点集 V_i 中计算出支撑顶点且无须测试 $V_i + \mathbf{t}$ 顶点集。（b）类似地，若 $\mathbf{d} \cdot \mathbf{t} > 0$，则只需考查 $V_i + \mathbf{t}$ 顶点集

通过修订支撑映射函数以在爬山法计算过程中对顶点添加位移向量，即可高效地实现上述（第二个）方案。需要注意的是，算法只是提供了对象间的冲突检测结果，而不包含其他详细碰撞数据。然而，可采用区间二分法这一有效方案计算对象间的碰撞时刻。当采取此方法时，上一次循环迭代中的单形体可作为下一次循环迭代中较为理想的候选数据。另外，也可以采用诸如 Brent 算法这一类迭代方案对碰撞时刻加以计算，具体内容可参见[Vlack01]。

[Bergen04]中也阐述了与碰撞时间相关的一种算法，该方法是一类基于 GJK 的迭代算法，可在光线与通用凸体对象间进行碰撞计算。通过构造对象的 Minkowski 和并对其实施光线投射，该算法也可直接用于移动对象间的碰撞测试，其中，光线对应着对象间的相对位移向量。因此，根据光线与 Minkowski 和之间的交点，可直接计算出碰撞时间。

9.6　Chung–Wang 分离向量算法

本章所讨论的最后一个凸体算法是 Chung、Wang 二人提出的分离向量算法。该算法源于多面体 P、Q 间的相交测试，快速并易于实现。实际上，Chung-Wang（CW）算法由两个算法构成：主算法最初考查一个候选分离向量，在 P、Q 分离状态下以迭代的方式更新该向量，并逐步逼近最终的分离向量；辅助算法负责检测 P、Q 间的相交状态。主算法总是处于无限循环中。

类似于 GJK 算法，CW 算法也将针对多面体的顶点进行操作。在每一次迭代过程 i 中，主算法相对于当前候选分离向量 $\mathbf{s}_i$ 计算 $\mathbf{p}_i$ 和 $\mathbf{q}_i$（$\mathbf{p}_i \in P, \mathbf{q}_i \in Q$）中的极值点。若该顶点组合的计算结果显示对象沿 s_i 方向处于分离状态，则算法退出并返回“不相交”这一信息；否则，通过计算关于直线 $\mathbf{p}_i$-$\mathbf{q}_i$ 垂线的、s_i 的反射向量，算法将得到一个新的候选分离向量。CW 算法的主算法包含下列 5 个步骤（相关计算符号可参见[Chung96]）：

1．算法始于某一候选分离向量 $\mathbf{s}_0$ 且令 $i = 0$。

2．分别计算 P 中的极值点 $\mathbf{p}_i$ 以及 Q 中的极值点 $\mathbf{q}_i$，并使 $\mathbf{p}_i \cdot \mathbf{s}_i$ 和 $\mathbf{q}_i \cdot -\mathbf{s}_i$ 具有最大值。

3．若 $\mathbf{p}_i \cdot \mathbf{s}_i < \mathbf{q}_i \cdot -\mathbf{s}_i$，则 $\mathbf{s}_i$ 为分离轴，算法退出并返回“不相交”这一信息。

4．否则，计算新分离向量 $\mathbf{s}_{i+1} = \mathbf{s}_i - 2(\mathbf{r}_i \cdot \mathbf{s}_i)\mathbf{r}_i$，其中 $\mathbf{r}_i = (\mathbf{q}_i - \mathbf{p}_i)/\| \mathbf{q}_i - \mathbf{p}_i \|$。

5．令 $i = i + 1$ 并返回步骤 2。

图 9.16 显示了主算法中的两个步骤。在图 9.16（a）中，s_0 为计算支撑点 $\mathbf{p}_0$、$\mathbf{q}_0$ 时的初始候选向量。其结果显示，s_0 并非是 P、Q 间的分离向量，继而可计算 $\mathbf{s}_0$ 相对于 $\mathbf{r}_0^{\perp}$ 的反射向量 $\mathbf{s}_1$。如图 9.16（b）所示，在算法的第二次迭代过程中，支撑点 $\mathbf{p}_1$、$\mathbf{q}_1$ 表明：$\mathbf{s}_1$ 即为分离向量，则算法结束并返回“两多面体不相交”这一信息。

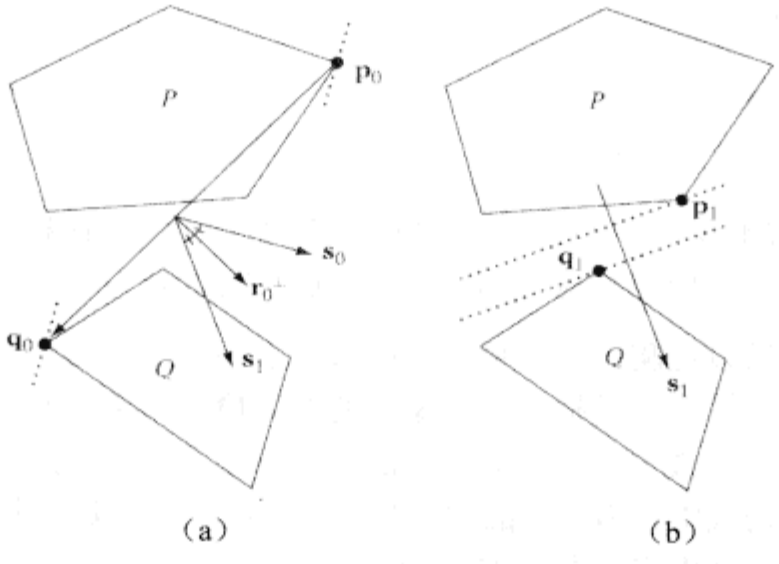


图 9.16　（a）多边形 P 和 Q 的 CW 算法的第一次迭代循环过程。（b）在第二次迭代循环过程中可得到分离向量

当多面体间相交时，子算法用于终止 CW 算法。算法试图获取向量 $\mathbf{m}$ 并满足 $\mathbf{m} \cdot \mathbf{r}_i \geqslant 0$，$0 \leqslant i \leqslant k$。若不存在该向量，则原点位于 P、Q 间的 Minkowski 差内，且多面体间不相交。算法所定义的时间复杂度为 $O(k)$，其中 k 表示迭代次数。这里，可在 n 次迭代间隔处执行子算法，从而降低算法的运行时间。

在上述步骤 4 中，更新相关向量的目的在于：对于两个球体而言，算法提供了步骤 1 中所示的分离向量。关于该规则的收敛证明，可参见[Chung96]。其中，若 $\mathbf{v}$ 表示为 P、Q 间的任意分离向量，则 $\mathbf{s}_{k+1} \cdot \mathbf{v} > \mathbf{s}_k \cdot \mathbf{v}$ 在每次迭代过程中将减小候选分离向量与最终分离向量之间的夹角，继而体现了算法的收敛特征。然而，[Bergen99]中指出了上述证明的缺陷之处，即只论证了 $\mathbf{s}_{k+1} \cdot \mathbf{v} \geqslant \mathbf{s}_k \cdot \mathbf{v}$ 这一情形，从而使 CW 算法的收敛性证明并不完善。即使如此，CW 算法在实际应用中也还工作良好。通过（迭代）记录两个对象间的分离向量，并将其作为后续测试中的初始候选向量，CW 算法的运行时间接近于常量级。同时，[Bergen99]还表明：CW 算法的平均计算速度将是 GJK 分离轴算法（增强版本）的 2 倍。[Chung96]中的相关实验数据也显示，当采用时间连贯性计算初始向量时，（95%以上）仅需 3 次迭代计算即可确定对象间处于分离状态。

9.7　小结

碰撞系统的测试范围一般仅限于凸体对象，相关的凸体特征使其非常适用于相交测试。特征之一：

分离凸体对象间的分割面的应用（或分离轴）。特征之二：凸体对象上两点间的距离既是局部最小值，也是全局最小值。因此可采用较为简单的爬山法计算最近点。

V-Clip 算法即使用了爬山法。该算法记录多面体间的一组特征，并通过遍历多面体的数据面以更新相应特征，从而不断缩小特征间的距离，直至获取一组最近特征。另外，可采用相应的层次结构以加速多面体上的最近点查询过程，例如 Dobkin-Kirkpatrick 层次结构。

多面体间的相交查询计算也可视为一类数学优化问题。例如，若仅确定最终的布尔值，则线性规划可用于确定两个（或多个）多面体间的相交状态。线性规划问题并不能确定两多面体间的最近点，基于此，可使用二次规划解决该问题。

针对特定的二次规划问题，GJK 算法则是计算多面体相交或（能够被支撑映射函数描述的）通用凸体对象时的有效方案之一。如前所述，GJK 算法还可用于处理线性平移对象。最后，针对处于分离状态的两个对象，CW 分离向量算法则可作为计算其分离轴时的一种高效的解决方案。

第 10 章　基于 GPU 的碰撞检测

当前，民用级图形处理器（GPU）已取得了长足的发展，与 CPU 相比，其固有的、强大的处理能力则更胜一筹。同时，在某些特定领域内，GPU 也起到了推波助澜的作用，例如，通过多个基于管线结构的、专用途的计算单元，可对渲染任务实现并行处理。虽然与 CPU 相比，GPU 的工作时钟速率较低，但这种并行机制仍使得 GPU 具有整体上的速度优势。鉴于当前 GPU 的发展趋势，例如顶点和像素着色器的可编程机制以及对浮点纹理的全面支持，GPU 强大的计算能力使其也能够实现某些原本在 CPU 上完成的常规任务（如映射某些非渲染任务、通用计算等）。针对碰撞检测，GPU 主要对应下列两类计算：基于图像空间的快速相交测试技术；将 GPU 视为一个协处理器，用以加速数学、几何计算。这里，基于图像空间的相交测试技术将依据颜色值、深度值以及模板缓冲区进行查询以正确地光栅化对象，从而确定对象间是否交叠。

图像空间相交测试技术通常易于实现，且可工作于任何可光栅化的几何图元之上（包括 Bezier 和 NURBS 面片），同时，不需要复杂的碰撞数据结构并可直接针对渲染几何体（以及可变形的非静态几何体）实施操作。由于测试过程基于渲染对象（多个）缓冲区的分辨率，因此，相应的碰撞检测只是一种近似计算。例如，当对象间处于分离状态时，算法有可能出现错误的“相交”返回信息。例如，具有较小间距的分离对象距渲染视平面较远时，便会产生这一类问题。同样，当对象间相交时，算法也可能会返回“分离”信息。例如，对象具有较小的尺寸或距离视平面很远时，将导致渲染对象中无像素可被渲染。尽量渲染近视平面对象（且未被剪裁）并选择具有最大屏幕投影的视见方向即可解决这一类问题。另外，若多边形的法线与视见方向垂直（多边形未被渲染），也会导致问题的出现，具体内容可参见后续章节。

通过将相关数据置入纹理中并使用可编程的像素着色器（以及混合操作），还可以在 GPU 中实现通用计算。例如，可通过多个纹理来刻画某一静态三角形网格，并对顶点的各分量值以及点-面连通关系进行编码。因此，可编写相应的着色器程序，并在光线与全部编码三角形之间实现并行相交测试。当前，GPU 通用计算技术是一个崭新且异常活跃的研究领域。例如，[Carr02]和[Purcell02]发布了利用 GPU 实现光线跟踪技术时的光线-三角形快速测试算法。

针对凸体或凹体对象间的相交状态，本章将探讨基于图像空间的碰撞检测技术。由于该领域（数学以及几何体的 GPU 加速计算）尚属起步阶段，对于碰撞检测问题其实用价值尚未可知。因此，本章并不深入探讨基于 GPU 的加速计算。

10.1　GPU 接口

将 GPU 作为计算设备并非其设计初衷，这也意味着需要解决某些实际问题。在 GPU 输入端，数据应以顶点流或纹理的方式实现包处理。包处理问题虽需谨慎处理，但总体而言仍不失为一种简单的

方法。这里，核心问题在于如何将存储于 GPU 中的计算结果返回至内存。在当前多数系统中，从 GPU 中回读数据将产生性能瓶颈并导致系统延迟，从而丧失了并行机制的优势。而在某些情况下，读取缓冲区又是不可避免的，通过巧妙地对算法加以组织，就可以消除这一瓶颈现象。例如，可采取硬件支持的遮挡查询方案。在必要情况下，也可对算法加以精心设计并限制相应的数据读取量。下面将详细讨论缓冲区读取机制，并最大限度地消除上述瓶颈问题。

10.1.1　缓冲区读取

依据某些算法，可将图像缓冲区中的数据读取至主存中，并对相关数据加以分析、判断以确定最终的碰撞结果（[Boyles99]和[Dinerstein02]）。例如，在 OpenGL 中可使用 **glReadPixels()**对缓冲区的某一矩形区域进行读取；对于某一特定的缓冲区，也可采用 **glReadBuffer()**。但是，从 GPU 内存缓冲区中读取数据则是一项十分缓慢的操作且难以令人满意。另外，读取操作须在管线其他操作完成后进行，并且在当前读取过程中，读取区域无法进行渲染操作。最终，缓冲区读取操作将导致渲染管线的延迟。除此之外，GPU 作为一类数据输出设备，对其发送数据时（而非读取数据）系统通常对总线传输操作进行了某种优化。最后，当数据返回至主存中，可能还需要对其进行修正并根据原来的读取要求实现格式匹配。如果缓冲区读取操作确实无法避免，则应尽量减少其读取次数以及相关的数据量。

某些算法使用了直方图查询以便从 GPU 中读取数据，例如，调用 OpenGL 扩展 EXT_histogram（[Boyles99]）。然而直方图扩展查询未必比读取缓冲区更快捷，在显卡驱动程序中，该查询操作通常也是以读取缓冲区的方式加以实现的，并运行于 CPU 之上（而非本地加速硬件上）。

若条件允许，可利用数据的填充率换取相应的传输带宽，从而减少缓冲区数据的读取量。例如，通过双线性过滤可将传输区域的采样降为原尺寸的 1/4。这里，假设读取缓冲区数据并检测黑色(0, 0, 0)背景中的某一白色(255, 255, 255)24 位像素。一次缩减采样后，像素值为(0, 0, 0)、(64, 64, 64)、(128, 128, 128)或(191,191,191)，假定取最近值。采用这种方式，执行 3 次缩减采样即可检测到原图像内白色像素（如图 10.1 所示）。4 次缩减采样后，仅需读取原图像的 1/256 至主存中，这极大地降低了传输时间。另外，也可以在数据读取至主存之前对缩减采样后的缓冲区内容实施累积操作（连接至同一邻接区域），从而降低缓冲区的读取次数。

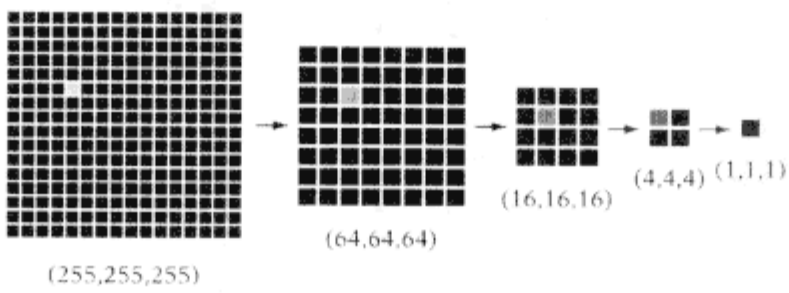


图 10.1　只需针对图像执行（最多）4 次双线性缩减采样，即可检测到黑色背景中的一个或多个白色像素。括号内的数字为对应像素的 RGB 值。其中，黑色像素的 RGB 值为(0,0,0)

10.1.2　遮挡查询

一类可完全避免读取 GPU 缓冲区的算法是使用遮挡查询，该特性为当前多数图形硬件所支持。当

开启此项功能时，遮挡查询将对已通过当前测试（如深度测试以及模板测试）的三角形（或其他渲染图元）中的像素（采样）数量进行计数。计数器的大小一般依赖于硬件的实现且不小于 1 位。另外，若计数器出现上溢，则其结果通常为最大饱和值。

在遮挡查询操作中，一般情况下将首先渲染场景中的主遮挡物，待其渲染完毕后，则针对每一个（复杂）细节对象 O_i，遮挡查询将对某一完全包含 O_i 的边界区域进行测试。若该边界区域不可见，则说明 O_i 已被遮挡，因而无须对其进行渲染并可执行剔除操作。在执行遮挡查询操作时，一般情况下需要关闭颜色缓冲区、深度缓冲区以及模板缓冲区的写入功能以节省处理时间，并确保查询过程不被外界干扰（如多次的连续查询操作）。

需要说明的是，遮挡查询功能出现于 OpenGL1.5 及其后续版本中（[Segal03]）。对于已通过模板测试以及深度测试的遮挡测试几何体，该操作将返回其采样数量。另外，通过降低遮挡查询等待时间所导致的时间延迟，可极大地提高 CPU 与 GPU 之间的并行操作。在 OpenGL 的早期版本中，遮挡查询测试则依赖于厂家提供的特定扩展操作，如 **HP_occlusion_test** 以及 **NV_occlusion_query**。

除了对象间的可见问题，遮挡查询还可有效地确定两凸体对象间的相交状态，具体内容可参见 10.2 节。

10.2 凸体对象间的测试

针对基于 GPU 的凸体对象碰撞检测，[Myszkowski95]和[Baciu99]很早便提出了相关的处理方案，其核心思想是将各渲染缓冲区内的像素视为一束垂直于视平面的光线，并向各对象进行投射。当光线与某一对象相交，其相交状态可记为光线首次以及最后一次相交所形成的区间。例如，针对两个凸体对象，沿光线方向其可能的相交状态包含 9 种情形，如图 10.2 所示。若针对某一像素出现 5～8 中的任一情形，则对象间处于相交状态，否则对象间处于分离状态。

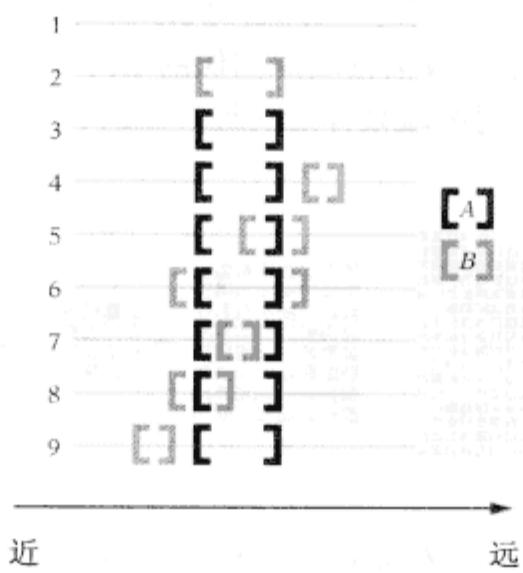


图 10.2　光线与凸体对象 A 和 B 相交的 9 种状态

这里，前期计算可通过读取模板缓冲区中的相关内容以执行测试。需要注意的是，应尽量避免缓冲区的读取操作，因而测试过程可在 GPU 上进行并采用遮挡查询。实际测试过程则非常简单——仅需执行两路测试，且首次测试后其提前退出（即对象间处于分离状态）的几率为 50%。

在首次测试过程中，对象 *A* 渲染至深度缓冲区中并采用“小于等于”测试条件。随后，深度测试条件更新为“大于”并关闭深度缓冲区的写入功能；渲染对象 *B*，同时开启遮挡查询功能。若遮挡查询操作返回“*B* 中像素不可见”这一信息，则对象 *B* 位于对象 *A* 的前方且对象间不相交。否则，执行第二次测试，测试过程同首次测试一样，只是 *A*、*B* 间的角色互换。在两次测试的开始处，深度缓冲区中的深度值皆初始化为远端 *Z* 值（1.0）。另外，在测试全过程中，禁用颜色缓冲区的写入功能。如果每次测试开始处的渲染操作只涉及对象的正面，且深度测试条件被设置为 GL_ALWAYS，在第二次测试时则无须初始化深度缓冲区。类似地，也可仅测试对象的背面以实现遮挡查询的优化操作。

图 10.3 显示了基于遮挡查询的碰撞检测过程。图 10.3（a）定义了对象 *A* 和 *B*，并将深度缓冲区初始化为远端 *Z* 值（如图灰色区域所示）。图 10.3（b）则显示了第一次测试过程，其中对象 *A* 被渲染至深度缓冲区内。其中，*B* 中多数像素片元通过了“大于”测试条件，因此需执行第二次测试。如图 10.3（c）所示，在第二次测试中，对象 *B* 首先被渲染至深度缓冲区中。当对对象 *A* 执行测试时，其像素片元皆未通过“大于”测试条件，因而其结论为：对象 *A*、*B* 间处于分离状态。需要注意的是，该方案不仅限于多面体间的测试，对于曲面类型，该测试同样有效。但对象必须为凸体，方可确保相交测试结果的正确性。针对基于遮挡查询的碰撞检测过程，其 OpenGL 实例如表 10.1 所示。

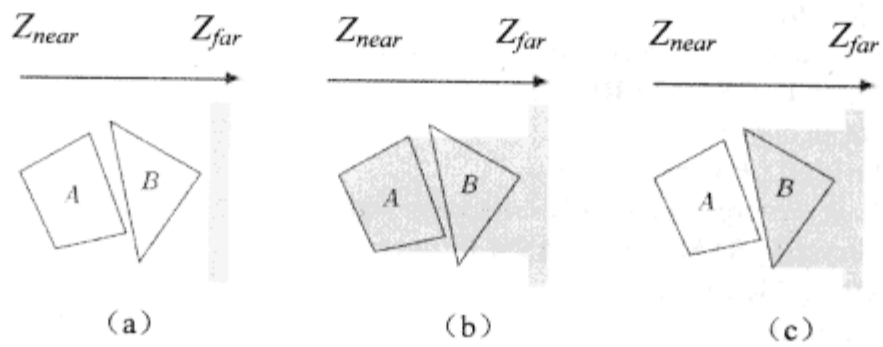


图 10.3　可采用遮挡查询方式判断（凸体）对象间的相交状态

表 10.1　采用 OpenGL 遮挡查询实现两个凸体对象间的碰撞测试

步　骤	代　码
0	*// Initialize depth buffer to far Z (1.0)* **glClearDepth(1.0f);** **glClear(GL_DEPTH_BUFFER_BIT);** *// Disable color buffer writes* **glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);** *// Enable depth testing* **glEnable(GL_DEPTH_TEST);** *// Initialize occlusion queries* **Gluint query[1], numSamplesRendered;** **glGenQueries(1, query);**
1	*// Set pixels to always write depth* **glDepthFunc(GL_ALWAYS);** **glDepthMask(GL_TRUE);** *// Draw front faces of object A* **glCullFace(GL_BACK);** **RenderObject(A);** *// Pass pixels if depth is greater than current depth value*

续表

步　骤	代　码
1	**glDepthFunc(GL_GREATER);** *// Disable depth buffer updates* **glDepthMask(GL_FALSE);** *// Render back faces of B with occlusion testing enabled* **glBeginQuery(GL_SAMPLES_PASSED, query[0]);** **glCullFace(GL_FRONT);** **RenderObject(b);** **glEndQuery(GL_SAMPLES_PASSED);** *// If occlusion test indicates no samples rendered, exit with no collision* **glGetQueryObjectuiv(query[0], GL_QUERY_RESULT, &numSamplesRendered);** **if (numSamplesRendered == 0) return NO_COLLISION;**
2	*// Set pixels to always write depth* **glDepthFunc(GL_ALWAYS);** **glDepthMask(GL_TRUE);** *// Draw front faces of object B* **glCullFace(GL_BACK);** **RenderObject(b);** *// Pass pixels if depth is greater than current depth value* **glDepthFunc(GL_GREATER);** *// Disable depth buffer updates* **glDepthMask(GL_FALSE);** *// Render back faces of A with occlusion testing enabled* **glBeginQuery(GL_SAMPLES_PASSED, query[0]);** **glCullFace(GL_FRONT);** **RenderObject(A);** **glEndQuery(GL_SAMPLES_PASSED);** *// If occlusion test indicates no pixels rendered, exit with no collision* **glGetQueryObjectuiv(query[0], GL_QUERY_RESULT, &numSamplesRendered);** **if (numSamplesRendered == 0) return NO_COLLISION;** *// Objects A and B must be intersecting* **return COLLISION;**

精确的碰撞结果不应只返回一个布尔值，当然，这也需要更加强大的硬件遮挡查询功能的支持。例如，除了通过相关测试的像素片元数，硬件不仅应返回基于屏幕空间的包围盒以及片元的最近和最远 Z 值，还应提供更加详细的信息，例如碰撞位置以及对象间的穿越程度。

需要注意的是，对于凹体对象 A、B 而言，上述测试则是一类保守算法。亦即，若测试结果返回 A、B 间不相交，则二者处于分离状态（位于光栅化近似精度范围内）；若测试结果返回凹体对象间相交，最终实际结果则并不确定。正确地处理凹体对象间的相交状态，其算法也不尽相同，详细内容可参见 10.3 节。

10.3　测试凹体对象

针对 10.2 节中的碰撞检测算法，为了避免凹体测试所返回的错误信息，[Knott03a，Knott03b]中提

供了正确的解决方案。回忆一下，当且仅当两对象 *A*、*B* 间（凸体或凹体）存在公共顶点时，二者处于相交状态。另外，若 *A*、*B* 为多面体，当且仅当对象边部分位于另一对象空间内时，二者处于相交状态。因此，若 *A*、*B* 间相交，则必存在边上一点共属于 *A*、*B*。同时，源于该顶点的投射光线（朝向观察者且为任意方向）将穿越对象 *A*（或 *B*）边界奇数次（光线离开对象要比进入对象的次数多一次），如图 10.4 所示。图 10.4（a）显示了对象 *A* 和 *B* 的透视图以及 *B* 中的各条边，图 10.4（b）则显示了对象间的侧视图，其中，分别从 *B* 中各边投射出 4 条光线并指向观察者。在 4 条光线中，存在一条光线穿越了对象 *A* 奇数次，因此对象间处于相交状态，为了确保获取正确的测试结果，还需选取对象 *B* 中全部边上的各个顶点并与对象 *A* 进行测试。另外，如果未检测到相交，还需再次选取对象 *A* 的各条边并与对象 *B* 进行测试。

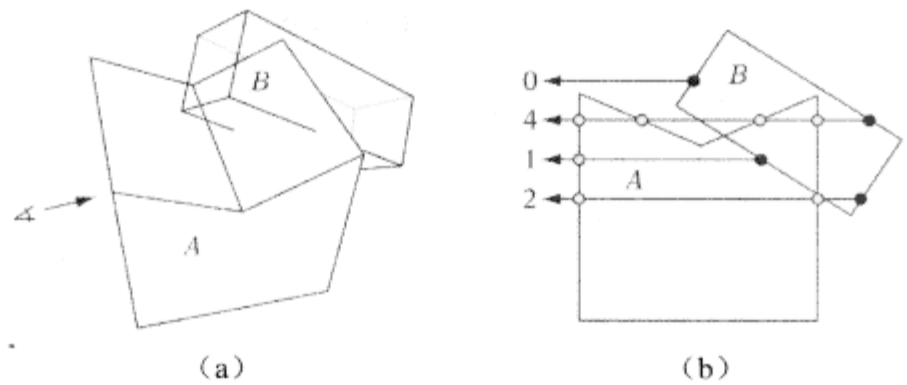


图 10.4　绘制 *B* 中各边并采用所讨论的相关算法与 *A* 中的各数据面进行测试，即通过 *B* 中各边并针对全部光栅化像素进行点-多面体之间的包含测试，以检测源于某一像素并朝向观察者方向的投射光线是否与对象相交奇数次

当依靠图形硬件时，单路光线相交测试可采用模板缓冲区测试以实现并行操作，相关步骤如下：

1．初始化操作。

☑　禁用颜色缓冲区的写入功能。

☑　初始化深度缓冲区和模板缓冲区。

☑　开启深度测试。

2．渲染各边。

☑　开启深度缓冲区的写入功能。

☑　将深度测试条件设置为 GL_ALWAYS，令全部顶点皆通过深度测试。

☑　绘制对象 *B* 中各边并将深度值写入深度缓冲区中。

3．渲染数据面。

☑　禁用深度缓冲区写入功能。

☑　将深度测试条件设置为 GL_LESS。

☑　开启模板测试功能。

☑　渲染对象 *A* 中的正面，对于通过模板测试的像素，相应模板值增加。

☑　渲染对象 *A* 中的背面，对于通过模板测试的像素，相应模板值减少。

待上述步骤执行完毕后，若 *A*、*B* 间处于相交状态，则模板中将包含多个非 0 值。表 10.2 显示了该碰撞检测的 OpenGL 实现过程。

这里不对模板缓冲区中的内容进行读取操作，针对两对象中较小对象的、基于屏幕空间的包围盒，可采用遮挡查询以检测模板缓冲区内的非 0 值。具体而言，通过传递非 0 模板值的查询像素，当且仅当遮挡查询返回“像素通过模板测试”这一信息时，对象间处于相交状态。

需要注意的是，上述算法并非完美无缺。例如，考查图 10.5 中的情形，其中轴对齐包围盒 *A*、*B* 处于相交状态。由于 *A* 中与 *B* 相交的各边正交于屏幕，因此无法实现其整体的光栅化。此处，*A* 所写入的深度值则由位于其最前方或最后方的数据面中的各边提供。不难看出，这一类边并不与 *B* 相交；同时，*B* 中各边也未与 *A* 相交。最终，测试过程将无法检测到二者间的相交状态。实际应用中，常采用透视渲染而非正交投影渲染对象，因此上述现象一般不会发生。即使对象间出现图 10.5（a）所示的构造方式，在透视投影中，其正交投影（如图 10.5（b）所示）出现的几率也并不高。另外，也可选择不同的视点进行碰撞测试，以进一步降低上述问题的出现几率。

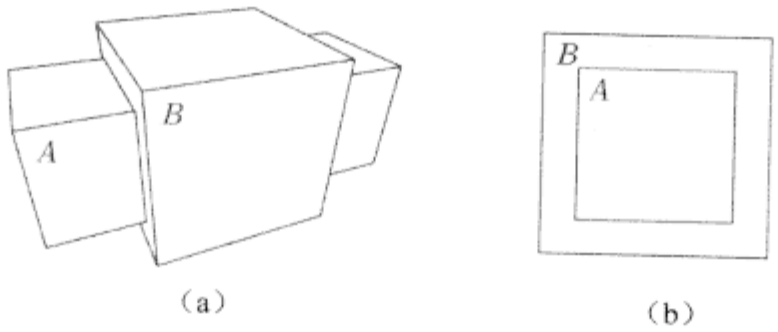


图 10.5　当采用正交投影时，针对此 AABB，算法将无法进行有效的相交测试。（a）侧视图。（b）正视图

这里所讨论的测试方案并不仅限于两个对象间的碰撞检测。对算法稍作改变即可实现多对象间的碰撞测试：首先以线框模式渲染全部对象，再以实体模式渲染相关对象。

表 10.2　采用 OpenGL 实现凹体对象间的碰撞测试

步　骤	代　码
0	*// Disable color buffer writes* **glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);** *// Clear depth and stencil buffers, initializing depth buffer to far Z (1.0)* **glClearDepth(1.0f);** **glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);** *// Enable depth buffer testing* **glEnable(GL_DEPTH_TEST);**
1	*// Enable depth buffer updates. Set all pixels to pass always* **glDepthMask(GL_TRUE);** **glDepthFunc(GL_ALWAYS);** *// Draw edges of object B* **glPushAttrib(GL_POLYGON_BIT);** **glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);** **DrawObject(b);** **glPopAttrib();**
2	*// Disable depth buffer updates. Pass pixels if nearer than stored depth value* **glDepthMask(GL_FALSE);** **glDepthFunc(GL_LESS);** *// Increment stencil buffer for object A frontfaces*

续表

步　骤	代　码
2	**glEnable(GL_STENCIL_TEST);** **glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);** **glCullFace(GL_BACK);** **DrawObject(A);** *// Decrement stencil buffer for object A backfaces* **glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);** **glCullFace(GL_FRONT);** **DrawObject(A);** *// Read back stencil buffer. Nonzero stencil values implies collision* **...**

另外，为了准确地定位处于碰撞状态的两个对象，可在线框渲染模式下以及实体渲染模式下，将对象（边和数据面）ID 编码至颜色缓冲区的对应位中。在非 0 模板值标识的某一对象的碰撞位置处，颜色缓冲区中的对应位即可定位该碰撞对象（边和数据面）。同时，考虑到硬件的光栅机制，直线的光栅操作一般不同于多边形。当一次测试多个对象时，在对象渲染过程中，各边须偏移某一量值以避免对象出现自相交现象。需要注意的是，该方法仅处理近距离对象间的测试，且渲染时并未对相关对象进行剪裁操作。除此之外，相应的剪裁操作将会极大地提高对象的渲染效率。

这里所采用的光线相交奇偶测试与模板阴影体计算（[Everitt02]和[Lengyel02]）、CSG 渲染（[Rossignac92]、[Stewart02]和[Guha03]）十分相像。而在模板阴影体计算中，模板缓冲区中的内容并不采用这里提供的计算方式。例如，模板操作将针对颜色缓冲区中的任一 RGBA 通道实现加、减混合操作（[Röttger02]）。

如果凹体对象定义为 CSG 对象，则基于 GPU 的 CSG 渲染技术也可用于这一类对象间的碰撞检测。当两棵 CSG 树交叠部分中的像素被渲染时，对象间处于相交状态。

10.4　基于 GPU 的碰撞过滤

10.2 节中的方案除了可用于组合测试以外，在此基础上对其稍加改动还可实现一类简单的碰撞过滤算法（[Govindaraju03]）。回忆一下，与第一个对象的深度值相比，若第二个对象中的、光栅操作后的全部像素片元更接近观察者，则两对象间处于分离状态。此时，相对于第一个对象，第二个对象完全可见。考查下列情形，首先对渲染多个对象并保留其深度值，随后对最后一个对象进行测试：当前计算过程可视为多对象间的测试。若当前对象完全可见，可知该对象不与任何前期对象相交，反之则不成立。亦即，处于非相交状态的某一对象相对于其他对象则不必完全可见。因此，当进行多对象间的相交测试时，该算法即是一类保守方案。

给定一个对象集，通过多次测试以查看该对象是否相对于其他对象完全可见即可实现上述碰撞过滤算法。若该对象完全可见，则其不与任何对象相交并执行剔除操作。该过程重复执行，直至无对象可被剔除。当测试过程结束时，结果对象集（潜在碰撞集，PCS）将涵盖可能处于相交状态的对象。如果必要，则可进一步采用组合测试以精确判断集合中处于相交状态的对象。

这里，令 $S=\{O_1,O_2,\cdots,O_n\}$ 为包含 n 个对象的对象集，相对于其他对象 $O_1, O_2, \cdots, O_{i-1}, O_{i+1},\cdots,O_n$，对象 O_i 的完全可见性测试可一次性完成，其中，可采取简单的两路算法以便在渲染对象的同时对其执行完全可见测试。首次计算时，对象间按照 $O_1, O_2,\cdots, O_n$ 的顺序逐一渲染、测试，并记录各对象的完全可见状态；在执行第二次（路）计算时，则按照 O_n , $O_{n-1},\cdots, O_1$ 的顺序渲染、测试对象，再次记录各对象的完全可见状态。当前，针对对象 O_i，若 O_i 在首次测试时相对于 $O_1, O_2,\cdots, O_{i-1}$ 完全可见，并且在第二次（路）测试时相对于 $O_n, O_{n-1},\cdots, O_{i+1}$ 完全可见，则对象 O_i 对于其他对象完全可见并处于分离状态。因此，在两路测试中标记为“完全可见”的任一对象都可从 PCS 中除去。另外，若最后 r 个对象在首次测试中（或最后 s 个对象在第二次（路）测试中）标记为“完全可见”，则相关对象皆可从 PVS 中除去。另外，关于对象 O_i 的各路“完全可见”测试，可采用遮挡查询实现其测试过程，如下所示：

1. 禁用深度缓冲区的写入功能。
2. 开启深度测试，设置深度测试条件为“大于”。
3. 针对对象 O_i 执行遮挡查询。
4. 开启深度缓冲区的写入功能。
5. 设置深度测试条件为“小于”或“等于”。
6. 渲染对象 O_i。
7. 考查相应的针对查询结果。若无像素通过测试，则在该路测试中标记对象 O_i 为“完全可见”。

上述步骤将针对全部 n 个对象重复执行，且深度缓冲区初始化为 1.0。

若场景中包含 n 个对象且按深度值实现 n 层栈式排列，则两路过滤算法仅能将最前端对象标记为“完全可见”，PCS 中剩余 $n-1$ 个对象。因此，将全部对象标记为“完全可见”将需要 n 次循环且 PCS 最终为空集。算法的整体性能将与场景的深度复杂度（depth complexity）呈线性相关。由于场景中各方向上的深度复杂度分布不均，对于较高的深度复杂度所导致的种种问题，可采用多视角对各对象实现多路渲染。通常情况下，可以沿 3 个主坐标轴方向分别对相关对象进行渲染。另外，较高的深度复杂度通常也不会出现在全部轴向上，进而可有效地解决这一类问题。

针对对象集 $S=\{A,B,\cdots,H\}$，图 10.6 显示了算法的实现过程。首先，算法将按照 $A\sim H$ 的顺序渲染对象，且对象 A、B、C、D、G 和 H 标记为“完全可见”（如图 10.6（a）所示）。其次，对象将按 $H\sim A$ 的顺序加以渲染，且对象 H、G、F 和 B 标记为“完全可见”（如图 10.6（b）所示）。待算法的第一轮计算（两路渲染）完毕后，可知对象 B、G 和 H 皆被标记为“完全可见”。因此，这 3 个对象可从 PCS 中剔除（如图 10.6（c）所示）。若还需要对过滤算法执行第二轮计算，待两路渲染完毕后，对象 C 将被剔除。至此，在当前观察轴向上，对象集 $\{A,D,E,F\}$ 为最终结果。若在水平轴向上进行渲染，则还可剔除对象 E。最终，PCS 中将保留对象 A、D 和 F（潜在碰撞集）。

碰撞过滤算法可直接应用于凸体或凹体对象之上。另外，如果对象形状比较复杂，在对该对象进行过滤操作之前，可根据其包围盒实现先期过滤。最后，该测试并不仅限于封闭几何体，当对象被确认为潜在碰撞时，也可将相关对象的数据面视为独立对象。因此，可在面集上实施上述过滤算法，并在算法结束时返回对应的潜在碰撞面集。在获取了相应的潜在碰撞面集后，如有必要，则可在 CPU 上计算最终的交点。

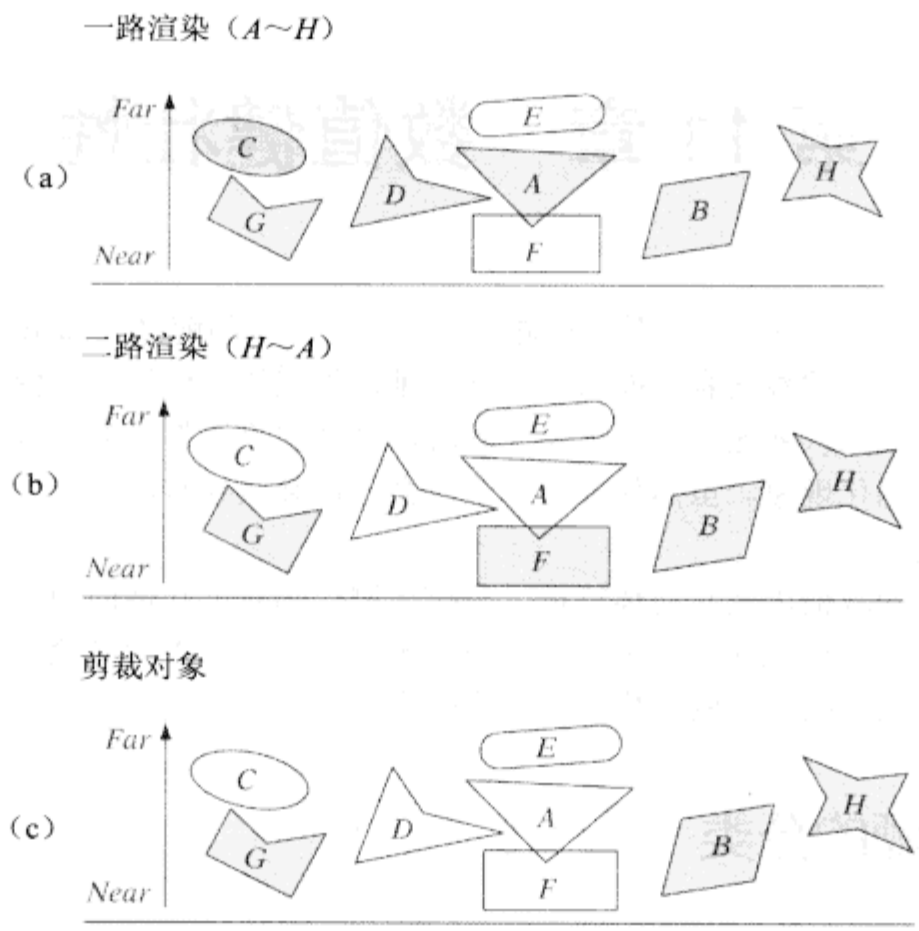


图 10.6　灰色对象视为完全可见。（a）一路渲染。（b）二路渲染。（c）两路渲染完毕后，对象 *B*、*G* 和 *H* 完全可见且从 PCS 中剔除

10.5　小结

针对图像空间的相交测试技术，本章探讨了基于 GPU 强大处理功能的对应解决方案。需要注意的是，应严格限制 GPU 与 CPU 之间的通信，以有效地避免缓冲区读取所导致的系统延迟问题。这里，可通过硬件支持的遮挡查询功能将相关数据反馈至 CPU，从而克服上述瓶颈现象，本章所阐述的相关算法也遵循了这一思想。另一种减少缓冲区读取量的相关方案是：通过双线性过滤法以降低数据的采样（这一过程可能需要重复执行）。

本章还探讨了凸体、凹体对象间的、基于图像空间的相交测试方案，并强调了该方案所导致的精确度问题。本章最后还分析了与最小潜在碰撞对象集相关的过滤算法，该算法可用于对象包围体的粗略测试，也可作为一种针对对象自身的精确过滤算法。

第 11 章　数值健壮性

到目前为止，与算法健壮性相关的某些问题仍未得到进一步的解释。如第 2 章所述，本书所涉及的健壮性问题将只限于程序的数值容错能力以及某些难以处理的几何结构。健壮的程序不会因数值误差问题导致其出现崩溃或陷入死循环这一现象。同时，健壮的算法将对多个等效输入数据返回一致的结果，并可正确地处理退化现象。虽然健壮的算法也会出现某些异常，但其返回结果仍会正确地反映算法当前所处的全局状态。

考虑到算法所涉及的数值以及几何体特征，碰撞检测代码对健壮性问题的要求将更加苛刻。特别地，实体对象间不允许出现彼此穿越的现象；一般情况下，对象也不应穿透地面。本章将从数值基础知识讲起，并详细讨论减少或完全避免数值误差的相关方法。

11.1　健壮性问题的分类

在碰撞检测过程中，与健壮性问题相关的内容主要包括以下两类：精确度误差以及退化现象。精确度问题源于计算机上的数值计算，并非采取严格的数学实运算，而是有限精度的浮点型运算或整型运算。退化现象一般是指：问题算法在执行过程中某些未被处理的特殊数据值或条件值。

通常情况下，算法假定在严格的数学实运算条件下进行，且实际操作过程中不会出现某些退化现象。但事实并非如此：首先，浮点型的计算结果并不精确，舍入误差即是一例；其次，退化现象经常出现于多种场合，有时也是相关设计要求所规定的（如对齐排列网格布局中的共线顶点）。

上述缺陷将导致相关计算结果缺乏一致性，最终得到错误的答案，严重时甚至使算法崩溃或陷入死循环。数值误差通常源于下列因素：

1．数值转换以及表示法导致的误差。通常情况下，这一类误差源于十进制数字未正确地转换为二进制数字，后者一般在定点或浮点运算中采用基数记数法。而无理数（如 $\sqrt{2}$ ）则无法以（整数）基数加以表示。

2．上溢和下溢错误。当数值极大且无法采用某一表示法加以显示时，常会出现上溢错误（相应地，当数值太小时也会产生下溢错误）。这一类错误往往出现在下列情形中：两个较大（或较小）的数字进行乘法操作时，或某一较大数字除以一个较小数字时（反之亦然）。

3．四舍五入形成的误差。考查实数乘法操作 $c = ab$，与 a 和 b 相比，结果 c 则需要更高的精度。若最终结果超出了表示法所允许的 n 位数字范围，则最后一位数字将产生误差。不仅是乘法操作，多数计算都将会出现这一类舍入误差。

4．抵消误差（cancellation error）。抵消误差往往出现在下列情形中：两个数值接近的变量进行减法操作；或一个较小的数值变量与一个较大的数值变量之间进行加、减法操作，即所谓的“大数吃小数”问题。

5．输入误差。误差也可能来自输入数据，包括建模误差、测量误差以及前次的计算误差。

另外，某些本不属于精度计算范围内的问题也应在有限的精度格式下加以适当的处理。例如，直线与三角形的交点位置最终可能会十分接近三角形的外部边界。类似地，线段的中点也有可能未处于该线段的支撑线上。

退化现象是指需要对特殊情况进行特殊处理，且通常采用“断言失败”这一方式加以标识。这里，断言定义为几何图元的查询结果返回某一枚举值。断言一般出现于错误的计算结果处，或调用代码未对其进行正确的处理。在几何算法中，断言常根据多项式的符号返回相应的结果。实际上，第 3 章中的相关示例即是这一类断言。这里考查下列断言：ORIENT2D(A, B, C)。该断言返回 2D 点 C 与有向直线 AB 之间的位置关系。当 A、B、C 共线时，任何一点的细小位置偏差都将导致其最终的返回结果截然不同。如果该断言采用浮点运算，则返回结果很可能难以令人满意。类似的与断言相关的健壮性问题可参见第 3 章。

许多算法（显式或隐式地）均采用断言的方式加以组织。若断言失败，则算法亦整体失效并返回错误的结果（或非正常结果）抑或陷入死循环状态中。处理退化现象将涉及修复相关算法以正确地处理问题断言，以及调整输入数据以消除错误实例，或采用扩展精度、任意精度实现断言表达式的准确计算。

为了全面理解并最终解决碰撞检测算法中的健壮性问题，深入理解有限精度计算的内在特征以及相关规则将是十分必要的。本章的后续部分将重点讲解浮点数的表示法，以及与健壮的浮点运算相关的计算方法。

11.2　实数表示法

虽然实数的数量是无穷的，但在计算机内部则须采用固定的尺寸对其加以表示。因此，相应的表示法无法涵盖全部实数集。对这一类无法直接表示的实数，可采用近似值的方式加以显示。其中，可精确表示的实数称作可示机器字（或简称为机器字）。

当论及计算机内部的实数表示法时，需要区分“精度”和“准确度”之间的差异。精度是指表示法中包含了多少位，准确度则定义为数值的正确性以及近似值中正确的位数。因此，抛开准确度而谈论精度是没有任何意义的。例如，3.123456789 具有较高的精确度，但相对于 π（3.14159…）而言却只是一个不太准确的粗略近似值。

计算机内部主要包括下列两种实数表示法：定点数和浮点数表示法。定点数则针对实数的整数位和小数位各自分配一组位数。例如，32 位定点数分别包含 16 位的整数位和小数位（即 16.16 格式的定点数），或包含 8 位的整数位以及 24 位的小数位（即 8.24 格式的定点数）。给定实数 r，则包含 n 位小数位的定点数可表示为 $2^n r$ 的整数部分。

例如，1/3 可表示为 16.16 格式的定点数 21846（$2^{16}/3$），或 8.24 格式的定点数 5592405（$2^{24}/3$）。其结果为，定点数均等分割数轴（如图 11.1 所示）。另外，定点数的表示范围与其小数精度之间存在一个折中方案：如果数值限定在 0 附近，则定点数可具有较高的小数精度（因此也具有较高的准确度）。

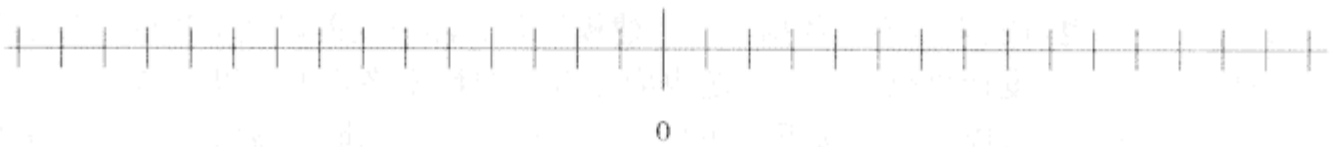


图 11.1　定点数在数轴上呈均匀分部状态

相比较而言，浮点数中小数点的位置则不固定，并可根据需要加以“浮动”。这里，可采取一种类似于科学计数法的标准格式实现小数点的浮动。在科学计数法中，十进制数字可记为 c（系数），其绝对值为 $1 \leqslant |c| < 10$ 且乘以 10 的（有符号）整数次幂。例如：

$$8174.12 = 8.17412 \cdot 10^3$$
$$-12.345 = -1.2345 \cdot 10^1$$
$$0.0000724 = 7.24 \cdot 10^{-5}$$

浮点数多采用二进制加以表示而非十进制。因此，（非 0）数值一般定义为下列格式：$\pm(1.f) * 2^e$。其中，f 对应于相应的小数位，e 为一个指数。例如，−8.75 可表示为 $-1.00011*2^3$。另外，$1.f$ 称作尾数或有效位，其中，f 项也称作分数。针对当前记数格式，数字 0 作为一个特例需加以区别对待（参见 11.2.1 节）。

格式化的浮点数具备下列优点：首先，数值间具有唯一表示方案，从而使记数方案不会出现混乱。否则，数字 100 将会出现多种表示方式：$10 \cdot 10^1$，$1 \cdot 10^2$ 或 $0.1 \cdot 10^3$。其次，诸如 0.00001 这样的数字，前面的第一个 0 无须显示，这也使得这一类较小的数字具有更高的精度。最后，小数部分之前的第一位永远保持为 1，因此无须对其加以存储，并可对尾数预留相对充足的位数。

与定点数不同，针对浮点数，由于全部（标准化）数字均采用相同数量的尾数位，因此数值密度随其指数的不同而不断变化。在数轴上，距离 0 越远的数字其间隔也就越大（如图 11.2 所示）。特别地，针对二进制浮点表示法，若指数分别为 $k+1$、k，其数字范围间隔前者为后者的 2 倍。因此，体现在数轴上则是 0 附近具有较高的浮点数密度。例如，对于 11.2.1 节将要阐述的 IEEE 标准，1.0～2.0 之间的数字量与 256.0～512.0 相仿；而 10.0～11.0 之间的数字量则要高出 10000.0～10001.0 许多（具体值分别为 1048575 和 1023）。

图 11.2　浮点数在数轴上呈非均匀分部状态：0 附近处的数值密度较大（除了近 0 处的非标准化的数值间隙）且距离 0 越远则越稀疏。另外，随着指数的逐次增加，数值间隔将成倍增长

与定点数相比，浮点数则具有更大的数值表示范围，同时，对于接近于 0 的较小值，浮点数也具有更高的精度。另外，与定点数相比，数字范围越大则越能体现浮点数的便捷性。然而，对于固定尺寸的指数，若浮点数超出了可表示的范围则称之为上溢。类似地，若数值过小以至于无法表示，则称之为下溢。由于指数部分通常保留固定的位数，在某些数值范围内浮点数的精度可能反而不如定点数。当前，除了某些掌上游戏机，所有的家庭计算机以及游戏机均支持浮点运算，这种快速的运算方式往往为整型和定点数所不及。

浮点数针对某些实数运算往往显示出不同的特征，尤其是数值未被正常初始化时，具体内容可参见后续小节。为了全面地掌握浮点运算，需要深刻理解相关表示法及其特征。目前，各类计算平台均

遵循 IEEE-754 标准，参见 11.2.1 节。

11.2.1　IEEE-754 浮点格式

IEEE-754 于 1985 年提出，是当前浮点运算的格式标准。该标准中定义了两种基本的二进制浮点格式：单精度浮点数和双精度浮点数。

在单精度浮点格式中，32 位数字包含下列 3 部分内容：一个符号位（S）、8 个指数位（E）以及 23 个小数位（F），如图 11.3 所示。这里并未存储小数部分前尾数位的首位：如前所述，该位通常为 1 且称作隐藏位或隐式位。另外，指数值也并未直接按原值加以存储。为了正确地表示正、负指数，往往在存储之前对其添加一个偏移量。对于单精度浮点数，该偏移量为+127。如前所述，指数 0（全 0）以及指数 255（全 1）通常为保留位，用以显示特殊值。因此，E 的有效范围记为$-126 \leqslant E \leqslant 127$。综上所述，对于采用 IEEE-754 标准的单精度浮点数 V，其值如下所示：

$$V=(-1)^S * (1.F) * 2^{E-127}$$

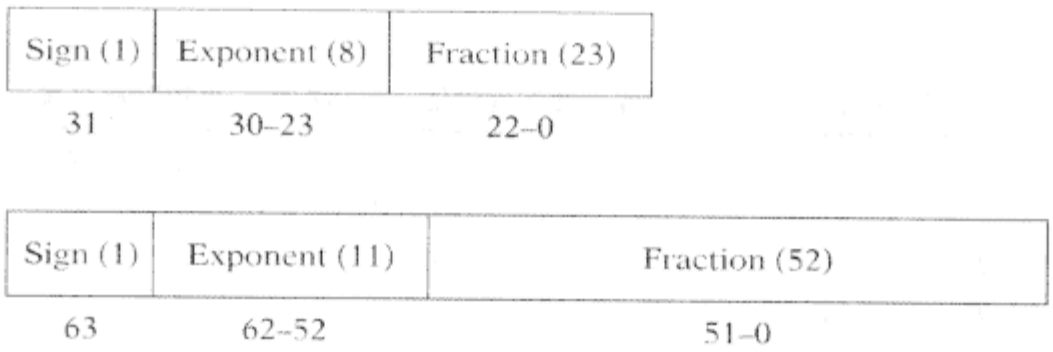


图 11.3　IEEE-754 单精度浮点格式（上图）以及双精度浮点格式（下图）

IEEE-754 标准规定双精度浮点数表示为 64 位格式。其中，包含了一个符号位、11 个指数位以及 52 个小数位（如图 11.3 所示），并且作用于指数上的偏移量为 1023。因此，对于采用 IEEE-754 标准的双精度浮点数 V，其值如下所示：

$$V=(-1)^S * (1.F) * 2^{E-1023}$$

IEEE-754 单精度浮点数包含了 8 个指数位，可表示的（绝对值）范围约为 10^{-38}～10^{38}。这里，24 位尾数意味着相应数值具有 6～9 位的小数精度。双精度浮点数可表示的（绝对值）范围约为 10^{-308}～10^{308}，且具有 15～17 位的小数精度。表 11.1 给出了一些单精度浮点数的表示形式。

表 11.1　IEEE-754 单精度浮点数符号位、指数以及尾数（二进制）的整数格式（十六进制）

数　值	位　模　式	符　号　位	指　数　位	尾数（二进制格式）
0.0f	0x00000000	0	0	(0.)000 0000 0000 0000 0000 0000
0.1f	0x3dcccccd	0	123	(1.)100 1100 1100 1100 1100 1101
0.33333333f	0x3eaaaaab	0	125	(1.)010 1010 1010 1010 1010 1011
1.0f	0x3f800000	0	127	(1.)000 0000 0000 0000 0000 0000
2.0f	0x40000000	0	128	(1.)000 0000 0000 0000 0000 0000
100.0f	0x42c80000	0	133	(1.)100 1000 0000 0000 0000 0000
1000000.0f	0x49742400	0	146	(1.)111 0100 0010 0100 0000 0000

续表

数　值	位 模 式	符 号 位	指 数 位	尾数（二进制格式）
−10.0f	0xc1200000	1	130	(1.)010 0000 0000 0000 0000 0000
机器 ε 值(2^{-23})	0x34000000	0	104	(1.)000 0000 0000 0000 0000 0000
最小正值(2^{-126})	0x00800000	0	1	(1.)000 0000 0000 0000 0000 0000

基于相应的移位操作，可比较两个正浮点数之间的位模式（正如比较两个 32 位整数那样），这与直接比较两个浮点数具有相同的效果。因此，正浮点数可按照对应的整数表示法加以存储。而针对负浮点数，对应的有符号整数将会产生下列数序：

$$-0.0 < -1.0 < -2.0 < \cdots < 0.0 < 1.0 < 2.0 < \cdots$$

可逆置负浮点数对应的整型表示范围（基于特定机器平台之上且将浮点数视为整数），随后，重新排列数序并逆向存储浮点值，即可解决上述问题。

需要注意的是，数值 0 不可按照 $V=(-1)^S * (1.F) * 2^{E-127}$ 加以表示。通常情况下，针对某些特殊值一般会设置相应的位串对其进行表示，例如 0。这一类特殊值的指数位往往为 0 或 255。表 11.2 为某些可能的特殊值。

表 11.2　IEEE-754 单精度浮点数的理论值，包括指数、小数以及符号

指　数	小　数	符　号	数　值
$0<E<255$			$V=(-1)^S * (1.F) * 2^{E-127}$
$E=0$	$F=0$	$S=0$	$V=0$
$E=0$	$F=0$	$S=1$	$V=-0$
$E=0$	$F\neq 0$		$V=(-1)^S * (0.F) * 2^{E-126}$（非标准化数值）
$E=255$	$F=0$	$S=0$	$V=+INF$
$E=255$	$F=0$	$S=1$	$V=-INF$
$E=255$	$F\neq 0$		$V=NaN$（若首个小数位为 0 则置位，否则为 NaN）

若仅采用标准化数值，数轴上 0 附近位置将会出现相应的数值间隙。IEEE 标准也包含了某些非规格化（异常）数字以填补这一间隙，它提供了相应的数值域以表示小于最小标准化数值的相关数字（如图 11.4 所示）。

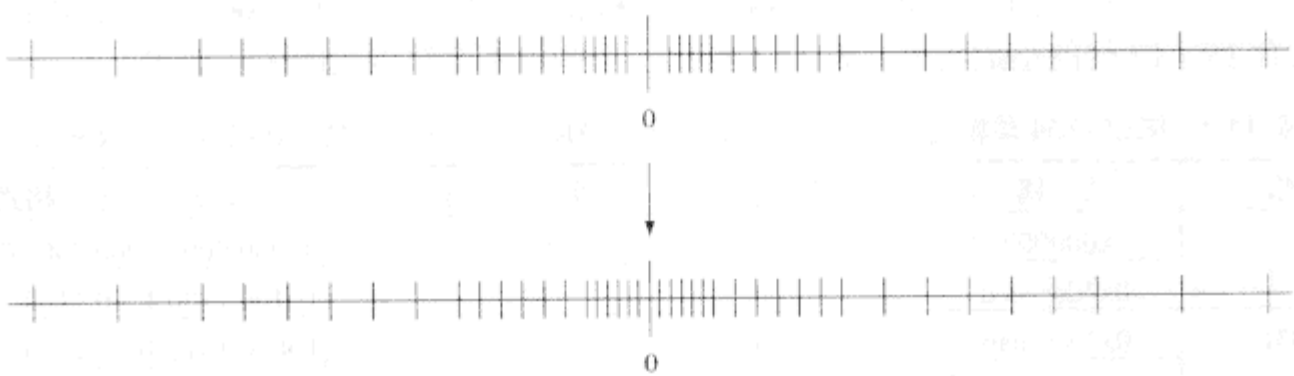


图 11.4　非规格化（异常）浮点数将填充 0 附近的数值间隙

IEEE 标准还定义了诸如正无穷大、负无穷大以及“非数值”这一类特殊值（分别表示为+*INF*、−*INF* 和 *NaN*），相关内容将在 11.2.2 节中加以讨论。另外，−0 也具有其特殊用途：除 0 操作后使得+*INF*

或-*INF* 生成正确的结果。该值与 0 比较时有-0 = +0。

IEEE 标准体现了数值的高精度特征，这既是引入该标准的初衷也是其被广泛接受的原因。简而言之，该标准要求相应计算结果的正确性达到位级。如果数值无法在机器上正常表示（即以 IEEE 格式直接显示），则需要根据该数值附近的可示机器数之一正确地设置舍入规则，从而实现相应的舍入操作（最终结果值稍有不同）。

为了确保舍入操作的正确执行，IEEE 标准在浮点计算过程中使用了 3 个辅助位——两个保护位和一个粘贴位。这里，两个保护位也充当精度位，使得尾数位达到 26 位而非 24 位。如果某一超出保护位的无效位变为非 0 值，则该粘贴位将被设置。同时，若保护位数量紧张时，采用粘贴位可确保实现正确的舍入操作。需要注意的是，这 3 个额外的辅助位只用于内部计算且不需存储。

IEEE 标准包含了 4 种舍入模式：最近舍入（舍入为最近可示数，即舍入至偶数）、负无穷舍入（向下舍入）、正无穷舍入（向上舍入）以及 0 舍入（即多余位的截取操作）。表 11.3 为 4 种舍入操作的相关示例。

表 11.3　IEEE-754 标准支持的 4 种十进制舍入模式

数　值	最 近 舍 入	负无穷舍入	正无穷舍入	0 舍入
−1.35	−1.4	−1.4	−1.3	−1.3
−1.25	−1.2	−1.3	−1.2	−1.2
1.25	1.2	1.2	1.3	1.2
1.35	1.4	1.3	1.4	1.3

舍入操作所引入的误差可通过一个 ε 值（单位舍入误差）加以限制。这里，ε 值表示加至 1 上的最小非 0 机器可示数且计算结果异于 1。特别地，若 $fl(x)$定义为实数 x 至其最近浮点数的映射函数，则有下列形式：

$$fl(x) = x(1+\varepsilon), |\varepsilon| \leqslant u$$

其中，u 即为机器所表示的 ε 值。对于单精度浮点数，该值为 $2^{-23} \approx 1.1921 \cdot 10^{-7}$；对于双精度浮点数，该值为 $2^{-52} \approx 2.2204 \cdot 10^{-16}$。在 C/C++语言中，上述值分别定义为 **FLT_EPSILON** 和 **DBL_EPSILON**，具体可参见 C 语言中的<float.h>文件以及 C++语言中的<climits>文件。其中，ε 值可用于浮点运算时的误差评估。

[Goldberg91]中详细阐释了浮点数的表示方法及其相关问题。在后续章节中，若不作特殊说明，相关浮点数皆定义为单精度浮点数。

11.2.2　无穷运算

IEEE-754 标准不仅可表示实数，还引入了诸如负无穷、正无穷以及 *NaN*（非数值，也称作不定值或不定数）这一类拟数（quasi-number）。这一扩展范围内的相关计算则称为无穷运算。无穷运算通常用于某些特殊情况的测试及处理（如除 0 操作），同时，这也意味着需谨慎处理各类相关细节以实现代码的健壮性。

正无穷和负无穷通常源于计算结果出现上溢，它们也可作为计算表达式的极限值。需要注意的是，当计算结果大于最大可示数值时便会产生上溢。例如，一个极大数除以一个极小数，如下所示：

```
float a = 1.0e20;
float b = 1.0e-20;
float c = a / b;          // gives c = +INF
float d = -a / b;         // gives d = -INF
```

对于形如 1/0 这样的计算表达式，通过选择较小的（正）分母值可使其结果无限大。当分母接近于 0 时，最终表达式值将趋于无穷大，因此可将其限定为 *INF*。类似地，选择较大的分母值将使相应结果无限趋于 0，因此 1/*INF* 结果为 0。在某些情况下（如 0/0），则不存在极限值，此时其结果为 *NaN*，如下所示：

```
float a = 0.0f;
float b = a / a;          // gives b = NaN (and _not_ 1.0f)
```

NaN 同时也是 $\sqrt{-1}$，$(INF - INF)$以及 $0 \cdot INF$ 这一类计算表达式的最终结果（实际上，不同的操作将返回各异的 *NaN* 代码。当然，这一差异不在当前讨论之列）。

IEEE-754 标准中还包含了 4 类互斥的比较操作：小于、等于、大于以及无序操作。对于任意数值对，上述操作都是唯一确定的，即使是 *INF* 和−*INF* 也不例外。例如，下列操作都为真：$INF = INF$、$-INF = -INF$、$-INF < INF$、$-INF < 0$ 以及 $0 < INF$。针对于 *NaN* 的操作则视为无序操作，亦即，除了 $NaN \neq NaN$，其他涉及 *NaN* 的比较操作都定义为假。而对于与 *NaN* 相关的运算操作，只要运算符两侧出现 *NaN*，其结果就为 *NaN*。

应谨慎处理 *NaN* 所导致的“三区间”问题。假设下列条件之一成立：$a < b$、$a = b$ 或 $a > b$。若 $a > b$，则有 $\neg(a \leqslant b)$，但这对 *NaN* 并不成立。当进行无穷运算操作时，下列表达式：

```
if (a > b) x; else y;
```

并不等价于：

```
if (a <= b) y; else x;
```

因此，下列范围测试将会遗漏 *NaN*：

```
// Test if val is in the range [min..max]
int NumberInRange(float val, float min, float max)
{
    if (val < min || val > max) return 0;
    // Here val assumed to be in [min..max] range, but could actually be NaN
    ...
    return 1;
}
```

上述测试较为安全的方式为：

```
// Test if val is in the range [min..max]
int NumberInRange(float val, float min, float max)
{
    if (val >= min && val <= max) {
        // Here val guaranteed to be in [min..max] range (and not be NaN)
```

```
        ...
        return 1;
    } else return 0;
}
```

为了有效地避免除 0 操作，考查 5.3.1 节中的线段 S 与平面 P 相交问题。这里令平面 P 定义为$(\mathbf{n} \cdot X) = d$，线段 S 的参数方程为 $S(t) = A + t(B - A)$，且 $0 \leqslant t \leqslant 1$，则线段与平面的相交 t 值为：

$$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A))$$

将 t 代入 $S(t)$方程中得到交点 Q：

$$Q = A + [(d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A))](B - A)$$

若线段平行于平面，则意味着分母为 0。根据 IEEE-754 标准所规定的运算法则，对于非 0 数 x，$x/0$ 的结果为 *INF* 或*–INF*（参照 x 的符号值）；若 x 等于 0，则结果为 *NaN*。综上所述，代码才可正确地返回“不相交”这一信息。当然，实际代码中可通过确认 t 值是否位于[0,1]范围内加以实现，如下所示：

```
// Test if segment AB intersects plane p. If so, return 1, along with
// the intersection t value and the intersection point Q. If not, return 0
int IntersectSegmentPlane(Point a, Point b, Plane p, float &t, Point &q)
{
    // Compute t value at which the directed line ab intersects the plane
    Vector ab = b - a;
    t = (p.d - Dot(p.n, a)) / Dot(p.n, ab);
    // If t in [0..1] compute and return intersection point
    if (t >= 0.0f && t <= 1.0f) {
        q = a + t * ab;
        return 1;
    }
    // Else t is +INF, -INF, NaN, or not in [0..1], so no intersection
    return 0;
}
```

依据 IEEE-754 标准，*INF* 以及 *NaN* 问题可通过“粘贴异常标志位”这一概念得到进一步完善。当出现异常时，该位将被设定直至实施清零操作。可首先执行某一快速但并非完美的计算方案，即不测试上溢、除 0 操作等异常。待计算完毕后，若粘贴位未被设定，则相关的计算结果有效；否则，需要调用慢速但相对健壮的代码版本。并非所有的编译器以及机器平台都遵循 IEEE 标准所规定的无穷运算法则，因此，针对与上述特征相关的实现方案，需仔细考查其操作的正确性。

11.2.3　浮点误差源

虽然几经修订，但浮点表示法及其运算的误差仍无法避免。首先，某些数字在特定的基数中无法实现准确的表示。例如，当采用基数 10 时，数值 1/3 则无法像数字 0.1 那样可以确切地表示其位数。然而，在二进制浮点表示法中，0.1 也无法实现准确的显示，即必须循环显示小数位$(0.0001100110011\cdots)_2$。当采取标准化表示方法并将该数值舍入至 24 位时（包括未加存储的一位），尾数位将止于…11001101，其中，最低有效位被舍入。因此，数值 0.1 在 IEEE 标准中的单精度表达值将大于其自身值。由于当前

CPU 均采用二进制浮点系统，因此下列代码很可能输出“Greater than”：

```
float tenth = 0.1f;
if (tenth * 10.0f > 1.0f)
        printf("Greater than\n");
else if (tenth * 10.0f < 1.0f)
        printf("Less than\n");
else if (tenth * 10.0f == 1.0f)
        printf("Equal\n");
else
        printf("Huh?\n");
```

在某些情况下，数值无法准确地表示还包括下列情形：例如，尽管利用 **x*0.5f** 替换 **x/2.0f** 可以得到准确的结果（2.0 和 0.5 皆可准确表示），而利用 **x*0.1f** 替换 **x/10.0f** 则无法实现最终的准确度。通常情况下，乘法操作要快于除法操作，因此，出于效率的考虑，采用 **x*0.1f** 替换 **x/10.0f** 这种计算方式仍然十分常见。

这里需要强调的一点是，浮点运算并不遵循一般的运算法则。例如，舍入误差造成的较小非 0 值将会作用于（加或减）某一较大值且不产生任何影响。因此，数学表达式可能会与最终的浮点运算结果大相径庭。

例如，考查表达式 **1.5e3 + 4.5e−6**。在实数运算中，该表达式对应于 1500.0 + 0.0000045，其最终结果为 1500.0000045。由于单精度浮点数最多包含 7 个小数位，因此相应结果截取为 1500.0 并出现位损失。因此，在浮点运算中，即使 **b** 为非 0 值且 **a** 和 **b** 皆可准确地加以表示，**a + b** 也有可能等于 **a**。

截断误差的出现使得浮点运算的最终结果将不再符合结合律，亦即，**(a + b) + c** 不等于 **a + (b + c)**。考查下列变量值 **a**、**b** 和 **c**：

```
float a = 9876543.0f;
float b = -9876547.0f;
float c = 3.45f;
```

将对应变量值代入上述表达式，其计算结果为：

```
(a + b) + c = (9876543.0f + -9876547.0f) + 3.45f = -4.0f + 3.45f = -0.55f
a + (b + c) = 9876543.0f + (-9876547.0f + 3.45f) = 9876543.0f + -9876544.0f = -1.0f
```

因此，编译器不会将表达式 **x + 2.0f + 0.5f** 转化为 **x + 2.5f**——二者截然不同。但编译器会针对左结合运算符（这也是多数运算符的计算方式）实现浮点常量的左结合操作。亦即，在编译期内可将 **2.0f + 0.5f+x** 转换为 **2.5f+x**。因此，一般将浮点常量置于变量的左侧以体现较好的编码规则（除了某些特定的操作）。

连续的舍入误差将会逐渐影响数值右侧的最低有效位，最终导致精度损失并产生较为严重的问题。更为糟糕的则是一类“数位消失”问题，该问题将导致数值丢失其左侧有效位且通常源于下列简单的计算：两个接近等值的子表达式进行减法操作（或数值相近而符号相反的两个数进行加法操作）。

考查下列示例：对于二次判别式 b^2-4ac，其中 $b=3.456$，$a=1.727$，$c=1.729$，其结果为 $4\cdot10^{-6}$。但采用单精度操作后，其最终结果则变为 $4.82387\cdot10^{-6}$，即只有第一位数字保持正确。这里，若消位问题所导致的误差问题十分明显，其计算结果往往是灾难性的。然而，也存在某些例外情况。例如，

考查减法操作 1.875 − 1.625 = 0.25，其二进制表示为 1.111 − 1.101 = 0.010。虽然在减法操作过程中出现了有效位丢失，但此时 3 个运算量均可准确表示，因而不会产生问题。在这种情况下，消位误差则是一类良性操作。

综上所述，在计算过程中需谨慎处理浮点运算。即使看似简单的两个表达式 $a(b-c)$和 $ab-ac$，其选择方案有时也需要多方斟酌。若 b、c 间数值接近，则表达式 $a(b-c)$将会产生消位问题；若 ab、ac 间数值接近，表达式 $ab-ac$ 同样会出现类似问题。当$|a|>1$ 时，ab 和 ac 之间的差值将会增大（相对于 b 和 c），因而可减少表达式 $ab-ac$ 消位问题的出现；相反，若$|a|<1$，则 ab 和 ac 项变小，因而将再次出现消位误差问题，但此时在表达式 $a(b-c)$中，该问题则得到了有效的控制。因此，最佳表达式的选取应视具体变量值而定（以及诸如上溢错误和下溢错误等其他因素）。

多数情况下，对表达式进行改写可有效地避免消位误差问题，例如：

$$f(x)=1-\cos(x)$$

当 x 接近于 0 时，上述函数将出现消位误差。这里，可将其改写为：

$$f(x)=2\sin^2(x/2) \quad 或\ f(x)=\frac{\sin^2(x)}{1+\cos x}$$

最终可有效地避免该问题。类似地，考查下列函数：

$$g(x)=\sqrt{x+1}-\sqrt{x}$$

当 x 较大时，同样会出现消位问题。可以按如下方式改写上述函数：

$$g(x)=\frac{1}{\sqrt{x+1}+\sqrt{x}}$$

[Acton96]中提供了多个有限精度计算中的误差消除实例。

11.3　健壮的浮点数用法

尽管存在着显著的缺陷，但对于实时应用程序而言（如碰撞检测），浮点数仍不失为一种较为方便的数值表示法。在对浮点数进行慎重处理后，通常可以实现其数值的健壮性。其中，一种较为重要的方法是使用误差容值，在后续小节中将着重考查这一内容，并详细讨论利用误差容值实现健壮的直线-平面、直线-三角形之间的相交测试。另外，还将阐述如何通过共享公共子表达式以确保相关计算间的数值一致性。该方案也将通过直线-三角形之间的相交测试加以演示。

11.3.1　浮点值的误差容值比较

当进行浮点运算时，考虑到舍入误差以及消位误差的累积效应，一般不建议在浮点值之间进行直接比较。例如，考查下列测试：

```
if (x == 0.0f) ...
```

该测试很可能无法达到预期的效果：x 上的累积误差将使该值十分接近于 0，但不会等于 0。一类较好的解决方案是使用误差容值，即当 x 位于以 0 为中心的某一较小范围内时视其为 0。如下所示：

```
if (Abs(x) <= epsilon) ...
```

其中，可选取 epsilon 作为误差容值，该值的选取范围可下至 10^{-6}，上至 0.1 或更大，一切皆取决于 x 的范围以及具体的计算过程。因此，x、y 间的比较操作则转换为下列格式：

```
if (Abs(x - y) <= epsilon) ...        // Absolute tolerance comparison
```

这里所采用的两个浮点数相等操作的 epsilon 误差容值也称作绝对容限，且该 epsilon 值固定不变。但在实际操作过程中，确定恰当的 epsilon 值并非易事，这也可视为绝对容限测试的一个缺陷。另外，epsilon 值还将依赖于输入数据的数值范围以及所采用的浮点格式。因此，仅确定某一 epsilon 值以满足全部浮点数值域是十分不现实的。对于较小 x 和 y 值（二者不等），其差值可能会小于既定的 epsilon 值；而对于较大值，相应差值又可能大于该 epsilon 值。一种处理方法是，当被测试的数值增大时，绝对容限测试所需的位数也就越多。但是，当相应结果值总大于既定 epsilon 值时，上述测试则会一直处于无效状态，除非二者完全相等（但原条件为 x、y 二者不等）。因此，绝对容限计算仅适用于数量级已知的情况下，并据此设置相应的误差容值。

与之相对的一种方案是相对容限，其核心思想是两个数字间进行除法操作，以考查最终结果与 1 的接近程度。这里，假设$|x|\leqslant|y|$，则相应测试变为：

```
if (Abs(x / y - 1.0f) <= epsilon) ...
```

上述表达式可改写为：

```
if (Abs((x - y) / y) <= epsilon) ...
```

为了避免开销较大的除法操作以及除 0 错，可将上式两边同时乘以 **Abs(y)**，如下所示：

```
if (Abs(x - y) <= epsilon * Abs(y)) ...
```

若不再考虑$|x|\leqslant|y|$这一假设条件，则表达式的最终结果为：

```
if (Abs(x - y) <= epsilon * Max(Abs(x), Abs(y))) ...        // Relative tolerance comparison
```

这里，针对相对比较测试，强调“小于等于”以及“不小于”测试条件是十分必要的。否则，当两个测试值皆为 0 时，该测试将处于无效状态。

需要注意的是，上述相对比较测试并非完美无缺。当 **Abs(x)**和 **Abs(y)**大于 1 时，测试表达式工作良好；但当其小于 1 时，epsilon 值也需适当减小，且测试数字越小其所需位数也就越多。这一问题也曾在绝对容限测试中出现，一种解决方案是对其实施组合，即当测试数值大于 1 时，采取相对容限测试；当其小于 1 时，则采取绝对容限测试。如下所示：

```
if (Abs(x - y) <= epsilon * Max(Abs(x), Abs(y), 1.0f)) ...        // Combined comparison
```

如果机器指令不支持 **Max()**，则上述表达式的计算开销将会相应地增加。因此，一种低开销的近似表达式如下所示：

```
if (Abs(x - y) <= epsilon * (Abs(x) + Abs(y) + 1.0f)) ...        // Combined comparison
```

由于相对容限测试较为复杂，因此，通常情况下会首选绝对容限测试。

通常，可对机器 epsilon 值放大某一小倍数以选取 epsilon 值。例如，可对机器 epsilon 值进行开方以确定最终的 epsilon 值，即相对于既定的机器表示法，只需约为 1/2 的数位量即可满足数值需求。

虽然一些计算几何文献并不赞同使用误差容值，并将其视为一种“epsilon 修补手段”或是“编程伎俩”，但在实时应用程序中且针对浮点运算的数值健壮性而言，误差容值确有其实际应用价值。另外，在碰撞检测过程中，误差容值的实际应用效果还是令人满意的。当采用较大的误差容值时，还可进一步消除误差累积这一现象。当然，这需要读者对所涉及的问题有着较为深刻的主观认知：没有任何 API 可供你调用，也没有任何函数可为你自动解决这一问题。同时，[Knuth97]也对误差容值问题进行了较好的阐释。

值得注意的是，在采用了误差容值处理后，“等值”这一概念将不再具有可传递性。亦即，若 $A = B$ 且 $B = C$，容值比较将无法保证 $A = C$。例如，在缺少了这种可传递性特征后，原来的排序操作可能会失效并与原排序结果截然不同。因此，健壮的程序代码必须考虑并处理这一类问题。

11.3.2　采用厚平面实现算法的健壮性

针对浮点数-实数间的显示误差问题，考查下列实例：计算线段 AB 和 CD 的交点，如图 11.5 所示，其中，各网格点表示可示机器值。由于交点 P 并未位于网格之上，则根据当前舍入规则，该顶点取为邻接网格顶点 Q。这里假设 CD 构造了一个 2D 几何场景边界，且 AB 为某一点对象的有向位移。而将交点舍入为 Q 则意味着将其移出当前几何场景，以致碰撞检测失效。

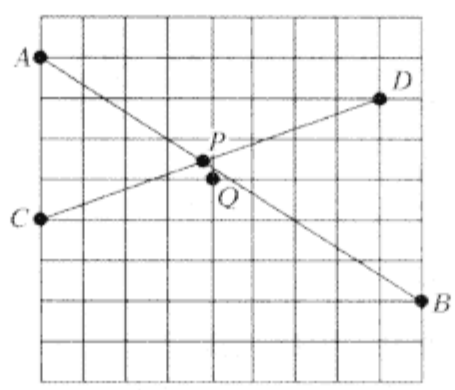


图 11.5　当使用浮点数时，线段 AB 和 CD 之间的交点 P 难以实现精确的表示。通常都会采用可示机器值附近处的某一近似值 Q

在 3D 环境下，类似的情形也存在于线段 AB 与平面 P 的相交测试过程中。基于浮点误差，最终的交点很可能既不位于线段 AB 上也不位于平面 P 上，而是位于二者真实交点附近处，如图 11.6（a）所示。其中，近半数交点位于当前平面的背面，因而点对象在由 A 至 B 的位移过程中很可能再次落入场景世界的外部。

针对交点位于平面外部这一问题，一种有效的解决方案是引入一个容值，即当前平面具有一定的厚度。这里，令交点偏离平面 P 的距离为 e，若平面的半径记为 r 且 $r > e$，则视交点位于当前厚平面之上，进而可有效地消除交点落入场景世界外部这一问题（如图 11.6（b）所示）。需要注意的是，e 值的确定（进而设定 r 值）并非易事，且需要根据与特定应用相关的诸多因素加以确定。实际的做法一般是根据经验对 r 进行计算：测试厚平面的交点并相应地增加 r 值，直至全部交点（针对全部输入数值域）正确地划分为厚平面上的点。

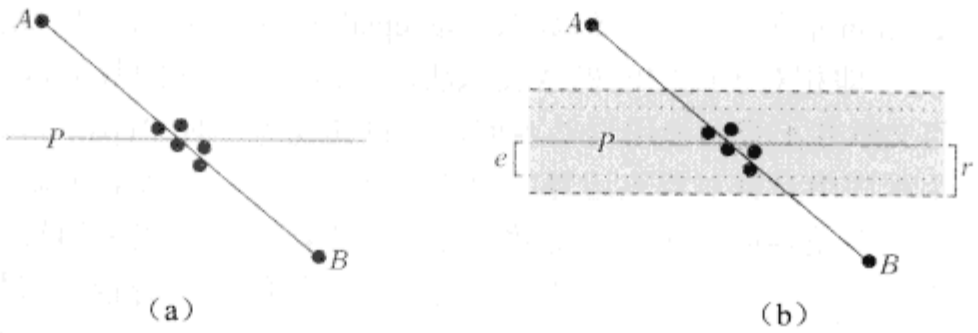


图 11.6 （a）若考虑到机器数的误差及舍入操作，则线段 AB 和平面 P 之间的交点很可能既不位于直线之上也不位于平面之上。相反，该交点有可能位于图中的各标记位置。（b）待采用半径为 r 的厚平面后，若 $r>e$，则可确保该交点位于当前平面之上。其中，e 值为交点偏离 P 的最大距离值

• 厚平面的引入还将有助于增加线段-平面间交点的精确度。如果线段近似垂直于当前平面，则线段位移量 d 与交点位移量 e 相似（如图 11.7（a）所示）；然而，随着线段与平面趋于平行，针对同一位移量 d，误差距离 e 将增长得更快（如图 11.7（b）所示）。这里，假设线段的长度值存在一个上限，厚平面的使用则反映了线段与平面之间的平行程度。当线段与平面间重合时，则无须计算二者之间的交点，因此，厚平面针对交点间的位移误差量设置了一个极限值。另外，厚平面还将有助于实现健壮的平面-多边形剪裁计算，具体内容可参见 8.3.3 节以及 8.3.4 节。

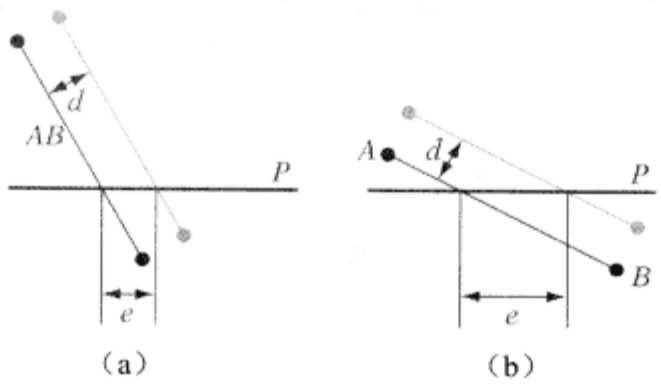


图 11.7 （a）令线段 AB 近似垂直于平面 P。当 AB 位移一较小距离 d 时，两交点之间的误差距离 e 也较小。（b）若线段与平面趋于平行时，相对于同一 d 值，则误差距离值 e 的增长幅度较大

11.3.3 采用共享计算实现算法的健壮性

考查下列直线（或线段）与三角形 T 之间的相交算法。测试首先在 L 与 T 的支撑面之间计算交点 P，随后，执行 P-T 间的包含计算。亦即，计算点 P 是否位于三角形各边（逆时针顺序）的同一侧（考虑到效率问题，可将 P 和 T 投影至包含最大 T 面积值的主平面上，进而可执行低开销的 2D 点-三角形包含计算——该步骤暂且忽略，因为它并不影响这里所讨论的健壮性问题）。

但是，当采取浮点运算实现上述方案时，测试过程将缺乏应有的健壮性。为了进一步说明该问题，考查直线 L 与三角形 ABC 和 ADB 之间的相交问题（如图 11.8 所示）。此处，π_1 和 π_2 分别定义为三角形 ABC 和 ADB 的支撑面。令直线 L 穿越边 AB，由于边 AB 为两个三角形的公共边，在实际运算中，L 可能与三角形 ABC 相交但不与三角形 ADB 相交。但是，浮点运算导致的误差将有可能产生下列情形：直线 L 不与任意三角形相交。其中的原因在于：首先，考查 L 与 π_1 之间的交点 P_1——浮点误差可能使

该点位于边 AB 的右侧，这将导致其位于三角形 ABC 的外部。该情形如图 11.9（a）所示。随后，直线 L 与平面 π_2 执行相交测试且交点为 P_2。由于 π_1 和 π_2 分属不同的平面且具有各自的平面方程，同时，方程中的数据项均受限于相应的浮点精度（误差），因此，交点 P_1 和 P_2 很难归为同一点。所以，也就不难理解 P_2 位于边 AB 的左侧且位于三角形 ADB 的外部（如图 11.9（b）所示）。最终结果为：直线与两三角形之间将呈现无效的相交测试。

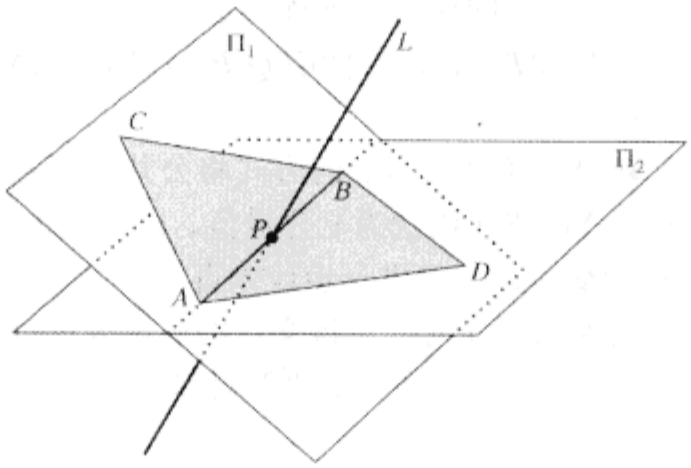


图 11.8　直线 L 与三角形 ABC 和 ADB 进行相交测试。由于 L 穿越两三角形的公共边 AB，因此易受到浮点误差的影响。若测试的健壮性较差，甚至无法正确地检测出与两个三角形之间的相交结果

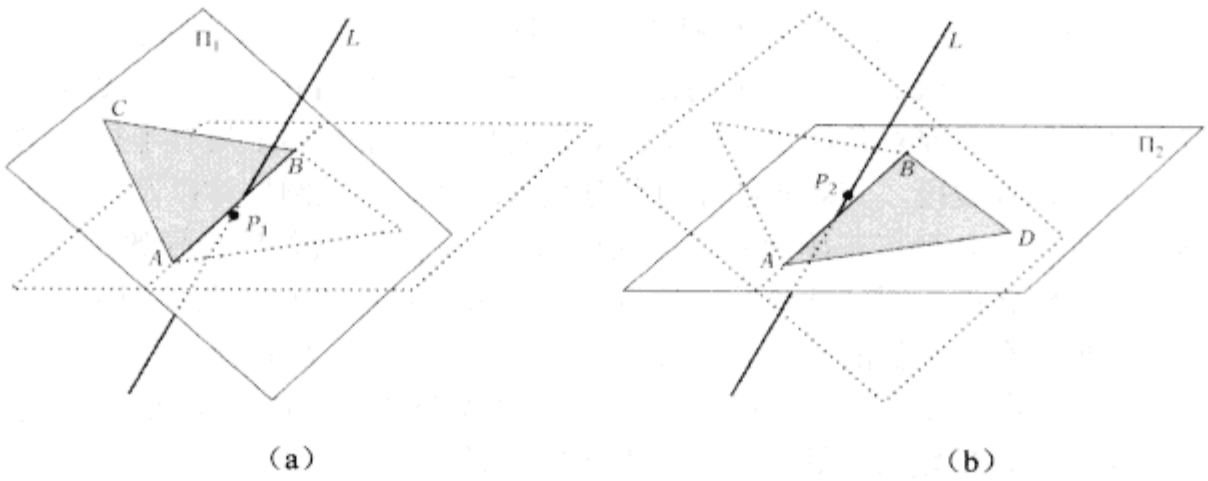


图 11.9　浮点误差可能会导致下列情况：（a）直线 L 和平面 π_1 之间的交点 P_1 位于三角形 ABC 的外部。（b）直线 L 和平面 π_2 之间的交点 P_2 位于三角形 ADB 的外部。因此，任何基于三角形平面的交点计算及其随后的点-三角形包含测试都难以满足健壮性的要求

测试失效主要源于以下两点：

1．数据丢失。在计算点 P_1、P_2 并将其表示为机器数时，存在精度损失。数据丢失使得三角形的包含测试缺乏可靠性。

2．在连续的相关测试之间缺乏信息共享机制。点 P_1 和 P_2 本应为同一个点，但由于采用了不同的交点计算方式，二者最终各异。因此，若在测试间采用具有某种相关一致性的判断条件（例如点 P 位于边 AB 的左侧抑或右侧）则不会出现上述情况。

一类较为健壮的直线-三角形相交测试算法曾在 5.3.4 节中进行过描述，该算法使用了标量三重积以确定直线 L 是否位于三角形各边内部。此处，假设直线 L 经过点 P 和 Q，标量三重积$[PQ\ PB\ PA]$的

符号决定了直线 L 与边 AB 的相对位置。

当标量三重积接近于 0 时，浮点误差有可能改变其符号值。但在 L 与 AB 间执行测试时，两个三角形将共用同一计算表达式，因而符号测试须具体指明 L 与哪一个三角形相交。亦即，共享计算确保直线 L 将于某一个三角形相交，即使该三角形并非目标三角形。此时，直线不再因浮点误差问题穿越多个三角形，测试的健壮性问题得到了改善。

需要注意的是，共享计算需在三角形间保持一致性。这也意味着，边 AB 与邻接三角形中的边 BA 不可同一而论。另外，针对标量三重积[PQ PB PA]和[PQ PA PB]，虽然实际操作中二者相等，但在浮点运算模式下，其结果则不尽相同。

针对索引顶点，可通过基于索引值排序边顶点的方式将公共边设置为同一有向线段；而对于非索引顶点，则可通过列选其较小的 x 分量值（依次为 y 分量值、z 分量值）实现顶点的排序。

仅当直线 L 与某一个三角形相交时，方可进一步计算直线与三角形支撑面的交点。这里需要注意的是，该交点有可能位于三角形的外部，任何针对该交点的后续处理过程都需要考虑到这一点。一种处理方案是抛弃直线或线段相交测试，并采用所谓的“厚对象”（或胖对象，fat object）进行测试。具体内容可参见 11.3.4 节。

11.3.4 厚对象的健壮性

如 11.3.3 节所述，直线或线段的相交测试通常都对浮点误差问题较为敏感，顶点测试也存在类似的情况。当涉及与顶点、直线、线段相关的健壮性问题时，一种解决方案则是避免使用这一类几何图元，转而执行前文所提到的厚对象测试。这里，如果对象在各个方向上均有半径 r，则称之为厚对象。因此，针对点、直线以及线段，其对应的厚对象分别为球体、无限圆柱体以及胶囊体。

如前所述，即使两三角形可正确地共享公共边的端点，浮点误差仍会导致直线与公共边的相交测试出现问题。但将直线转换为圆柱体或胶囊体后，只要满足 $r > e/2$ 这一条件即可。该方案适用于各种情况，即使两三角形间存在明显的缝隙。在实际应用中，r 通常远大于 e，因此，浮点误差大大降低，从而实现了健壮的厚对象测试。图 11.10 显示了上述两种测试类型之间的不同之处。厚对象与相应的薄对象相比，其测试开销往往较大，这也可视为厚对象测试的一个缺陷。其中，厚对象测试并不仅局限于相交测试，还将涉及距离的计算。

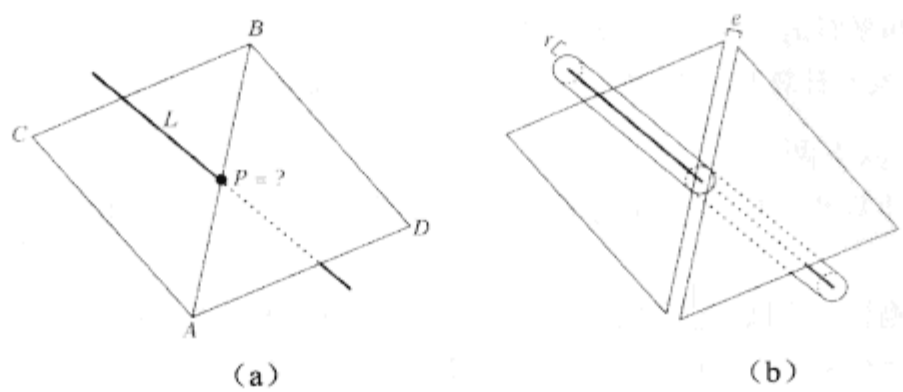


图 11.10 （a）如 11.3.3 节所述，浮点误差使得直线 L 穿越三角形 ABC 和 ADB，且不与任何一个三角形相交。（b）使用厚对象测试时，例如胶囊体，若厚对象的半径大于 $e/2$（其中 e 为最大间隙处的宽度值），则可有效地调整几何体中的间隙

11.4　区间计算

一种记录浮点误差范围的方法是使用区间计算（区间分析，[Moore66]）。

在区间计算中，通过将各数值定义为某一有效区间$[a, b]$，进而使得误差容值内置于自身的计算过程中。最终，针对数值 x，可表示为 $a \leqslant x \leqslant b$。这里需要定义一些运算符以正确地实施区间计算，并获取新的区间值。某些常见的运算符如下所示：

$$[a, b] + [c, d] = [a + c, b + d]$$

$$[a, b] - [c, d] = [a - d, b - c]$$

$$[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

$$[a, b]/[c, d] = [a, b] * [1/d, 1/c],\ 0 \notin [c, d]$$

$$[a, b]^2 = \begin{cases} [0, \max(a^2, b^2)], & 0 \in [a, b] \\ [\min(a^2, b^2), \max(a^2, b^2)], & 0 \notin [a, b] \end{cases}$$

不同于正则浮点运算，区间计算的结果将定义一个舍入误差范围，并试图证明下列事实：表达式的目标值最终位于结果区间内。随后，该范围区间将会与其他对象的范围区间进行比较：若区间不存在交集，则两对象间将处于分离状态。

11.4.1 节将集中显示一些区间计算实例，随后将结合碰撞检测问题对其实现进一步的讨论。

11.4.1　区间计算实例

首先，考查表达式 $x(6 - x)$且 $x = [1, 3]$，如下所示：

$$x(6 - x) = [1, 3]([6, 6] - [1, 3]) = [1, 3][3, 5] = [3, 15]$$

在此例中，对于给定的 x，表达式的最终范围区间为$[5, 9]$，且包含于$[3, 15]$。同时，这也反映了区间计算的一个主要问题：结果区间的范围通常会超出实际区间许多。在较为复杂的嵌套表达式中，这种“误差爆炸”最终将导致计算区间不可用。这种“依赖问题”具体表现为：每一次出现于表达式中的变量都将视为一个新的变量且依赖于其先前的状态。此例中，用于乘法操作的子表达式 x 和 $6 - x$ 皆依赖于 x。但乘法操作对此并不关注，因此结果区间最终将出现不必要的扩展。因此，若相关变量多次出现于表达式中，最终结果将超出表达式的实际范围。

这种情况也并非不可避免：对相关表达式加以重新组织即可消除这种依赖关系。例如，当 $x = [3.0, 3.1]$时，下列表达式：

$$f(x) = x/(x - 1)$$

其结果为 $f([3.0, 3.1]) = [3.0, 3.1]/[2.0, 2.1] = [1.4, 1.6]$。同时，其等价表达式可表示为：

$$f(x) = 1 + 1/(x - 1)$$

其中，变量 x 只出现了一次。最终结果为：$f([3.0, 3.1]) = 1+1/[2.0, 2.1] = 1+[0.476, 0.5] = [1.476, 1.5]$——结果区间变得更加紧凑。类似地，对于下列表达式：

$$g(x, y) = \frac{x + y}{x - y}, h(x, y) = \frac{xy}{x + y}$$

其紧凑格式可记为：

$$g(x,y)=1-\frac{2}{1-x/y},h(x,y)=\frac{1}{1/x-1/y}$$

除此之外，对于平方操作，其紧凑格式可写为 x^2，而非 $x*x$。但是，改写表达式以调整某一变量的出现次数也会引起其他问题，例如前面已讨论过的消位误差问题。总之，为了防止依赖关系的扩大化，应尽可能地避免出现冗长的计算链。

另外，可将输入区间划分为相应的子区间，待计算完毕后，可求得结果区间的并集以获取最终的结果区间（子区间划分思想亦出现于区间 Newton 法中，针对给定的输入区间，该算法可健壮地实现函数求解。相关内容可参见[Hansen92]）。下列示例实现了紧凑区间的子区间划分过程，再次考查表达式 $x(6-x)$，且有子区间 $x=[1,2]$和 $x=[2,3]$。原表达式如下所示：

$$x(6-x)=[1,2]([6,6]-[1,2])=[1,2][4,5]=[4,10]$$

经过子区间划分后，原表达式可写为：

$$x(6-x)=[2,3]([6,6]-[2,3])=[2,3][3,4]=[6,12]$$

这里，两个区间的并集为[4, 12]，且在 $x=[1,3]$时，该结果区间将比[3, 15]更为紧凑。

不难看出，采用浮点格式的区间计算，其实现过程较为直观。其中，各区间可表示为数值组合，相关计算操作可定义为参数组合。最终，将生成代表结果区间的组合数据。唯一值得注意的地方在于：为了确保区间端点的机器可示，区间需向外进行舍入操作以实现结果区间的正确性。注意，相应的舍入操作以及对两个数字需要执行的存储、计算，将使软件实现的区间计算比本地浮点操作约慢上一个数量级。然而，区间计算可实现为并行化操作，因此可对其进行 SIMD 优化。

[Walster98]和[Hickey01]对 IEEE-754 浮点格式的区间计算操作进行了较为详细的阐述，此外，健壮的区间计算操作实现也可参见[Abrams98]。

11.4.2 碰撞检测中的区间计算

区间计算可在碰撞检测过程中体现其实用价值。例如，在隐式数据面 S 和 AABB 之间考查相交检测问题。这里，定义 $f(x,y,z)=0$ 为相关数据面的隐式函数。当前，针对 AABB 的 x、y、z 区间，可采用区间计算以求解 f。此处，$[r_1,r_2]=f([x_1,x_2],[y_1,y_2],[z_1,z_2])$表示最终的结果区间。对于位于 S 外部的全部顶点，皆有 $f(x,y,z)>0$。因此，若 $r_1>0$，则 AABB 与 S 不相交。另外，若最终的计算结果实现为紧凑型区间，则有：若 $r_1\leqslant 0\leqslant r_2$，AABB 将与 S 相交；若 $r_2<0$，则 AABB 位于 S 的内部。另一个更加具体的例子以如下方式给出：根据最小顶点(a_x, a_y, a_z)以及最大顶点(b_x, b_y, b_z)定义 B 为某一 AABB；定义 S 为球心(c_x, c_y, c_z)、半径为 r 的球体，进而执行二者的相交计算。函数 $f(x,y,z)$如下所示：

$$f(x,y,z)=(x-[c_x,c_x])^2+(y-[c_y,c_y])^2+(z-[c_z,c_z])^2-[r,r]^2$$

最终，$[r_1,r_2]=f([a_x,b_x],[a_y,b_y],[a_z,b_z])$。由于$[a_x,b_x]$、$[a_y,b_y]$和$[a_z,b_z]$只出现一次，因此若 n^2 采取了优化处理（即不使用 $n*n$ 这种格式），则最终可得到较为紧凑的范围区间。因此，若 $r_1>0$，则 B 位于 S 的外部；若 $r_1\leqslant 0\leqslant r_2$，则 B 与 S 相交；若 $r_2<0$，则 B 位于 S 内部。更多关于碰撞检测的区间计算，可参见[Mitchell90]、[Duff92]、[Snyder93]以及[Redon02]。

11.5　精确计算和近似计算

虽然浮点数使用方便，但正如前文所述，其精度常受限于舍入操作。为了有效地避免该问题，一种解决方案是前期处理相关的浮点数，且暂不使用精确计算模式。所谓精确计算模式是指：在计算过程中全部数值位皆无损失，从而实现健壮的数值计算。然而，对这一类计算须辅以高精度数值方可执行。一种较好的处理方案是采用高精度运算库，但这对于实时计算则过于缓慢。

实际情况是，高精度计算并非不可或缺。在多种场合下，相关计算不过是执行某些布尔表达式或断言，例如 3.1.6 节中的 ORIENT2D()、ORIENT3D()、INCIRCLE2D()和 INSPHERE()判别式。由于上述断言表达式通常具有固定的复杂度，因此可限定相应的位数以实现准确的表达式计算。所以，相关计算仅需使用扩展精度（而非高精度）即可，一般不会超过 64 位或 128 位精度。而针对某些特定算法，常需要准确的断言。例如，可采用准确的 ORIENT2D()和 ORIENT3D()断言以完成相应的凸体计算。在计算几何领域，采用扩展精度的断言计算已经被很好地研究，并提出了多种高效的实现方法，这其中包括[Priest91]、[Fortune93]以及[Shewchuk96a]。

然而，准确的断言操作仅是某些应用的必备条件之一。在特定的碰撞检测应用中，常需要返回交点信息（如直线与平面之间的交点）。与线段的端点或平面方程系数相比，交点则需要更高的数值精度。而随后的交点计算等操作过程依然需要高精度数值的支持。为了防止无界精度并确保相关计算在实时应用程序中的正确执行，需要交点以某种方式进行截位或舍入操作——一旦出现了舍入操作，计算结果将不再精确。

下面将分别讨论精确计算的整型实现方式，以及与其对应的线段-平面的相交测试方案，并进一步阐释如何将交点截位至整型坐标。

11.5.1　采用整型数据实现精确计算

如果不出现数值上溢这一情况，加法、减法以及乘法的整型操作可得到准确的结果（除法操作则需要单独处理，后文将对其加以讨论）。不同于对应的浮点操作，上述整型计算将遵循结合律。因此，在相交测试中，共享边皆可视为同一有向线段。

为了防止上溢的发生，整型结果变量需要具备足够的精度位。对于包含 n 位的两个加、减法操作数，其结果一般需要 n+1 个精度位；而两个 n 位乘法操作数其结果将生成 $2n$ 个精度位。例如，考查两个 3D 向量的点积操作，且 CPU 支持 64 位整型计算。若点积结果处于 64 位范围内，则点积内部的 3 个乘积值将不得超过 62 位。否则，待执行加法操作后将会出现上溢。由于乘积结果不能超过 62 位，因此各乘积项不可超过 31 位。综上所述，64 位的点积操作将只支持 31 位的向量分量。这里，若表达式中包含两个点积的加法操作，则其附加位将消失且向量分量最多为 30 位宽可保证不出现上溢。类似的分析过程也可用于其他表达式中。

除非对上溢执行显式测试，否则很难发现该问题。一类较为直接的上溢检测方法是将操作数设置为较大的整型值，进而执行相应的计算操作并与低精度结果值加以比较。若二者不等，则原结果值上溢（值得注意的是，单精度或双精度浮点也可用于整型上溢测试）。如果测试数值已经为当前所支持

的最大整数类型，则该测试将无法正常工作。此处，考查无符号整数 a 和 b 的加法操作，并通过测试输入参数对 $c = a + b$ 进行上溢检测，如下所示：

```
bool overflow = (a >~b);
unsigned int c = a + b;
```

或者，也可对最终结果进行测试（如果结果值小于任一参数，则发生上溢）：

```
unsigned int c = a + b;
bool overflow = (c < a);
```

而对于有符号数 a 和 b，上溢测试过程则稍显复杂。这里，可将输入参数与 INT_MAX 和 INT_MIN 进行比较，如下所示：

```
bool overflow = (a > 0 ? (b > INT_MAX - a) : (b < INT_MIN - a));
signed int c = a + b;
```

也可首先执行加法操作，而后检测 a、b 符号是否统一、结果值的符号是否与其相反（需要注意的是，该行为并未在相关语言标准中加以定义），如下所示：

```
signed int c = a + b;
bool overflow = ((a^b) >= 0 && (b^c) < 0);
```

类似的上溢检测过程也可用于减法和乘法操作。

通过分配足够的数位以存储中间值和结果值，可有效地避免上溢（及其测试）现象，这通常需要实现扩展精度操作。采用 CPU 所支持的进位标志（或类似机制）可有效地实现基于扩展精度的加法和减法操作。此处，假设相关数据为 32 位整型值，64 位无符号数的加法操作可表示为：

```
typedef uint32 uint64[2];

void Uadd64(uint64 x, uint64 y, uint64 res)
{
    uint32 a = x[1] + y[1];      // Compute sum of higher 32 bits
    uint32 b = x[0] + y[0];      // Compute sum of lower 32 bits
    if (b < x[0]) a++;           // Carry if low sum overflowed
    res[0] = b;
    res[1] = a;
}
```

两个 32 位数的扩展精度乘法操作将生成一个 64 位的结果值，这里，令 x 和 y 分别表示两个 32 位数，并按下列方式对其加以改写：$x = x_h2^{16} + x_l$，$y = y_h2^{16} + y_l$。则 xy 等价于：

$$xy = (x_h2^{16} + x_l)(y_h2^{16} + y_l) = x_hy_h2^{32} + (x_hy_l + x_ly_h)2^{16} + x_ly_l$$

其中，x_hy_h，x_hy_l，x_ly_h 以及 x_ly_l 皆为硬件所支持的 16 × 16→32 位乘法操作。32 × 32→64 位的乘法操作代码如下所示：

```
void Umult32to64(uint32 x, uint32 y, uint64 res)
{
    uint16 xh = x >> 16, xl = x & 0xffff;
```

```
    uint16 yh = y >> 16, yl = y & 0xffff;
    uint32 a = xh * yh;
    uint32 b = xh * yl;
    uint32 c = xl * yh;
    uint32 d = xl * yl;
    d = d + (b << 16);
    if (d < (b << 16)) a++;
    d = d + (c << 16);
    if (d < (c << 16)) a++;
    a = a + (b >> 16) + (c >> 16);
    res[0] = d;
    res[1] = a;
}
```

类似的操作也可应用于有符号数。另外，相关的文献资源也对扩展精度运算的算法实现进行了详细的描述（如[Knuth97]）。

作为一个应用实例，考查顶点 P 与三角形 T 之间的包含计算，算法将测试顶点 P 是否位于三角形各边的左侧（或右侧，参见 5.3.4 节）。其中 P 和 T 距离原点越远，相关的测试用值也就越大，因而上溢的几率也就越大。若 P 位于 T 内部，则也必定位于其 AABB 内部。因此，首先在顶点 P 与三角形 T 的 AABB 之间进行测试，从而可相应地推迟（或完全避免）上述包含计算（如图 11.11（a）所示）。由于 AABB 测试只进行某些比较操作且根本不执行任何运算，因此也就不会产生上溢问题。

如果 P 和 T 通过了 AABB 测试，二者将全部转换至以 T 的 AABB 为中心的新的坐标系统中（如图 11.11（b）所示）。这里，假设三角形相对于坐标空间较小，因而 T 以及顶点 P（顶点 P 位于其 AABB 内部）的坐标分量值也不会太大，进而可确保不会出现上溢现象。

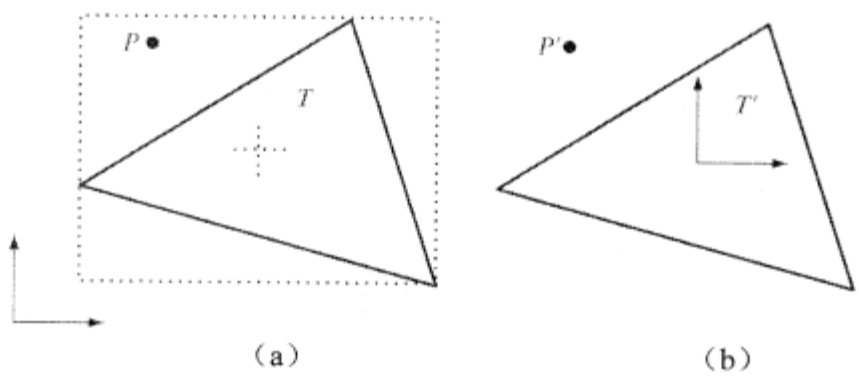


图 11.11　（a）首先在点 P 与三角形 T 的 AABB 之间进行测试，以降低点-三角形包含测试所需的精度。（b）若点 P 通过测试，则在以 T 的 AABB 为中心位置的新坐标系统中执行 P 和 T 之间的相交测试

值得注意的是，单精度浮点数可在[−224, 224]范围内准确地表示整型数据。因此，若输入、输出值位于该范围内，则基于整型的浮动运算（限于加法、减法以及乘法操作）将不会出现精度损失。类似地，双精度浮点数可表示的整数范围为[−253, 253]。鉴于此，双精度浮点数可实现 26 位整型数据的乘法操作，以及 25 位的 3D 点积操作。

尽管加法、减法以及乘法运算操作起来相对简单一些，整型除法操作还需要考虑其他一些问题。11.5.2 节将集中讨论除法操作的处理方案，相关实例将在 11.5.3 节中给出。

11.5.2 整型除法

在相关计算过程中，若不加以妥善处理，除法操作将会产生一系列较为严重的问题。这里，再次考查线段 $S(t) = A + t(B - A)$（$0 \leqslant t \leqslant 1$）与平面 P 之间的相交测试，其中，P 的平面方程记为$(\mathbf{n} \cdot X) = d$。求解 t 则有：

$$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A)) = t_{nom}/t_{denom}$$

由于整型除法舍入操作可能导致最终结果趋于 0，因此直接对表达式进行计算并确认 $0 \leqslant t \leqslant 1$ 以判断 S 与 P 之间是否相交将使得测试失效。例如，若 t 值为-0.4 或 1.2，则对应的整型值为 0 和 1，其判断结果也由“分离”变为“相交”。所以，正确（准确）的 S、P 间相交测试需要消除这一类除法操作，即表达式 $0 \leqslant t \leqslant 1$ 两边都乘以分母 t_{denom}，如下所示：

$$\begin{cases} 0 \leqslant t_{nom} \leqslant t_{denom}, & t_{denom} > 0 \\ 0 \leqslant -t_{nom} \leqslant -t_{denom}, & t_{denom} < 0 \end{cases}$$

若 $t_{denom} = 0$，则线段将平行于平面并可视为“不相交”，或者，当且仅当 $t_{nom} = 0$ 时，也可视为 $t = 0$ 处的交点。亦即，线段位于平面内部。

对于所有子表达式中的除法操作皆可采用类似的方式加以消除。例如，可利用叉乘操作将 $a/b > c/d$ 转化为 $ad > bc$（考虑到相关变量的符号，相应结果也可能为“小于”测试）。最终，不等式等价于 $ad - bc > 0$，这相当于计算下列行列式的符号：

$$\begin{vmatrix} a & c \\ b & d \end{vmatrix}$$

一种采用扩展精度显式计算判别式符号的方法为连分式展开法，参见[Michelucci96]。Michelucci 强调，针对有理数 $0 < a < b$，$0 < c < d$，由于：

$$order\left(\frac{a}{b}, \frac{c}{d}\right) = order\left(\frac{d}{c}, \frac{b}{a}\right) = order\left(\left\lfloor \frac{d}{c} \right\rfloor + \frac{d \bmod c}{c}, \left\lfloor \frac{b}{a} \right\rfloor + \frac{b \bmod a}{a}\right)$$

其中，a/b 和 c/d 项的比较测试如下所示：

$$order\left(\frac{a}{b}, \frac{c}{d}\right) = \begin{cases} \text{大于}, \left|\frac{d}{c}\right| > \left|\frac{b}{a}\right| \\ \text{小于}, \left\lfloor \frac{d}{c} \right\rfloor < \left\lfloor \frac{b}{a} \right\rfloor \\ \text{否则}, order\left(\frac{d \bmod c}{c}, \frac{b \bmod a}{a}\right) \end{cases}$$

对于任意值 a、b、c、d，比较测试可实现为：

```
// Compare rational numbers a/b and c/d
int Order(int a, int b, int c, int d)
{
    // Make c and d be nonnegative
    if (c < 0) b = -b, c = -c;
    if (d < 0) a = -a, d = -d;
    // Handle a and/or b being negative
```

```
if (a < 0 && b < 0) {
        int olda = a, oldb = b;
        a = c; b = d; c = -olda; d = -oldb;
}
if (a < 0) return LESS_THAN;
if (b < 0) return GREATER_THAN;

// Make a <= b, exit if order becomes known
if (a > b) {
        if (c < d) return GREATER_THAN;
        int olda = a, oldb = b;
        a = d; b = c; c = oldb; d = olda;
}
if (c > d) return LESS_THAN;

// Do continued fraction expansion (given that 0<=a<=b, 0<=c<=d)
while (a != 0 && c != 0) {
        int m = d / c;
        int n = b / a;
        if (m != n) {
                if (m < n) return LESS_THAN;
                if (m > n) return GREATER_THAN;
        }
        int olda = a, oldb = b;
        a = d % c; b = c; c = oldb % olda; d = olda;
}
if (a == 0) return c == 0 ? EQUAL : LESS_THAN;
    return GREATER_THAN;
}
```

类似的算法也可参见[Avnaim95]。

待上述除法操作消除完毕后，剩余问题即是如何处理实时计算中的除法操作，它们往往也是问题的根源所在，例如线段 S 与平面 P 之间的交点计算。一般情况下，交点需要舍入或截位至整型坐标，因而无法准确地对其加以表示。最终，交点很可能并不位于平面之上。这里，若 S 表示为某一粒子由 A 运动至 B 时的位移，则舍入操作将使得交点位于平面的背面——即该交点位于当前场景世界的外部，若该粒子从截取位置处继续保持运动，这将导致错误的行为状态。

为了使计算结果满足碰撞检测系统所规定的条件，需要对当前问题进行多方考查并谨慎地处理编码问题。11.5.3 节将给出这一问题的相关实例。

11.5.3　采用整型运算处理线段相交问题

继续考查线段 S 与平面 P 间的相交问题，其中，线段定义为 $S(t) = A + t(B - A)$，$0 \leqslant t \leqslant 1$，平面定义为$(\mathbf{n} \cdot X) = d$。此处，假设线段 S 表示点对象 O 在时间 t 内由 A 至 B 的线性位移，则对象 O 与平面的相交时刻为：

$$t = (d - \mathbf{n} \cdot A)/(\mathbf{n} \cdot (B - A))$$

代入到 $S(t)$中可得交点 Q 为：

$$Q = A + [(d - \mathbf{n} \cdot A)(B - A)]/\mathbf{n} \cdot (B - A)$$

假设线段穿越平面且顶点 A 位于平面的正面，顶点 B 位于平面的背面。由于交点 Q 的坐标分量皆为整数，考虑到整数除法操作导致的舍入误差，交点 Q 最终很可能位于平面之外。因此，需对顶点 Q 进行校正以确保该顶点位于平面内或位于平面的正面空间。首先，校正值要足够大以保证新顶点位置处于平面的正空间内；其次，校正值不可使 AQ 与其他几何体相交。

针对上述第一个问题，为了不失一般性，假设顶点 A 位于平面的正面空间，并对交点 Q 进行调整以使其也处于平面的正面空间内且满足$(\mathbf{n} \cdot Q) \geqslant d$。一种简单的方法是将 Q 中个坐标分量舍入为 $\mathbf{n} \cdot Q$ 的最大值。亦即，若 $\mathbf{n}_x > 0$ 则 Q_x 向上舍入；若 $\mathbf{n}_x < 0$ 则 Q_x 向下舍入。特别地，令 $t_{nom} = (d - \mathbf{n} \cdot A)$，$t_{denom} = \mathbf{n} \cdot (B - A)$，校正值 Q_x 如下所示：

$$Q_x = A_x + (t_{nom}(B_x - A_x) + k)/t_{denom}$$

其中，$k = t_{denom} - 1$ 且向上舍入；$k = -t_{denom} + 1$ 且向下舍入。同理，Q_y 和 Q_z 也可采用类似的方法加以计算。虽然上述过程并不会得到最小的校正值，但对于碰撞检测而言，与交点 Q 的最终坐标位置相比，判断 Q 相对于平面 P 所处的、正确的空间位置则显得更加重要。

函数 **TestSegmentPlane()**的整型版本显示了交点校正值的实现过程。其中，数据类型 **Point**、**Vector** 以及 **Plane** 皆定义为 32 位整型 int32。

```
// Test if segment ab intersects plane p. If so, return 1, along with
// an adjusted intersection point q. If not, return 0
int TestSegmentPlane(Point a, Point b, Plane p, Point &q)
{
    // Compute t value, t=tnom/tdenom, for directed segment ab intersecting plane p
    Vector ab = b - a;
    int64 tnom = p.d - Dot(p.n, a);
    int64 tdenom = Dot(p.n, ab);
    // Exit if segment is parallel to plane
    if (tdenom == 0) return 0;

    // Ensure denominator is positive so it can be multiplied through throughout
    if (tdenom < 0) {
        tnom = -tnom;
        tdenom = -tdenom;
    }

    // If t not in [0..1], no intersection
    if (tnom < 0 || tnom > tdenom) return 0;
    // Line segment is definitely intersecting plane. Compute vector d to adjust
    // the computation of q, biasing the result to lie on the side of point a
    Vector d(0,0,0);
    int64 k = tdenom - 1;
    // If a lies behind plane p, round division other way
    if (tdenom > 0) k = -k;
    if (p.n.x > 0) d.x = k; else if (p.n.x < 0) d.x = -k;
    if (p.n.y > 0) d.y = k; else if (p.n.y < 0) d.y = -k;
    if (p.n.z > 0) d.z = k; else if (p.n.z < 0) d.z = -k;
```

```
    // Compute and return adjusted intersection point
    q = a + (tnom * ab + d) / tdenom;
    return 1;
}
```

这里，假设 **tnom** 和 **tdenom** 涉及 64 位操作，为了有效地避免上溢问题，相关输入值不可超过 30 位精度。除此之外，计算 **q** 时须采用 64 × 32 位精度的乘法操作以及 96/32 位精度的除法操作。

关于第二个问题，待交点 Q 调整完毕后，该交点有可能位于其他平面的背面空间内。因此，需要另一相关测试以确定线段 AQ 是否穿越场景几何体。若是，则再次生成一个新交点。当然，该交点也有可能位于第 3 个平面的背面空间内，依此类推。这里，确定位于 m 个平面正面空间内的顶点 Q（满足条件$(\mathbf{n}_i \cdot Q) \geqslant d_i$，$1 \leqslant i \leqslant m$）则属于整型线性规划问题这一范畴（参见 9.4 节），该问题处理起来则较为复杂。因此，这里并不直接求解交点 Q，而是采用一种较为简单的迭代方法：每次与一个平面进行相交计算，若 n 次迭代计算后仍未得到优化交点，则算法退出（返回交点 A）。

图 11.12 显示了线段 AB 与多边形 P_1、P_2 构成的场景之间的交点计算过程。初始状态下，线段 AB 穿越 P_1 和 P_2 的支撑面。由于 AB 经过 P_1 的内部区域（P_2 则不是），因此二者相交。如图所示，调校之前的交点 Q_1 位于 P_1 平面的背面空间，待校正（上一级交点）之后，交点 Q_2 最终将位于 P_1 的正面空间。随后，线段 AQ_2 须与场景进行测试——结果与 P_2 相交并生成未校正交点 Q_3，该交点校正后得到 Q_4。最终，线段 AQ_4 不与任何几何体相交，因此，Q_4 将作为结果交点返回。

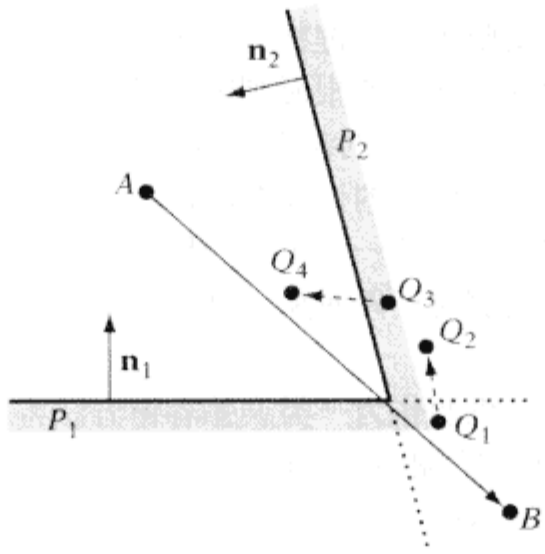


图 11.12　线段 AB 与平面 P_1 和 P_2 之间的相交状态（经适当放大）

综上所述，与浮点数的误差容值算法相比，相应的整型算法则要复杂得多，这也间接阐述了浮点数在碰撞检测计算中的广泛用途。

11.6　提高数值健壮性的进一步讨论

针对健壮的数值计算以及几何算法，除了本章前面所讨论的内容之外，下列建议同样值得考虑：

☑　针对某些关键算法，可通过实现相应的参考算法对其进行测试。例如，可采用某一截然不同

的算法（不排除蛮力算法）。随后，可并行地运行这两类算法。若二者在 k 个有效位上出现异常，则可在此基础上进行进一步的分析。

- ☑ 可采用开销较大的、较为安全的设计方案。例如，添加断言以检测交点是否最终位于相应（厚）平面上；抑或检测反射顶点是否远离该平面。应谨慎处理断言语句自身的（浮点）误差问题。
- ☑ 应确保测试程序中的全部特例均已处理。例如，多数三角形-三角形之间的测试过程都需添加某些特定代码以处理二者共面问题。同时，还需要考虑三角形退化为直线或顶点这一现象。最后，还应特别注意对象间的边界测试问题。
- ☑ 可通过（IEEE-754 标准）所提供的浮点运算标准并依据各种舍入模式测试相关计算对于累积放大舍入误差的敏感性。如果输出结果与期望值差异过大，则当前计算过程将变得不可靠，因而需要进行相应的改写。
- ☑ 对同一数值应避免使用多个不同的公式进行计算。例如，当计算值用于决策变量时，累积误差导致的差异将使得计算结果前后不一致（参见 11.3.3 节中的标量三重积共享机制）。
- ☑ 当采用浮点格式时，以原点为中心区域进行计算可节省符号位所占据的附加精度位。类似的方案也可应用于有符号整数。

11.7 小结

碰撞检测系统中的多项内容均受限于数值健壮性问题。本章主要阐述了这一类问题的起源，并集中讨论了浮点误差问题。深入理解浮点误差的根源则需要了解浮点数的表示方法。目前，普遍采用的浮点格式为 IEEE-754 标准，这也是本章首选的参考标准。

本章深入剖析了采用浮点格式的、健壮的计算方法，相关内容包括：首先，采用容值对累积误差加以屏蔽。这里，厚平面可视为容值误差的一个重要应用。其次，共享机制将（以一种稍显不同的方式）确保数值免于重复计算。本章还探讨了厚对象（胖对象）的使用方法，这一类对象将有助于安全地消除误差。

本章还引入了区间运算机制，该机制作为一类误差容错方法将对误差计算的区间边界实施维护。最后，本章还详细讲解了基于准确计算的相关选择方案。第 12 章将继续讨论与输入几何体特征相关的健壮性问题，例如对象的缝隙以及凹体对象。

第 12 章　几何健壮性

与数值健壮性问题一样，如何正确地构造碰撞测试几何体是另一个重要的课题。如果几何健壮性需求不能得到满足，邻接数据面之间的细小缝隙将会导致其他对象穿越原本完好无损的几何实体。下列内容涵盖了碰撞检测代码中与几何特征相关的一系列问题：

- ☑ 冗余顶点、边以及数据面。
- ☑ 退化数据面，这其中包括：非凸数据面（或称之为自身相交的数据面）、面积为 0 或接近于 0 的数据面。
- ☑ 包含错误表示方向的单面数据面。
- ☑ 邻接数据面间的孔、缝隙、间隙以及 t-连接。
- ☑ 共面现象。

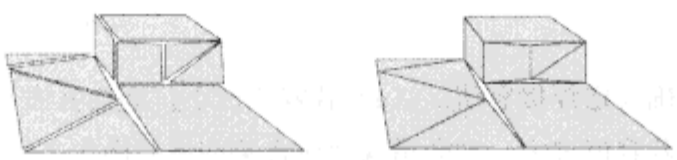
另外，交叉面是否视为一类问题则依据所采用的碰撞解决方案而定。特别地，若数据面间具有不同的表面特征，则交叠的数据面通常会导致问题的出现。[Gu01]中列举了大量的 CAD 中所出现的问题模型，当然，多数错误也会影响到这里所讨论的碰撞检测几何体。

关于几何体的错误排除以及校正方法，已有多部相关文献对其加以研究，多数方案都针对 CAD 实体模型的边界表示方法进行了着重分析（如[Barequet97]和[Murali97]）。需要注意的是，虽然碰撞几何体有时会作为实体模型加以构造，但其常用的建模手段仍是网格和多边形汤。不同于 CAD 几何体，这一类几何体不一定具备相应的连接性或拓扑性等期望特征。亦即，数据面间可能呈现分离、交叉以及多个面共享一条边等状态。同时，CAD 中的几何体也不会定义为带孔的封闭网格。而对于任意类型的网格而言则不存在上述假设，且需要针对几何体中的缝隙和孔采取不同的处理方法。

针对任意类型的网格修补存在多种可选方案。特别地，本章将集中讨论与多种算法相关的核心步骤，其中包括：

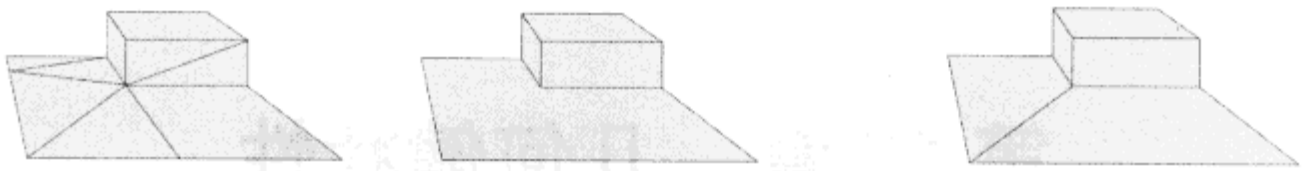
1. 顶点焊接。
2. 邻接数据面之间的缝隙修补。
3. 合并共面数据面为单一面。
4. 凸（三角形）面片的分解。

图 12.1 显示了上述步骤在多边形汤实体模型中的应用。除此之外，本章还将探讨邻接信息的计算、存储方案，以及针对问题输入数据的快速检测方法。



（a）原始多边形汤模型　　（b）执行顶点焊接

图 12.1　将多边形汤转换为具有良好结构的网格包括下列核心步骤：顶点焊接、消除 t-连接、合并共面数据面以及凸面片剖分



（c）t-连接修复　　（d）合并共面数据面　　（e）凸（四边形）面片剖分后的最终模型

图 12.1　将多边形汤转换为具有良好结构的网格包括下列核心步骤：顶点焊接、消除 t-连接、合并共面数据面以及凸面片剖分（续）

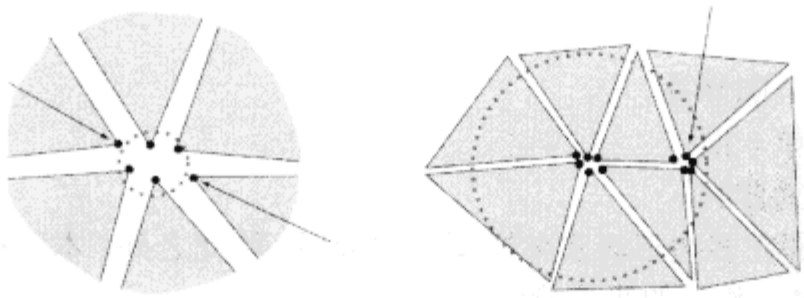
一般来讲，任意类型输入几何体的修补过程通常都比较复杂。[Erikson95]中也给出了网格修补操作中的具体分析。

12.1　顶点焊接

碰撞几何体中的多边形顶点应彼此连接，但事实并非如此。因此，这种单一顶点的冗余表达方式常会在多边形间生成缝隙。虽然肉眼难以察觉，但这些缝隙仍会给碰撞检测系统带来问题——可导致碰撞查询操作（如光线测试）穿越原本完好无损的几何实体。另外，冗余顶点还将破坏原邻接数据面之间的连接信息。同时，冗余顶点还会占用额外的内存空间。例如，包含冗余顶点的三角形网格可能会占据原网格 3 倍的内存空间。

首先，顶点修正问题需要获取全部彼此临近的顶点组合或顶点集，随后，将其连接为单一顶点——这一过程称作顶点焊接（或顶点平移、顶点合并）。当相关顶点位于某一距离范围内时，则可认为其彼此临近并称之为焊接容值或焊接 epsilon 值。另外，位于焊接容值内的全部空间范围则称作顶点的焊接域。

设定一个恰当的焊接距离是非常重要的。如果焊接容值过小，将遗漏某些焊接顶点；相反，若焊接距离过大，将使得焊接过程包含某些冗余顶点，从而导致几何体数据面中的相应边出现塌陷，如图 12.2 所示。同时，焊接操作常需要用户手工设置焊接距离，自动设置通常较为困难。



（a）　　（b）

图 12.2　应正确设定焊接容值。（a）若容值过小，则会丢失某些焊接顶点。（b）若容值过大，则会错误地包含某些焊接操作之外的顶点

这里，主要问题将集中于特征点的选择。考查直线上的多个顶点，其中，顶点间等间距排列且每一组顶点皆位于焊接距离内（如图 12.3 所示）。此处假设对上述顶点执行迭代计算，且各顶点均参照

前一焊接顶点进行比较。若前顶点并未处于焊接距离内，则将其保留，否则可采用下列方案进行替换：

1. 利用前一保留顶点替换当前顶点。
2. 利用当前顶点替换前一保留顶点。
3. 利用前一顶点和当前顶点的中间顶点替换这两个顶点。

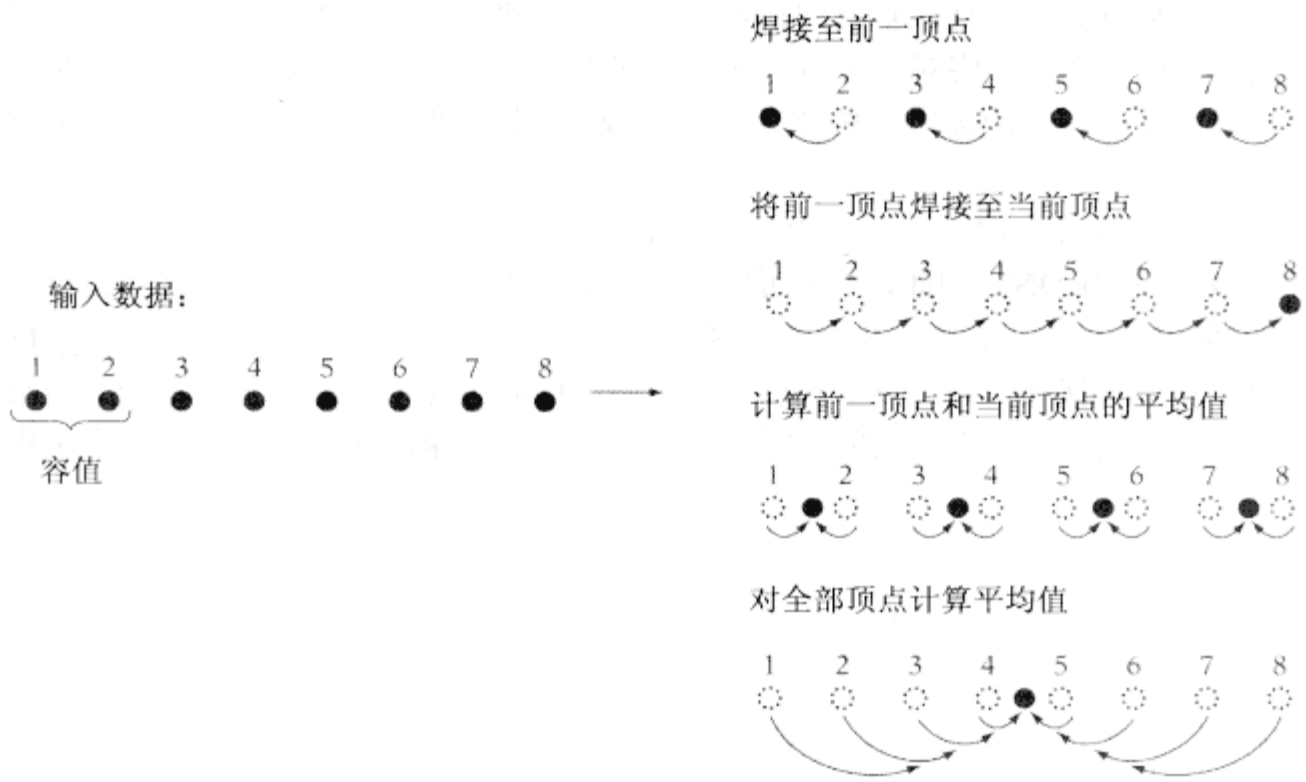


图 12.3　采用不同方法在焊接容值距离内焊接一组顶点

如图 12.3 所示，上述方案的计算结果截然不同。方案 1 较具吸引力，该方案将顶点偏移值（原顶点与最终位置间的距离）限定在焊接 epsilon 值内。而方案 2 将会导致最坏情况的出现：设定 n 为焊接顶点数，则顶点偏移值将正比于 n。针对方案 3，虽然图中未标明，该方案仍会出现顶点偏移值正比于 n 这一最坏情况（考查图 12.3 中加倍焊接 epsilon 值后的效果）。另外，若不考虑导致额外顶点偏移的退化现象，其最终的平均值将与代表点的期望值十分接近。

顺序操作依赖于顶点间的位置。顺序操作同样要处理位于当前顶点焊接距离内的多个前顶点。那么，焊接操作采用最近顶点抑或是全部顶点？除此之外，针对上述 3 种方案，仍须考虑退化现象的发生，其中，两个十分接近的顶点将生成两个不同的焊接位置。

最后一个问题连同位于焊接距离内的、多个前顶点的选择方案均可通过同步焊接模型加以解决。同步模型将确定顶点对应的全部集合，进而该集合内的每一个顶点均位于其他顶点的容值距离内，如图 12.3 所示。与顺序焊接操作相比，该方案计算开销较大。另外，该方案易受到退化现象的影响，从而使得焊接值变得不确定。

最后一个问题将讨论如何在焊接距离内确定顶点。当然，可采用蛮力法在顶点 A 以及各对应顶点 B 之间进行测试。虽然该算法易于实现，但对于较大的数据集，其执行速度则难以令人满意。可使用空间划分数据结构将顶点划分为多个独立的顶点集，并对集合中的顶点进行测试，从而实现焊接操作的加速计算过程。从本质上讲，若将顶点及其焊接邻接顶点视为供相交测试使用的小型球体或盒体，则顶点焊接操作可归于碰撞检测问题。因此，任一快速的碰撞检测算法均可用于顶点焊接操作中。

实际处理方案则是采用 7.1.3 节中的哈希均匀网格，该优化方案的算法复杂度为 $O(n)$。从技术层面上讲，算法最坏情况下的复杂度为 $O(n^2)$，即全部顶点皆位于同一网格单元中，且不会产生任何顶点焊接操作。当然，实际操作过程中则不会出现这一情况。另外，该方案还是一类在线算法，在顶点添加后即可执行焊接操作，而无须等待全部顶点添加完毕。

下面将采用顺序处理方案实现顶点的焊接操作，并基于前一顶点焊接当前顶点。算法将一次添加一个顶点至网格单元中，随后，将对位于当前顶点 V 的邻接焊接网格单元进行测试。若某一网格单元包含顶点 W，且 V 位于其焊接距离内，则 W 作为焊接顶点返回；否则，将 V 添加至其对应的网格单元中且 V 作为焊接顶点返回。

这里需要强调两点注意事项。首先，为了有效地限制参与测试的网格单元的数量，**CELL_SIZE** 应大于 **2 * WELD_EPSILON**。因此，V 附近的正方形焊接邻域则最多与 4 个邻接网格单元相交（在 3D 环境下，正方体焊接邻域则最多与 8 个邻接网格单元相交）。在实际操作中，**CELL_SIZE** 常作为一个系数或按照 2 倍于焊接邻域这一规则加以设定。由于仅需对与 V 附近容值区域相交的网格单元进行检测，因此，在平均情况下只需测试单一网格单元中的数据内容。图 12.4 显示了两个采样点时的算法工作过程。

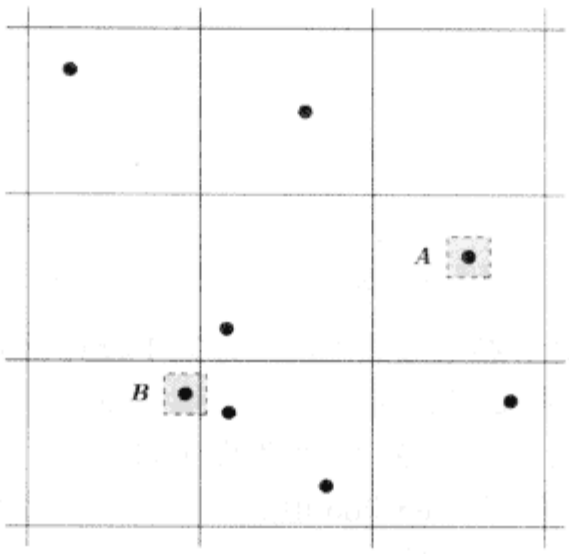


图 12.4　在顶点焊接操作过程中，仅对与顶点容值区域范围内相交的网格单元（采用灰色表示）进行测试。针对顶点 A，只需测试一个网格单元；而对于顶点 B，则应检测两个网格单元

其次，按照哈希映射规则，两个或多个顶点网格邻域可能会映射至同一哈希桶中——这并不会对算法产生任何副作用，其中，将会对位于测试网格单元内的全部顶点进行距离测试。最坏情况下，针对哈希桶中的数据内容所执行的焊接测试只会出现少许附加测试顶点。下列代码显示了 2D 环境下的哈希网格实现过程。首先，定义下列常量值：

```
#define MAX_VERTICES 100000        // max number of vertices that can be welded at once
#define NUM_BUCKETS 128            // number of hash buckets to map grid cells into
#define CELL_SIZE 10.0f            // grid cell size; must be at least 2*WELD_EPSILON
#define WELD_EPSILON 0.5f          // radius around vertex defining welding neighborhood

// Maps unbounded grid cell coordinates (x, y) into an index
// into a fixed-size array of hash buckets
```

```
unsigned int GetGridCellBucket(int x, int y)
{
    const unsigned int magic1 = 0x8da6b343;    // Large multiplicative constants;
    const unsigned int magic2 = 0xd8163841;    // here arbitrarily chosen primes
    unsigned int index = magic1 * x + magic2 * y;
    // Bring index into [0, NUM_BUCKETS] range
    return index % NUM_BUCKETS;
}
```

这里，顶点链表采用了 3 个数组加以实现。数组 **first** 用于存储哈希桶中首顶点的索引；数组 **vertex** 存储顶点数据；数组 **next** 则包含了哈希桶中下一个顶点的索引。

```
int first[NUM_BUCKETS];              // start of linked list for each bucket
int next[MAX_VERTICES];              // links each vertex to next in linked list
Point vertex[MAX_VERTICES];          // unique vertices within tolerance
int numVertices;                     // number of unique vertices currently stored
```

当给定哈希桶索引 bucket 以及顶点 **v** 后，函数 **LocateVertexInBucket()**将计算是否存在一个顶点位于 **v** 的 **WELD_EPSILON** 范围内。若是，则该顶点返回至 **weldVertex** 中。

```
int LocateVertexInBucket(Point v, unsigned int bucket, Point **weldVertex)
{
    // Scan through linked list of vertices at this bucket
    for (int index = first[bucket]; index >= 0; index = next[index]) {
        // Weld this vertex to existing vertex if within given distance tolerance
        if (SqDistPointPoint(vertex[index], v) < WELD_EPSILON * WELD_EPSILON) {
            *weldVertex = &vertex[index];
            return 1;
        }
    }

    // No vertex found to weld to. Return v itself
    *weldVertex = &v;
    return 0;
}
```

若焊接容值距离范围内不存在顶点，则通过导入链表而将 **v** 添加至给定的哈希桶中。

```
void AddVertexToBucket(Point v, unsigned int bucket)
{
    // Fill next available vertex buffer entry and link it into vertex list
    vertex[numVertices] = v;
    next[numVertices] = first[bucket];
    first[bucket] = numVertices++;
}
```

当相关计算工作在浮点数环境下时，对于较大的坐标值，其与较小 epsilon 值的加法操作不会对该坐标值产生任何影响（参见第 11 章）。对于这种情况，代码不会得到正确的邻接网格单元测试结果。因此，健壮的代码应确保 **WELD_EPSILON** 相对于坐标值足够大（在实际应用中，可在代码中加入断

言以检测该问题的出现。若断言被触发，则相应地增加 epsilon 值，依此类推，直至该问题消除）。相应的焊接代码如下所示：

```
Point *WeldVertex(Point *v)
{
    // Make sure epsilon is not too small for the coordinates used!
    assert(v->x - WELD_EPSILON != v->x && v->x + WELD_EPSILON != v->x);
    assert(v->y - WELD_EPSILON != v->y && v->y + WELD_EPSILON != v->y);

    // Compute cell coordinates of bounding box of vertex epsilon neighborhood
    int top = int((v->y - WELD_EPSILON) / CELL_SIZE);
    int left = int((v->x - WELD_EPSILON) / CELL_SIZE);
    int right = int((v->x + WELD_EPSILON) / CELL_SIZE);
    int bottom = int((v->y + WELD_EPSILON) / CELL_SIZE);

    // To lessen effects of worst-case behavior, track previously tested buckets
    unsigned int prevBucket[4];      // 4 in 2D, 8 in 3D
    int numPrevBuckets = 0;

    // Loop over all overlapped cells and test against their buckets
    for (int i = left; i <= right; i++) {
        for (int j = top; j <= bottom; j++) {
            unsigned int bucket = GetGridCellBucket(i, j);
            // If this bucket already tested, don't test it again
            for (int k = 0; k < numPrevBuckets; k++)
                if (bucket == prevBucket[k]) goto skipcell;
            // Add this bucket to visited list, then test against its contents
            prevBucket[numPrevBuckets++] = bucket;
            Point *weldVertex;
            // Call function to step through linked list of bucket, testing
            // if v is within the epsilon of one of the vertices in the bucket
            if (LocateVertexInBucket(*v, bucket, &weldVertex)) return weldVertex;
            skipcell: ;
        }
    }

    // Couldn't locate vertex, so add it to grid, then return vertex itself
    int x = int(v->x / CELL_SIZE);
    int y = int(v->y / CELL_SIZE);
    AddVertexToBucket(*v, GetGridCellBucket(x, y));
    return v;
}
```

针对全部顶点，可重复调用 **WeldVertex()**函数以实现其焊接操作。

```
void WeldVertices(Point v[], int n)
{
    // Initialize the hash table of linked vertex lists
    for (int k = 0; k < NUM_BUCKETS; k++)
        first[k] = -1;
```

```
    numVertices = 0;

    // Loop over all vertices, doing something with the welded vertex
    for (int i = 0; i < n; i++) {
        Point *pVert = WeldVertex(&v[i]);
        if (pVert != &v[i])
            ...report v[i] was welded to pVert...
    }
}
```

待全部顶点处理完毕后，**vertex** 数组的前 **numVertices** 项将包含全部的独立顶点，因而可方便地将其输入到相应的数据文件中。

```
fwrite(&numVertices, sizeof(numVertices), 1, stream);      // Output number of verts
fwrite(&vertex[0], sizeof(Point), numVertices, stream);    // Output verts themselves
```

遍历全部数据面并焊接其顶点将有效地剔除未曾使用的顶点，其中仅对数据面引用的顶点进行焊接操作。

这里，较为简单的一种焊接算法是对全部顶点(x_i,y_i,z_i)的坐标分量绝对值之和进行排序。

待顶点排序完毕后，可比较 V_i 的后续顶点 V_j（$i<j$）以实现 V_i 的焊接操作。如下所示：

$$(|x_j|+|y_j|+|z_j|)-(|x_i|+|y_i|+|z_i|)\leqslant 3\varepsilon$$

其中 ε 表示焊接 epsilon 值。该方案虽易于实现，却是一类离线算法，因而全部顶点须进行一次性的焊接操作。另外，若顶点排列为网格状，将会产生退化现象：对角线$|x_i|+|y_i|+|z_i|=k$ 上的全部顶点都将映射为同一值。

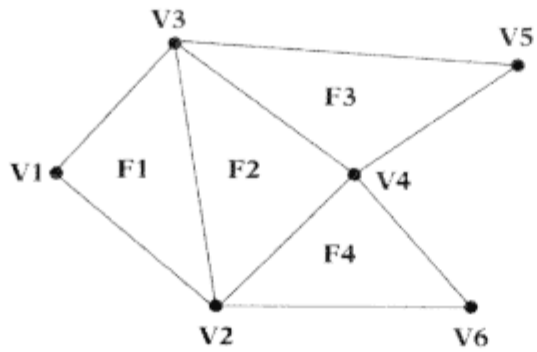
当对非三角形几何体进行焊接操作时，需要注意的一点是，待焊接操作完毕后，最终几何体可能会包含非共面数据面或非凸数据面以及共线顶点。通过 12.5 节中所描述的算法，这一类问题数据面皆可实现三角形化并最终呈现为凸数据面。

综上所述，在经历过顶点焊接操作后，任意两个顶点间的距离均大于焊接容值。在顶点焊接系统的调试过程中，这也可作为实际的测试标准。

12.2　计算邻接信息

针对模型几何体的修正（包括消除缝隙以及合并共面数据面），了解数据间的邻接信息是十分必要的，这些邻接信息描述了模型所连接的数据面、边以及顶点。需要注意的是，模型数据未必提供这一类邻接信息。在最坏情况下，相应数据将只呈现为包含独立数据面的多边形汤。此时则需要计算邻接信息。

常见的几何体表达方式是面表，其中可在面表（通常为三角形，有时也可为四边形）中添加顶点索引。图 12.5 显示了一个小型面表示例。数据面/顶点表形式可视为一种紧凑的表示方案，且可以用来表达各种网格模型。



Face	Vertices (ccw)
F1	(V1,V2,V3)
F2	(V2,V4,V3)
F3	(V3,V4,V5)
F4	(V4,V2,V6)

Vertex	Coordinates
V1	(x1,y1,z1)
V2	(x2,y2,z2)
V3	(x3,y3,z3)
V4	(x4,y4,z4)
V5	(x5,y5,z5)
V6	(x6,y6,z6)

图 12.5　简单网格的数据面和顶点

上述基本格式完全可以应用于相关计算中，但其执行效率仍有改进的余地。为了有效地修正网格几何体，去除可能会对碰撞检测代码带来麻烦的退化现象等问题，还需要计算一些附加的邻接信息。

完全邻接信息将有助于在常量时间内检索某一给定顶点、边或数据面所对应的邻接顶点、邻接边以及邻接数据面等信息。图 12.6 显示了网格的完全邻接信息。

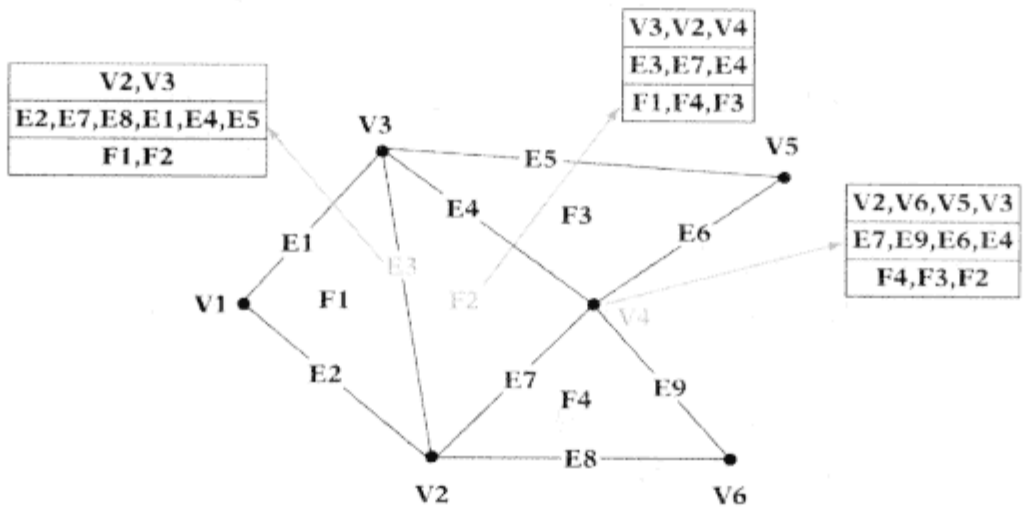


图 12.6　与各顶点、边以及数据面关联的邻接信息有助于实现其邻接顶点、边和数据面的快速访问

然而，存储网格特征数据的全部邻接信息将受到内存空间的限制。针对快速的邻接信息查询和较小的内存需求，一种折中方案是仅存储部分邻接数据，以便在固定时间内获取必要的邻接信息。

针对流形实体模型，存在多种数据结构用以存储局部邻接信息，其中包括：有翼边（winged-edge，[Baumgart75]）、半边（half-edge，[Mäntylä88]）以及有翼三角形（winged-triangle，[Paoluzzi89]）数据结构，如图 12.7 所示。

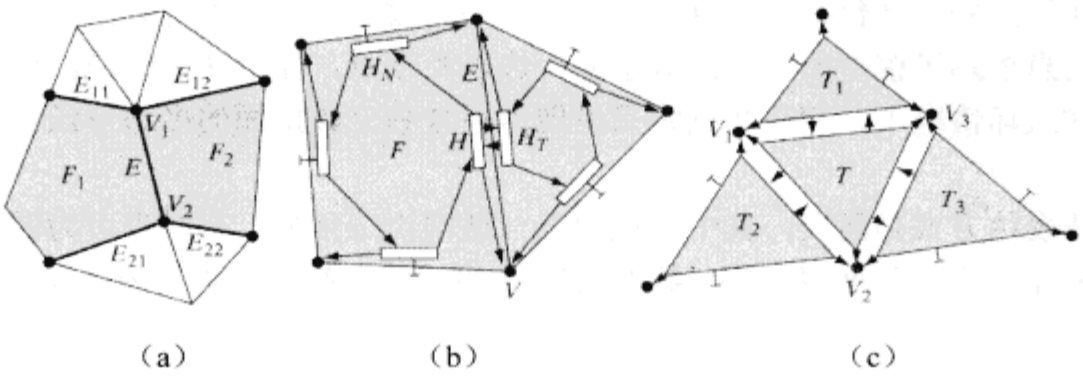


图 12.7　(a) 有翼边 E。(b) 半边 H。(c) 有翼三角形 T

有翼边和半边结构是一类基于边的数据结构。在有翼边结构中，将存储边顶点、该边连接的两个数据面，以及两个连接数据面的前一条边和后一条边；在半边结构中，每一条边由两条半边组成，即两顶点间的有向边且分别指向相反方向。另外，每条边均记录其对应的一组半边、边面、逆时针方向上边面的下一条半边以及半边源点。在两种方案中，顶点和数据面均记录关联（半）边的信息。

有翼三角形结构则是一类基于数据面的数据结构，每一个三角形均存储顶点以及具有公共边的 3 个三角形。另外，顶点还将记录关联三角形的信息。这里，边并未显式地加以存储。上述 3 种数据结构如下所示：

```
// Basic representation of an edge in the winged-edge representation
struct WingedEdge {
    Vertex *v1, *v2;                // The two vertices of the represented edge (E)
    Face *f1, f2;                   // The two faces connected to the edge
    Edge *e11, *e12, *e21, *e22;    // The next edges CW and CCW for each face
};

// Basic representation of a half-edge in the half-edge representation
struct HalfEdge {
    HalfEdge *ht;   // The matching "twin" half-edge of the opposing face
    HalfEdge *hn;   // The next half-edge counter clockwise
    Face *f;        // The face connected to this half-edge
    Vertex *v;      // The vertex constituting the origin of this half-edge
};

// Basic representation of a triangle in the winged-triangle representation
struct WingedTriangle {
    Vertex *v1, *v2, *v3;           // The 3 vertices defining this triangle
    WingedTriangle *t1, *t2, *t3;   // The 3 triangles this triangle connects to

    // Fields specifying to which edge (0-2) of these triangles the connection
    // is made are not strictly required, but they improve performance and can
    // be stored "for free" inside the triangle pointers
    int edge1:2, edge2:2, edge3:2;
};
```

另外一种较为常见的数据结构是 4 叉边（quad-edge），其详细内容可参见[Guibas85]。

针对非流形几何体，也存在相应的数据结构与之对应。例如，放射边（radial-edge，[Weiler88]）以及星形边（star-edge，[Karasick89]）结构。但由于非流形几何体中的某一条边可能会连接任意多个数据面，因此其数据结构与流形几何体相比将会复杂得多。

若初始状态下的数据按照顶点/面表的方式给出，邻接结构的初始化操作需从该顶点/面表中获取邻接信息。一种简单的方法是：针对每一个数据面，分别与其他数据面进行比较，当两个数据面之间具有公共顶点时，即可写入邻接信息。但该方法 $O(n^2)$的时间复杂度使其只能处理尺寸适中的网格数据。下面将详细讨论两种复杂度为 $O(n)$的邻接信息计算方法，并可作为该问题的准优化方案。12.2.1 节将具体阐述针对顶点关联面的面表构造方法；12.2.2 节将全面分析边关联面的边表计算方案。其中，相关准则也可用于任一与之类似的表计算。例如，计算顶点的全部关联边。

12.2.1 计算顶点-面表

当作为输入数据的面/顶点表确定后，即可得到该面的对应顶点。但计算某一顶点的全部数据面其操作复杂度则为 $O(n)$。为了加快搜索速度，可构造一个逆向表：针对每一个顶点，可直接定位顶点的全部关联面。为了简单起见，这里假定全部数据面皆为三角形。算法的基本思想则适用于任意类型的数据面。这里，面表（类型为 **Triangle** 的数组）中定义了 **numTris** 个三角形，如下所示：

```
int numTris;      // The number of triangles
Triangle *tri;    // Pointer to face table array containing all the triangles
```

Triangle 结构按如下方式加以定义：

```
struct Triangle {
    int vertexIndex[3];    // Indices into vertex table array
};
```

由于某一顶点可能会关联多个三角形，因此逆向顶点表须实现为一个与顶点关联的、动态可增长的数据结构。这里，基于数据集合中的每一个顶点，可构造一个三角形数组的索引链表。链表元素按如下方式加以定义：

```
//Linked-list element to keep track of all triangles a vertex is part of
struct TriList {
    int triIndex;          // Index to some triangle the vertex is part of
    TriList *pNext;        // Pointer to the next triangle in the linked list
};
```

此处，顶点数量值 **MAX_VERTICES**（最多）为三角形数量的 3 倍。另外，数组 **triListHead** 将包含表元素指针，并用以存储各表头指针。数组 **triListEntry** 则定义为一个等尺寸的数据池，每当三角形与某一既定顶点表相关联，就相应地分配一个表元素。

```
// If all vertices are unique, there are at most 3 * MAX_TRIANGLES vertices
const int MAX_VERTICES = 3 * MAX_TRIANGLES;

TriList *triListHead[MAX_VERTICES];        // The head of the list maintained for each vertex
TriList triListEntry[MAX_VERTICES];        // Entries of the linked list
```

依据上述结构，主代码将遍历全部三角形，且针对每一个三角形顶点，代码将在链表中插入三角形索引，如下所示：

```
// Reset the list of triangles associated with each vertex
for (int i = 0; i < MAX_VERTICES; i++) triListHead[i] = NULL;
// Reset the triangle list entry counter
int cnt = 0;
// Loop over all triangles and all three of their vertices
for (int i = 0; i < numTris; i++) {
    for (int j = 0; j < 3; j++) {
        // Get the vertex index number
```

```
        int vi = tri[i].vertexIndex[j];
        // Fill in a new triangle entry for this vertex
        triListEntry[cnt].triIndex = i;
        // Link new entry first in list and bump triangle entry counter
        triListEntry[cnt].pNext = triListHead[vi];
        triListHead[vi] = &triListEntry[cnt++];
    }
}
```

因此，针对某一由索引 i 定义的顶点，其全部关联三角形可按如下方式进行访问：

```
for (TriList *pEntry = triListHead[i]; pEntry; pEntry = pEntry->pNext) {
    /* do something with triangle tri[pEntry->triIndex] here */
}
```

一般情况下，同类型的逆向顶点表也是具有实际意义的。例如，可计算与三角形 T 的 3 个顶点关联的各表的并集，进而确定三角形 T 的邻接三角形。但对于这一类特定应用，一种更加有效的方法是计算一个边表，其中数据面与相应边关联。因此，给定一个三角形后，可通过边表搜索连接数据面以实现其邻接三角形的定位。12.2.2 节将讨论该类型表的计算方法。

12.2.2　计算边-面表

前面所讨论的输入数据，其构造方式是由顶点索引的面表。该问题还可以实现为边表，其中各边均与其连接数据面关联。再次说明，为了简化数据面的表达，此处仍将采用三角形。另外，输入数据限定为（有界）流形网格。因此，各边的关联三角形的数量将不超过两个。

通常情况下，三角形和其他类型的数据面均以某一固定排列顺序的相关顶点加以表示（按照惯例，正面观察数据面时，其顶点以逆时针排列），因此，公共边 AB 在两个三角形中的表现形式分别为(A,B)和(B,A)。若将(A,B)和(B,A)视为同一边，则有必要强调顶点 A 和 B 之间的排列顺序。由于顶点可按索引方式定义，因此可采用小索引值优先这一方式表示数据边，即边$(\min(A,B), \max(B,A))$。

当顶点坐标以 3 坐标值而非索引形式给出时，则可采用 x、y、z 最小分量优先的方式实现顶点的排列。下列代码显示了顶点的“词典顺序”排列方式：

```
// Compare vertices lexicographically and return index (0 or 1) corresponding
// to which vertex is smaller. If equal, consider v0 as the smaller vertex
int SmallerVertex(Vertex v0, Vertex v1)
{
    if (v0.x != v1.x) return v1.x > v0.x;
    if (v0.y != v1.y) return v1.y > v0.y;
    return v1.z > v0.z;
}
```

边可采用顶点索引组合加以表示，且表中数据均使用该索引组合，因此，这里将定义一个开放式哈希表而不采用顺序数组。同时，哈希键将从两个顶点索引中获取。各个键均关联一个数据项，其中包含了两个顶点索引以及该边两个关联三角形中的边数据。该数据项的结构如下所示：

```
struct EdgeEntry {
    int vertexIndex[2];        // The two vertices this edge connects to
    int triangleIndex[2];      // The two triangles this edge connects to
    int edgeNumber[2];         // Which edge of that triangle this triangle connects to
    EdgeEntry *pNext;          // Pointer to the next edge in the current hash bucket
};
```

此处，假设全部边均为独立边，则最大边数量 **MAX_EDGES** 为三角形数量的 3 倍。数组 **edgeListHead** 则包含了多个表元素指针，分别指向各边表的表头。哈希键将对该表指针数组进行索引。另外，等尺寸数组 **edgeListEntry** 则可视为一个数据池，待给定边表确定某一个三角形后，以获取各链表中的数据项。

```
// If all edges are unique, there are at most 3 * MAX_TRIANGLES edges
const int MAX_EDGES = 3 * MAX_TRIANGLES;

// Hash table over edges, with a linked list for each hash bucket
EdgeEntry *edgeListHead[MAX_EDGES];
EdgeEntry edgeListEntry[MAX_EDGES];        // Entries of the linked list
```

在构造边表时，主代码将遍历全部三角形以及各三角形中的每一条边。如果某一条边未曾出现于哈希桶所映射的链表中，则需要增加一个新的边数据项链接，并通过该边及其两个顶点以记录当前三角形。如果该边已出现于哈希表中，则可利用当前三角形索引值填充与第二个三角形相关的数据项。这里，函数 **ComputeHashKey()**将计算映射至 **edgeListHead** 数组中的哈希键值。主代码如下所示：

```
// Reset the hash table
for (int i = 0; i < MAX_EDGES; i++) {
    edgeListHead[i] = NULL;
}
// Reset the edge list entry counter
int cnt = 0;
// Loop over all triangles and their three edges
for (int i = 0; i < numTris; i++) {
    for (int j = 2, k = 0; k < 2; j = k, k++) {
        // Get the vertex indices
        int vj = tri[i].vertexIndex[j];
        int vk = tri[i].vertexIndex[k];
        // Treat edges (vj, vk) and (vk, vj) as equal by
        // flipping the indices so vj <= vk (if necessary)
        if (vj > vk) Swap(vj, vk);
        // Form a hash key from the pair (vj, vk) in range 0 <= x < MAX_EDGES
        int hashKey = ComputeHashKey(vj, vk);
        // Check linked list to see if edge already present
        for (EdgeEntry *pEdge = edgeListHead[hashKey]; ; pEdge = Edge->pNext) {
            // Edge is not in the list of this hash bucket; create new edge entry
            if (pEdge == NULL) {
                // Create new edge entry for this bucket
                edgeListEntry[cnt].vertexIndex[0] = vj;
                edgeListEntry[cnt].vertexIndex[1] = vk;
```

```
                edgeListEntry[cnt].triangleIndex[0] = i;
                edgeListEntry[cnt].edgeNumber[0] = j;
                // Link new entry first in list and bump edge entry counter
                edgeListEntry[cnt].pNext = edgeListHead[hashKey];
                edgeListHead[hashKey] = &edgeListEntry[cnt++];
                break;
            }
            // Edge is in this bucket, fill in the second edge
            if (pEdge->vertexIndex[0] == vj && pEdge->vertexIndex[1] == vk) {
                pEdge->triangleIndex[1] = i;
                pEdge->edgeNumber[1] = j;
                break;
            }
        }
    }
}
```

上述代码可与 12.2.1 节中的代码进行合并，这样，顶点表可获得各顶点关联数据面的全部顶点索引值。另外，各顶点还可引用其关联边的某一个（或多个）顶点。这使得特征（顶点、边或数据面）索引项的全部邻接信息访问时间复杂度为 $O(1)$。

12.2.3　连通性测试

作为一个邻接信息的应用示例，这里将考查面表中全部数据面之间的连通性。如果数据面之间并非全部连接，则应输出连通分支的数量以及各连通分支所属的数据面。需要注意的是，该示例只限于三角形。

此处，可在三角形数据结构中添加 **label** 字段以记录连通分支号，从而确定三角形位于哪一个连通分支中。初始状态下，各三角形的 **label** 字段均标记为 0，即不属于任何连通分支。随后，遍历全部三角形且对于每一个未加标记的三角形，当前 **componentNumber** 值加 1，同时，**componentNumber** 值用以标记当前三角形，并采用同一 **componentNumber** 值递归访问该三角形的全部邻接三角形。

若全部连通分支之间均呈现连通状态，则递归调用最终将终止且所有三角形均标记为 **componentNumber** 的初始值。当代码返回至外部循环时，则不存在未标记的数据面且当前调用过程结束，亦即只包含一个连通分支。若存在多个连通分支，递归过程将无法访问某些数据面且最终返回至外部循环，并在递归访问下一个连通分支之前相应地增加其记数值，依此类推。下列代码片段实现了这一过程：

```
// Unlabel all triangles
for (i = 0; i < numTris; i++)
    tri[i].label = 0;
// Loop over all triangles, identifying and marking all components
int componentNumber = 0;
for (i = 0; i < numTris; i++) {
    // Skip triangles already labeled
    if (tri[i].label != 0) continue;
    // Mark this triangle with the current component number
```

```
        tri[i].label = ++componentNumber;
        printf("Component %d starts at triangle %d\n", componentNumber, i);
        // Recursively visit all neighboring triangles and mark them with the same number
        MarkAllNeighbors(i, componentNumber);
    }
```

待上述代码执行完毕后，**componentNumber** 值将包含网格中连通分支的数量。其中，代码调用了一个递归函数以访问给定数据面的全部邻接三角形，如下所示：

```
void MarkAllNeighbors(int triIndex, int componentNumber)
{
    int neighborIndex;
    // Visit neighboring triangles of all three edges (if present)
    for (int i = 0; i < 3; i++) {
        neighborIndex = GetTriangleNeighbor(triIndex, i);
        // If there is a neighboring triangle not already marked...
        if (neighborIndex >= 0 && tri[neighborIndex].label != componentNumber) {
            // ...mark it, and visit it recursively
            tri[neighborIndex].label = componentNumber;
            MarkAllNeighbors(neighborIndex, componentNumber);
        }
    }
}
```

上述函数实现了递归调用，其中，传入三角形索引以及边号并对边表进行查询，以获取该公共边的其他连接三角形并返回其索引（若不存在，则返回-1），如下所示：

```
int GetTriangleNeighbor(int triIndex, int edgeNum)
{
    // Get vertex indices for the edge and compute corresponding hash key
    int vi = tri[triIndex].vertexIndex[edgeNum];
    int vj = tri[triIndex].vertexIndex[(edgeNum + 1) % 3];
    if (vi > vj) Swap(vi, vj);
    int hashKey = ComputeHashKey(vi, vj);
    // Search hash bucket list for a matching edge
    for (EdgeEntry *pEdge = edgeListHead[hashKey]; pEdge != NULL; pEdge = pEdge->pNext) {
        // ...
        if (pEdge->vertexIndex[0] == vi && pEdge->vertexIndex[1] == vj) {
            // Get index of the OTHER triangle of this edge
            int whichEdgeTri = (pEdge->triangleIndex[0] == triIndex);
            return pEdge->triangleIndex[whichEdgeTri];
        }
    }
    // The input edge was a boundary edge, not connected to any other triangle
    return -1;
}
```

这里，函数 **MarkAllNeighbors()**简单地实现为递归版本。在实际应用中，该函数一般采取显式堆栈处理方案：针对包含数千个数据面的网格而言，递归操作很难胜任这一计算过程（参见第 6 章的相

关内容，以便将递归操作转换为栈式操作）。堆栈的使用将有助于书写精练的代码，并为数据类型以及数据量提供了多种选择方案。

12.3　孔、缝隙、间隙以及 t-连接

网格中的孔可表示为由一组封闭边连接起来的顶点集合，且不存在连接上述顶点和边的相关数据面。不同于通常意义上的、尺寸较大的孔，这一类孔一般细小、狭长，多出现于连接数据面之间（如图 12.8 所示），也可称作缝隙（crack）。

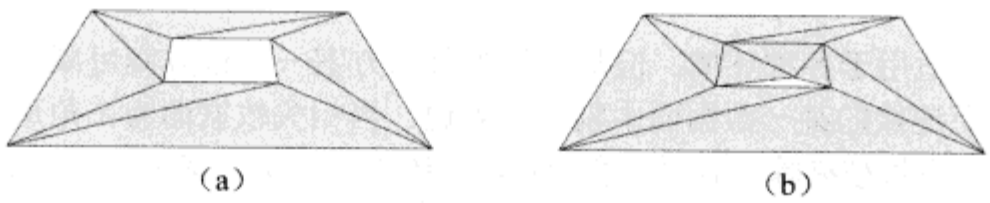


图 12.8　（a）孔状网格。（b）带有缝隙的网格（放大图）

缝隙有时也指两个部分连接的几何体之间的楔形狭缝。另外，两个完全分离的几何体间的狭长空间则称作间隙（gap）。缝隙可通过局部检测法加以处理，而消除间隙则只能依赖于全局检测法。

网格中的缝隙多源自（冗余）复制顶点。这一类缝隙可通过网格的顶点焊接操作得到有效解决。当某一边的顶点位于另一条边的内部时（该顶点导致直线出现少量偏差），这一现象则称作 t-连接（两条边呈现 T 型），该问题顶点则称作 t-顶点。亦即，t-顶点将包含于一个或多个多边形中，且错误地出现于多边形的某一边上。同时，t-连接也可能由间隙或缝隙的某一部分构成，这取决于相关几何体的形状。图 12.9 分别显示了上述 3 种情形：（非孔类）缝隙、间隙以及 t-连接。

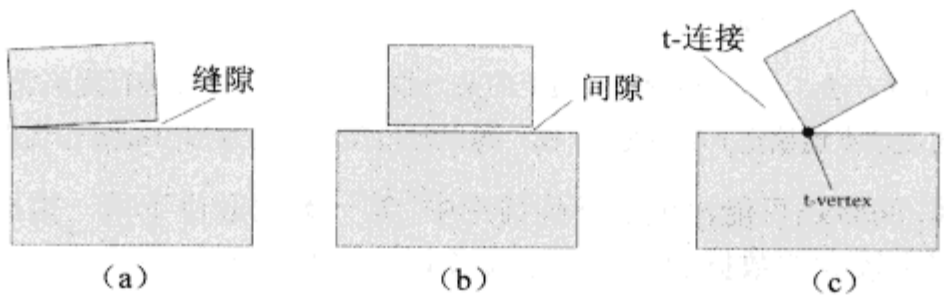


图 12.9　（a）（非孔）缝隙。（b）间隙。（c）t-连接（及其对应的 t-顶点）

存在多种方案可用于消除 t-连接。一种较为直接的方法是遍历网格中的全部顶点，且针对每一顶点两两地测试其关联边以考查它们的（近似）共线性。若存在两条共线边，则这两条边可视为缝隙的组成部分。此处，可以通过合并边顶点的方式消除缝隙。由于顶点可按照任意顺序加以访问，因而针对消除包含任意数量顶点的缝隙，须再次递归遍历两条合并边的端点，以确定缝隙边上的全部顶点。另外，如果所修复的网格为二维流形，则对于只包含一个关联数据面的边，上述操作有可能会受到限制。综上所述，该算法易于实现且能够正确地消除缝隙。但该算法却不能有效地解决间隙问题。根据定义，间隙问题无法通过局部检测法加以识别。

因此，正确地处理间隙问题需要采用更为复杂的全局检测法。给定一条边，算法须能够获取位于

该边之上或附近的全部网格顶点；另外，给定一个顶点，算法还应能够确定该顶点附近的全部边。在实际应用中，可将全部网格边插入至某一层次数据结构中，例如层次网格或松散 8 叉树，以实现网格顶点的快速查询。

对于包含间隙或缝隙的 t-连接，须采用相关方案对其加以剔除，且通常包括下列 3 种方案（如图 12.10 所示）：

- ☑ 压平顶点（顶点塌陷）。t-顶点在其对边上向任一端点偏移，最终与某一端点重叠。
- ☑ 分裂边。一种更加常见的方法是分割 t-顶点的对边，并将两个新增端点连接至 t-顶点——这将使连接至该对边的全部数据面的顶点数量加 1。
- ☑ 顶点捕捉。类似于边的分裂操作，该方案首先获取位于对边上的 t-顶点，随后按照上一方案对该边进行分割。

顶点塌陷操作并非适用于各种环境，但却常作为首选方案——该方案可减少一个顶点。另外，该方案还可以在 t-顶点所连接的某一数据面中剔除一条边。当相关数据面为三角形时，该三角形将会因退化现象而被合并且最终将被剔除。需要注意的是，上述 3 种解决方案都会生成非平面或非凸数据面，因而在后续操作中需要对这一类数据面执行三角剖分。

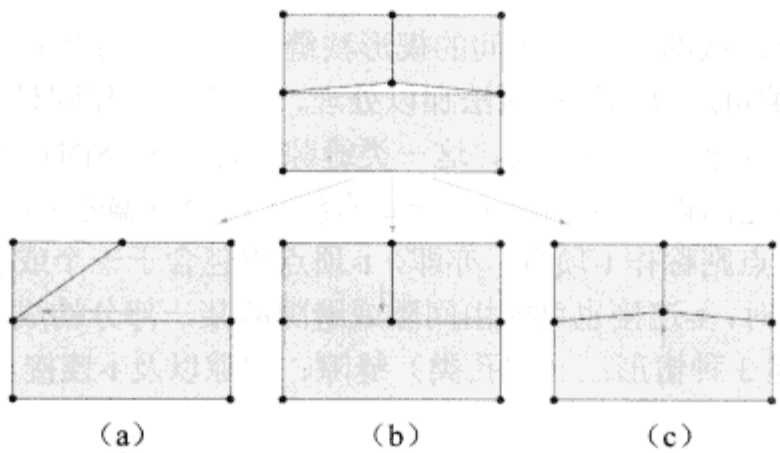


图 12.10　解决 t-连接的 3 种方法。（a）相对于对边上的某一临界顶点对 t-顶点进行塌陷操作。（b）将对边分裂为两条边，并将两个新增的边端点连接至 t-端点处。（c）捕捉对边顶点并通过边分裂操作实现此边上的顶点插入操作

对于非流形几何体，间隙还可能出现于下列情形中：某一数据面位于另一数据面的上方，如图 12.11 所示。这里，需要对图中较大的数据面实施分割操作以实现数据面间的连接。同样，可以使用边分裂操作时的同一层次数据结构对网格数据面（而非边）执行插入操作，以完成数据面的分裂操作。

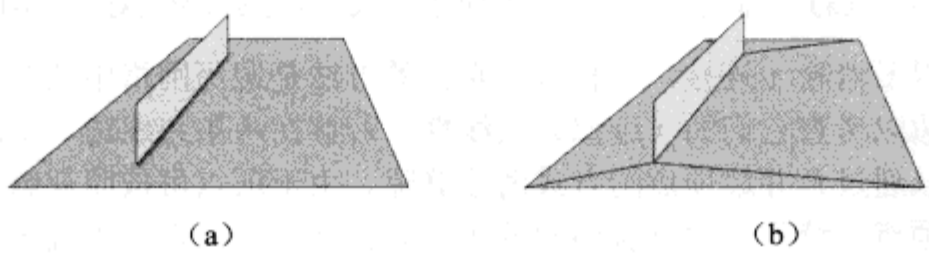


图 12.11　（a）某一数据面位于另一数据面的上方所形成的间隙。（b）通过数据面分裂操作可消除间隙

针对某些相交查询（如光线-网格测试），彼此贯穿的表面并不会对查询操作产生影响。此时，当边顶点位于另一数据面支撑面的负半空间中时，则无须执行数据面的分裂操作。

12.4　共面数据面的合并操作

边和数据面的分裂操作可能会引入额外的数据面。随着数据面数量的增加，碰撞检测的速度将会相应地减慢，因此，有必要实施某些优化步骤以生成健壮的碰撞几何体，从而降低数据面的数量。

曾有多种算法针对模型简化问题提出了相应的解决方案，[Luebke02]和[Puppo97]对此类问题进行了较为全面的回顾。针对碰撞检测，须保证几何体的形状相对于体空间不会出现收缩，却允许对象间在视觉上互相贯穿，即使它们在碰撞检测环境下未真正地相交。针对模型简化算法库，一类特别适用于碰撞检测的机会算法是合并（近似）共面数据面。数据面的合并操作还有助于剔除条状数据面以及退化数据面，这一类数据面通常会影响计算的健壮性（参见第 11 章）。

两个共面数据面除了要满足某一容值误差之外，还应符合其他一些合并规则，如下所示：

- ☑ 二者皆为双面数据面或单面数据面。针对单面数据面，应具有同一主方向。
- ☑ 具有相同的关联表面特征（如脚步音效、物理摩擦参数等）。
- ☑ 具有一条或多条公共（边界）边。

在后续计算过程中，假设进行合并操作的数据面均遵循上述规则。

考虑到最终结果很可能为三角形或四边形网格，因而，一种可选方案是：一旦结果数据面为三角形或四边形，即对其实施合并操作。不难看出，算法较为直观且易于实现。这里，为了生成较为优化的结果，还需要考查并行合并操作过程中的数据面群集问题。在图 12.12 所示的螺旋形网格中，9 个数据面形成了一个凸面，但并不存在进一步的合并操作。因此，有必要测试最坏情况下数据面的组合数量——但这并不具备实际操作意义。

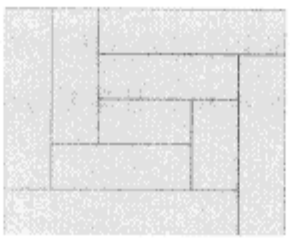


图 12.12　若仅考查两个（或多个）数据面的合并操作且最终的结果数据面为凸面，则螺旋形网格无法实现相应的合并操作；若在合并过程中允许出现凹面，则该网格可合并为一个（四边形）数据面

较为常见的算法是将全部共面邻接数据面合并为一个（凹）多边形，随后对其进行三角剖分或将其分解为计算结果所要求的凸面片类型。该算法通常能够产生较为理想的结果。

若相关邻接信息已知，则共面邻接数据面的合并操作可简单地实现为：根据所提供的邻接连接访问相应的邻接数据面。例如，当给定边-面表后，合并处理过程则可视为一类基于边数的线性操作，这可通过遍历相关边数据并合并该边的关联数据面（若共面）加以实现。若邻接数据当前未知，此处将不再使用上述基于邻接信息的合并操作，转而采取一种类似于环境立方体映射的方案来计算相应的共面数据面（参见[Akenine-Möller02]）。

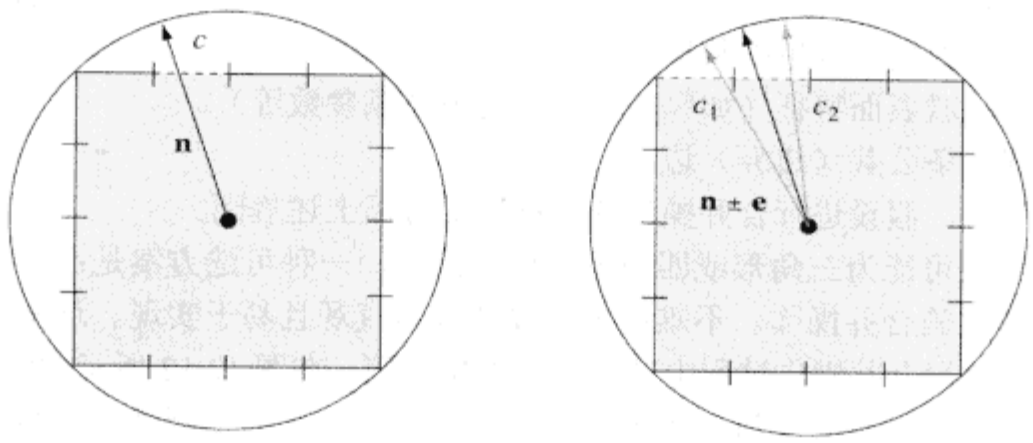
各多边形法线均可视为描述单位球表面的定位数据。这里定义一个单位球的内切立方体，且每一个立方体面均被划分为等尺寸的网格单元。如图 12.13（a）所示，多边形法线相交于某一立方体数据

面网格单元。随后，令多边形与其法线所相交的数据面网格单元关联。

准共面多边形多数都将会映射至同一网格单元中，待其共面性得以验证后，即可进入后续处理过程。但考虑到网格单元内部的数值离散性，两条彼此靠近的面法线仍会指向不同的网格单元。一种简单的解决方案是：测试与多边形法线相交的网格单元的全部 8 个邻接数据面网格单元。这里，应谨慎处理立方体的边区域，以确保获取正确的网格单元。

为了减少邻接网格单元的测试数量，可在相交数据面的平面内对多边形法线 n 加以扰动，这将在该数据面内得到一个矩形区域，并根据与 n 共面的法线对该区域加以表示。因此，可对与矩形区域相交的网格单元进行测试，如图 12.13（b）所示。一般情况下，与该矩形区域相交的网格单元的数量将会小于 8。若矩形区域扩展至立方体中的某一条边上时，需确保获取正确的网格单元。

上述算法的复杂度为 $O(n)$。[Salesin92]中给出了另一个复杂度为 $O(n \log n)$的树排序算法。另外，[Hinker93]和[Kalvin96]也提供了合并（准）共面数据面的解决方案。



（a）　　（b）

图 12.13　（a）法线 **n** 击中单位球体中内切立方体上方数据面得网格单元 c（虚线表示）。（b）扰动法线 **n** ± **e** 击中网格单元 c_1 和 c_2 且需要与法线 **n** 表示的多边形进行共面测试

12.4.1　测试多边形的共面性

两个多边形的共面测试可通过计算其法线间的角度加以实现。如果该角度小于某一给定容值，则两个多边形可视为共面且可执行合并操作。

该方法适用于多种应用场合，例如模型的简化计算（[Hinker93]）。但针对碰撞检测过程，该算法则存在先天缺陷。这里，考查多边形 A_1 和 B_1 且二者间的夹角为 θ，构造厚度为 d_1 的表示平面以包含全部顶点，如图 12.14 所示。同时，适当地放大多边形且表示为 A_2 和 B_2。显然，在放大过程中，夹角 θ 保持恒定，而表示平面厚度增至 d_2。

不难看出，多边形法线间的夹角仅是一类相对共面性测试。针对绝对误差测试，还应考虑其表示平面的厚度。因此，一类较好的解决方案是：执行两个多边形的合并操作，并对结果多边形执行共面测试（参见 12.4.2 节）。该方案可扩展为任意数量的多边形合并操作。

在碰撞检测系统中对合并操作实施管理时，需采用相对共面测试方案，且合并后的多边形表示面应能够捕捉到该多边形的相关顶点，这将对基于该表示面的多边形顶点进行有效的限制，对于健壮的

相交测试而言，这将使距离误差控制在多边形厚度所表示的容值范围内。

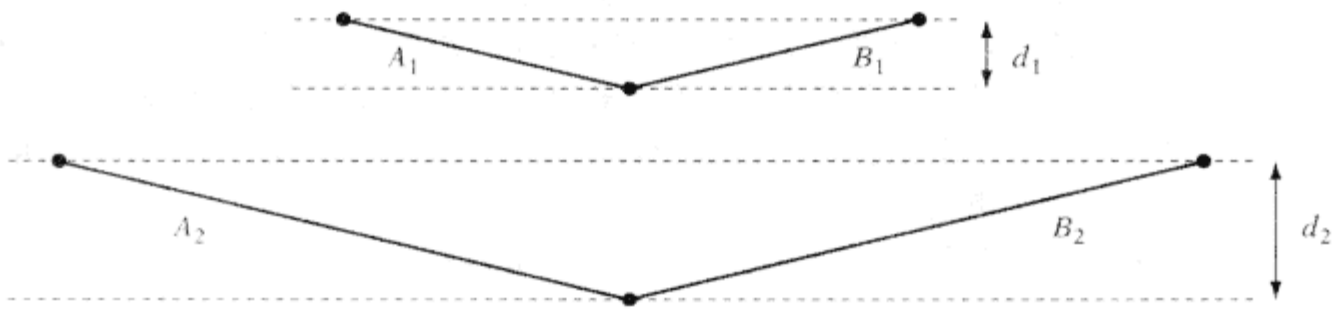


图 12.14　测试两多边形平面法线间的夹角称之为相对共面测试，且不受多边形的缩放影响；计算包含多边形全部顶点的最佳匹配平面的厚度值则称之为多边形绝对共面测试

然而，当采用焊接顶点对多边形的顶点实施平面捕捉时，针对与多边形共享公共顶点的数据面，这将不可避免地引入非共面误差。另外，对于当前对象所涉及的全部顶点，这还会产生一系列的连锁反应。一种方法是：针对超出容值的非共面数据面，进行有效的是三角剖分计算。但与合并操作所剔除的数据面数量相比，三角剖分计算无法确定数据面的数量最终是否会减少。

更糟糕的是，如果顶点未加焊接且在合并过程中产生移动，除非绝对合并操作的容值小于平面厚度的容值，否则邻接数据面将会在边附近出现缝隙。图 12.15 显示了合并容值大于平面厚度容值时，数据面与合并后的数据面之间的缝隙产生过程。为了避免容值问题对健壮的计算过程产生影响，并确保合并操作后相关数据面数量的减少，在碰撞检测系统中，应采用绝对误差测试以实现几何体的合并操作。

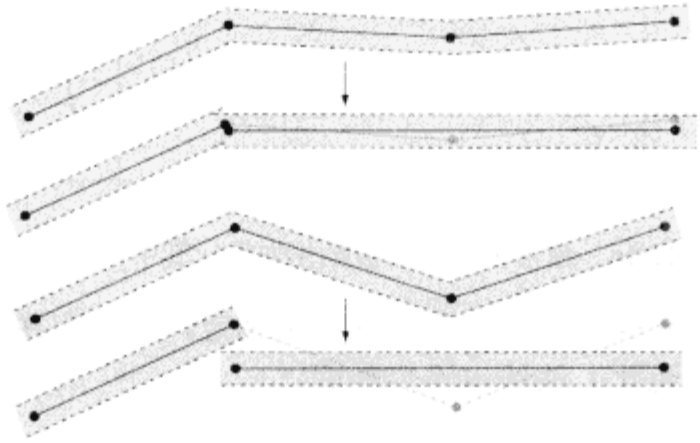


图 12.15　上图显示了绝对误差容值小于平面厚度容值（采用灰色表示）时如何消除邻接数据面之间的缝隙；下图显示了当合并容值大于平面后的容值时缝隙的产生过程

12.4.2　多边形的共面测试

多边形通常定义为平面内的一组点集。多边形的共面测试应确保全部顶点位于该平面内。这一过程可通过下列方式加以实现：计算多边形支撑面的平面方程，测试全部顶点以确保其位于平面的某一容值范围内。平面方程则可通过多边形法线及其内部参考点得到。

上述问题叙述起来较为简单，但需要考虑下列问题——如何计算多边形法线 $\mathbf{n} = (n_x, n_y, n_z)$。一种方案是计算多边形两条邻接边的叉积。但该方案由于边的（准）共线问题将导致叉积的结果为 0 向量（从

而使计算过程产生较大的消除误差）。

即使两边之间的夹角足够大，该方案仍然存在一个问题。这里，除了两条边的公共顶点 V 之外，假设所有顶点皆位于平面之上，且令顶点 V 偏离当前平面某一距离 d。若在当前平面内对多边形进行适当的放大，相对于该平面，顶点 V 仍将保持原有距离 d 且多边形的绝对共面测试结果将不会受到影响。但由于两条边的叉积将被用作其表示平面的法线，因此，随着多边形被放大，除了与两边重合的 3 个顶点之外，其余顶点都将远离当前平面。

法线的计算过程将涉及全部顶点。一类常见的算法是计算各顶点 V_i 的法线 $\mathbf{n}_i$（顶点关联边的叉积），随后对其计算均值，如下所示：

$$\mathbf{n}=\frac{1}{n}\sum_{0\leqslant i<n}\mathbf{n}_i\text{，其中 }\mathbf{n}_i=(V_{i+1}-V_i)\times(V_{i-1}-V_i)$$

这里，假设 $V_n=V_0$，$V_{-1}=V_{n-1}$。若不考虑法线的长度，则可对各法线求和而非进行均值计算。

虽然该方案针对凸多边形工作良好，但在凹多边形环境下仍存在某些缺陷。其中，相对于凸顶点，凹顶点处的法线将指向相反方向。例如，考查图 12.16 所示的星形多边形，偶数顶点处的法线将指向纸面以外，而奇数顶点处的法线将指向纸面以里。

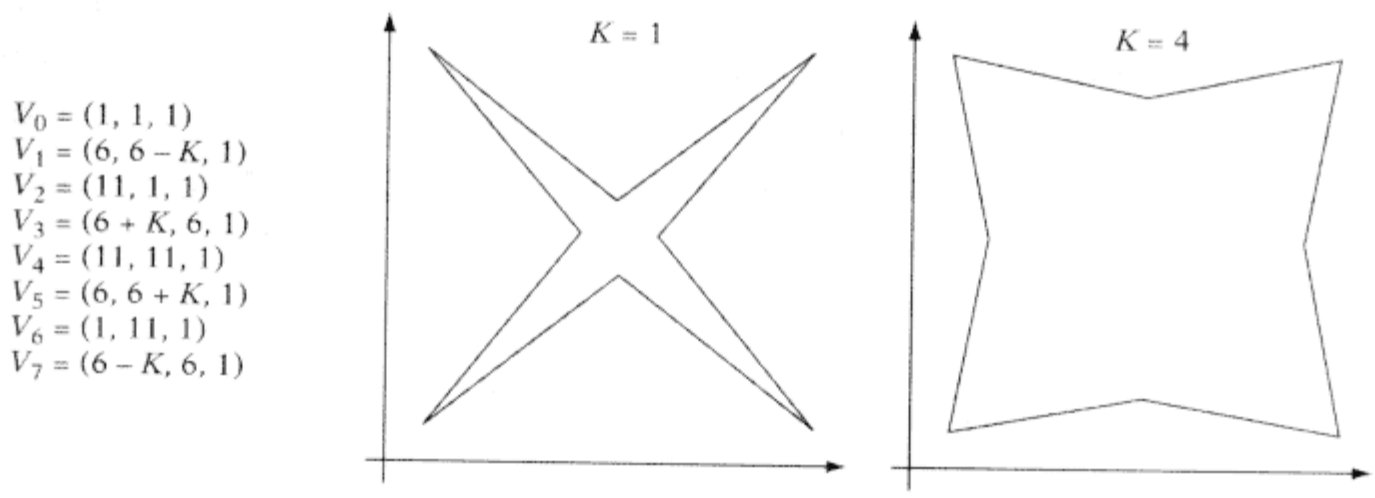


图 12.16　星形多边形的参数化方案。其中，$0<K<5$

当求和计算包含反方向的法线时，其结果将变得不确定，这将取决于相关的输入值。如下所示，针对不同的 K 值，相应的（非范化）法线也不尽相同：

K	$\mathbf{n}$
1	(0,0,−124)
2	(0,0,−56)
3	(0,0,4)
4	(0,0,56)

其中，法线在 K=2、K=3 时方向各异。实际上，在 K 约为 2.928932 时法线即变为 0 向量，从而导致算法崩溃。

一种较为健壮的法线算法是 Newell 算法（[Tampieri92]）。Newell 算法基于下列事实：多边形法线分量正比于其 yz，xz 以及 xy 平面的有符号投影面积。因此，法线 $\mathbf{n}$ 可按如下方式计算：

$$n_x = \sum_{0 \le i < n} (V_{i,y} - V_{i+1,y})(V_{i,z} + V_{i+1,z})$$

$$n_y = \sum_{0 \le i < n} (V_{i,z} - V_{i+1,z})(V_{i,x} + V_{i+1,x})$$

$$n_z = \sum_{0 \le i < n} (V_{i,x} - V_{i+1,x})(V_{i,y} + V_{i+1,y})$$

其中，求和结果对应于 2 倍的梯形有符号面积。在各个主平面内，梯形可由当前多边形边及其在对应轴上的投影加以表示。在上式中，第一项表示梯形的宽度；第二项则 2 倍于该梯形的高度。通过重组各项数据，Newell 算法也可通过求和的方式计算法线，如下所示：

$$\mathbf{n} = \sum_{0 \le i < n} \mathbf{m}_i \text{，其中 } \mathbf{m}_i = V_i \times V_{i+1}$$

针对上述星形多边形，Newell 算法将生成方向一致的法线（均指向纸面外），如下所示：

K	**n**
1	(0,0,40)
2	(0,0,80)
3	(0,0,120)
4	(0,0,160)

下列代码显示了健壮的多边形表示面的实现过程。其中，在计算平面方程时，多边形质心（全部顶点的平均值）将用于表示平面内的参考点。

```
// Given n-gon specified by points v[], compute a good representative plane p
void NewellPlane(int n, Point v[], Plane *p)
{
    // Compute normal as being proportional to projected areas of polygon onto the yz,
    // xz, and xy planes. Also compute centroid as representative point on the plane
    Vector centroid(0.0f, 0.0f, 0.0f), normal(0.0f, 0.0f, 0.0f);
    for (int i = n - 1, j = 0; j < n; i = j, j++) {
        normal.x += (v[i].y - v[j].y) * (v[i].z + v[j].z);      // projection on yz
        normal.y += (v[i].z - v[j].z) * (v[i].x + v[j].x);      // projection on xz
        normal.z += (v[i].x - v[j].x) * (v[i].y + v[j].y);      // projection on xy
        centroid += v[j];
    }
    // Normalize normal and fill in the plane equation fields
    p->n = Normalize(normal);
    p->d = Dot(centroid, p->n) / n;                             // "centroid / n" is the true centroid point
}
```

Newell 算法可视为一类采用全顶点的、计算“最佳匹配”平面的解决方案。另一种方法是采用最小二乘匹配法实现多边形顶点的平面匹配（详细内容可参见[Eberly01]）。

利用 Newell 算法计算平面方程的开销显然要大于两边的叉积计算。当然，存储平面方程可作为一种选择方案，但这将占用大量的内存空间，从而使该算法丧失了存储优势。这里，一种妥协方案是在运行期内利用叉积计算法线和平面方程，且待当前多边形确认为输出后，前 3 个存储顶点即为所求。另外，针对特殊多边形，还可对 Newell 算法进行适当的简化。例如，对于四边形 *ABCD*，其 Newell 法线可简化为计算对角线 *AC* 和 *DB* 的叉积，如下所示：

$$2(AC \times DB) = (AB \times AD) + (BC \times BA) + (CD \times CB) + (DA \times DC)$$

其中，表达式右侧对应着各顶点的叉积法线和。有趣的是，四边形法线的计算开销反而小于三角形。

待多边形的表示面计算完毕后，需要将多边形顶点代入平面方程以考查各顶点相对于当前平面的偏移距离，从而确定该多边形的共面性。若全部顶点皆位于平面的容值距离内，则该多边形则视为共面，其实现方法如下所示：

```
// Test if n-gon specified by vertices v[] is planar
int IsPlanar(int n, Point v[])
{
    // Compute a representative plane for the polygon
    Plane p;
    NewellPlane(n, v, &p);
    // Test each vertex to see if it is farther from plane than allowed max distance
    for (int i = 0; i < n; i++) {
        float dist = Dot(p.n, v[i]) - p.d;
        if (Abs(dist) > PLANARITY_EPSILON) return 0;
    }
    // All points passed distance test, so polygon is considered planar
    return 1;
}
```

[Sunday02]给出了相关方法以减少 Newell 法线和多边形面积的计算量。

12.5 三角形剖分和凸划分

虽然可直接对合并数据面执行碰撞检测，但测试所涉及的数据面通常是一类简单、快速以及较为健壮的凸面数据。另外，针对非凸数据面，则有必要将其分解为一系列凸面片，该过程通常称作凸划分。一种较为简单的方法是对非凸数据面进行三角剖分——将相关数据面分解为一组三角形。另一种方案是将数据面划分为尽可能少的凸面片，以降低测试数据面的数量。

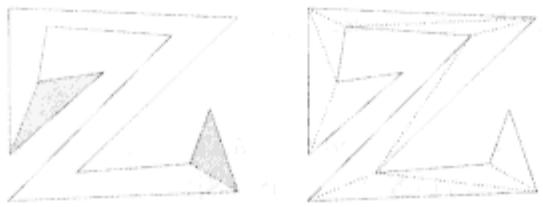
下面将详细讨论两种算法：非凸数据面的三角剖分算法和非凸数据面-凸面片剖分算法。最后，还将阐述如何将非凸多面体划分为凸面片这一问题。

12.5.1 耳式剪裁实现三角剖分

多种算法可实现多边形的三角剖分计算，其中包括某些易于实现的慢速算法以及实际应用中难以实现的、理论上的最佳方案。一种较为常见的直观算法是割耳式（耳式剪裁）算法。

对于由多边形顶点 V_i、前驱顶点 V_{i-1} 以及后继顶点 V_{i+1} 构成的三角形，如果 V_{i-1} 和 V_{i+1} 之间的对角线完全包含于当前多边形，则该三角形称为“耳”，且顶点 V_i 称为“耳尖（ear tip）”顶点。因此，一般可通过耳尖顶点定位该耳式三角形。另外，任何一个（非三角形）多边形都至少包含两个互不交叠的耳式三角形（[Meisters75]）。因此，可通过相应的确认-删除（切除）过程实现多边形的三角剖分计

算。多边形的三角剖分结果将会生成 n-2 个三角形。图 12.17 显示了一个包含两个耳三角形的多边形及其对应的三角剖分方案。



（a）　　（b）

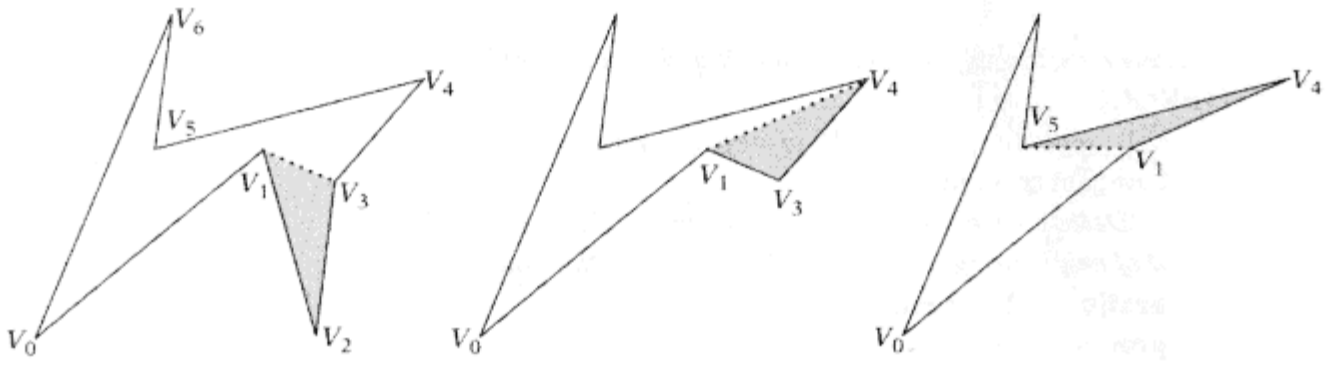
图 12.17　（a）一个包含两个耳三角形的多边形对应的三角剖分方案（其中，耳三角形采用灰色表示）。（b）与该多边形对应的三角剖分方案之一

针对较为简单的多边形，相对于 3 个连续顶点 V_{i-1}、V_i 和 V_{i+1} 所构成的耳三角形 E，其等价测试须确保顶点 V_i 为凸顶点，且 E 中不包含其他多边形顶点。这里，若 E 具有与多边形相同的环绕方向，则 V_i 为凸顶点。例如，对于一个顶点逆时针排列的多边形，若 E 也为逆时针排列，则 V_i 为凸顶点。

下面将通过割耳式算法考查三角形的剖分过程。初始状态下，多边形将包含 n 个耳式三角形，$n \geq 2$。待某一耳式三角形被剔除后，n 值可能会增加、减少抑或保持不变。n 值的变化取决于割除后的顶点 V_{i-1} 和 V_{i+1} 的耳尖状态。亦即，相应的状态变化仅发生在被割除的耳式三角形顶点中。

通过逆时针遍历多边形的边界，可实现多边形的三角剖分——依次连续地访问各顶点并执行耳式测试。待发现耳式顶点后，即可执行剔除操作。为了保证被剔除的耳式三角形中的两个顶点可被再次访问，可将（被移除的）耳尖顶点的前驱顶点设置为待访问的下一个顶点。

图 12.18 显示了采用该算法进行三角剖分时的第一个步骤。算法始于顶点 V_0，在耳式测试过程中，顶点 V_5 位于耳式三角形内，因此顶点 V_0 不是耳式顶点，且算法继续处理顶点 V_1。由于顶点 V_1 为非凸顶点，三角形(V_0,V_1,V_2)为顺时针排列，因此顶点 V_1 也不是耳式顶点，且算法继续处理顶点 V_2。由于 V_2 为凸顶点，且三角形(V_1, V_2, V_3)不包含其他多边形顶点，因此顶点 V_2 为耳式顶点并被剪裁掉。此时，顶点 V_1 的耳式状态可能会发生改变，因此算法将回溯至顶点 V_1 并再次对其加以访问。这里，顶点 V_1 仍为凹顶点，则算法将转而处理后续顶点 V_3——该顶点经测试为耳式顶点且被剪裁，则算法将再次访问顶点 V_1。这里，顶点 V_1 为凹顶点。随后，处理顶点 V_4 且为耳式顶点，则再次访问顶点 V_1，其结果为：顶点 V_1 为耳式顶点。待顶点 V_1 被剪裁后，剩余部分显示为一个三角形且算法结束。



（a）　　（b）　　（c）

图 12.18　割耳式三角剖分的相关步骤。（a）确认 V_2 为耳尖顶点。（b）待切除 V_2 后确认 V_3 为耳尖顶点。（c）待切除 V_3 后确认 V_4 为耳尖顶点

单一顶点的耳式测试需执行其他所有顶点的包含测试，因而其复杂度为 $O(n)$。同时，该算法的全部操作复杂度为 $O(n^2)$。下列代码片段实现了三角剖分过程：

```
// Triangulate the CCW n-gon specified by the vertices v[]
void Triangulate(Point v[], int n)
{
    // Set up previous and next links to effectively form a double-linked vertex list
    int prev[MAX_VERTICES], next[MAX_VERTICES];
    for (int i = 0; i < n; i++) {
        prev[i] = i - 1;
        next[i] = i + 1;
    }
    prev[0] = n - 1;
    next[n - 1] = 0;

    // Start at vertex 0
    int i = 0;
    // Keep removing vertices until just a triangle left
    while (n > 3) {
        // Test if current vertex, v[i], is an ear
        int isEar = 1;
        // An ear must be convex (here counterclockwise)
        if (TriangleIsCCW(v[prev[i]], v[i], v[next[i]])) {
            // Loop over all vertices not part of the tentative ear
            int k = next[next[i]];
            do {
                // If vertex k is inside the ear triangle, then this is not an ear
                if (TestPointTriangle(v[k], v[prev[i]], v[i], v[next[i]])) {
                    isEar = 0;
                    break;
                }
                k = next[k];
            } while (k != prev[i]);
        } else {
            // The 'ear' triangle is clockwise so v[i] is not an ear
            isEar = 0;
        }
        // If current vertex v[i] is an ear, delete it and visit the previous vertex
        if (isEar) {
            // Triangle (v[i], v[prev[i]], v[next[i]]) is an ear
            ...output triangle here...
            // 'Delete' vertex v[i] by redirecting next and previous links
            // of neighboring verts past it. Decrement vertex count
            next[prev[i]] = next[i];
            prev[next[i]] = prev[i];
            n--;
            // Visit the previous vertex next
            i = prev[i];
        } else {
```

```
                // Current vertex is not an ear; visit the next vertex
                i = next[i];
            }
        }
}
```

在某些情况下，算法应能够支持带孔的多边形三角剖分计算。相关内容可参见 12.5.1.1 节。

12.5.1.1　孔多边形的三角剖分

为了能够对孔多边形实施有效的三角剖分，首先需要建立正确的表达形式。常见的表示方法为：逆时针的外部多边形边界附带一个或多个顺时针的孔边界。图 12.19 中的左图显示了一个孔多边形。

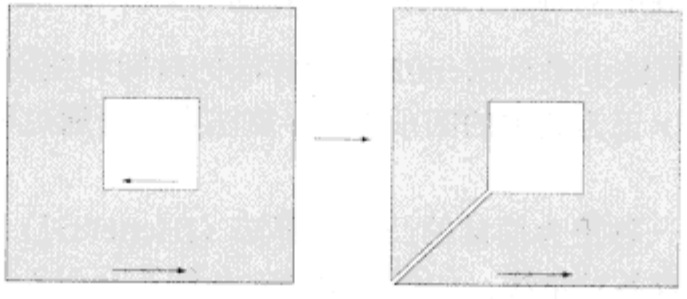


图 12.19　从外部边界（CCW）向（CW）孔边界采用一个 0 宽度通道实现分割操作以处理孔
（为了清晰地显示通道，其端点稍稍分开）

为了确保能够正确地实现多边形的三角剖分，相应的外部、内部边界必须合并为一条连续的边链——可从外部边界向内部边界分割出一条 0 宽度的“通道”以实现这一步骤。此处，该通道将分别在内、外边界某一顶点处打破原有链接，并采用两条反向边连接这两个顶点。此时，外边界和孔边界将构成一个逆时针的连续多边形边界。图 12.19 显示了一个孔多边形的处理步骤。另外，对于包含多个孔的多边形，可针对每一个孔采用上述步骤。

有时，会出现边界间方向各异这一现象，且外边界与孔边界之间定位模糊。在这种情况下，具有最大面积的多边形可定义为外部边界，其余边界即为孔。待外部边界确定后，如有必要，可通过相反的顶点顺序分别指定外边界和孔——逆时针为外边界，顺时针为孔。

存在多种三角剖分的程序代码并可通过网络得到。较为健壮的三角剖分程序实现应首推 Jonathan Shewchuk 算法，其具体实现可参见[Shewchuk96b]。另外，[Held01]中也探讨了健壮的三角剖分算法的实现过程。

12.5.2　多边形的凸剖分

如第 5 章所述，光线与四边形的相交测试其计算开销并不大于三角形。因此，碰撞几何体可涵盖三角形、四边形，甚至是任意形状的凸多边形，并且只需增加少许复杂度。例如对多边形进行排序，即可实现加速计算。

虽然简单多边形的三角剖分叙述起来较为直观，但将其剖分至某一多边形集仍需要较为复杂的计算过程，这还不包括优化过程。实际上，该剖分过程需要引入额外的顶点，称作 Steiner 顶点。图 12.20 分别显示了采用对角线划分时 5 个多边形剖分为 3 个凸面片的计算过程，以及引入 Steiner 顶点后 4 个多边形剖分为两个凸面片的计算过程。

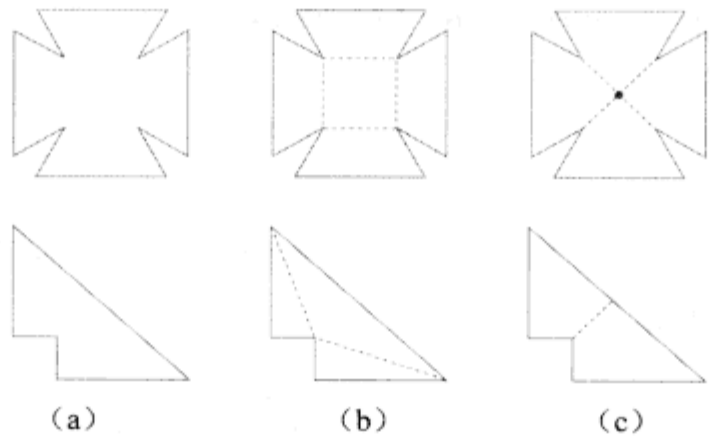


图 12.20　(a) 原多边形。(b) 不使用 Steiner 顶点时的优化凸剖分方案。(c) 使用 Steiner 顶点时的优化凸剖分方案

简单多边形的最小凸剖分算法可参见[Keil02]，该算法的时间和空间复杂度为 $O(n+r^2\min(r^2, n)) = O(n^3)$。这里，$n$ 表示为顶点数，r 记为其中的凹顶点数。算法实现可通过网络得到，另外，[Schneider02]中也给出了该算法的伪代码。

Hertel 和 Mehlhorn 提出了一种更为简洁的凸剖分算法（[Hertel83]）。首先，该算法基于简单多边形的三角剖分方案，并逐一考查各对角线。如果移除该对角线不会引入凹顶点，则移除当前对角线。

否则，继续处理下一条对角线，直至全部对角线均以检测完毕。因而，其复杂度为 $O(n)$。考虑到三角剖分处理过程，算法的全部复杂度为 $O(n^2)$。图 12.21 显示了采用 Hertel-Mehlhorn 算法的凸剖分示例。相关代码如下所示：

```
Mesh HertelMehlhorn(Polygon p)
{
    // Produce a triangulated mesh from the original polygon
    Mesh m = TriangulatePolygon(p);
    // Loop over all diagonals of the triangulated polygon
    int numDiagonals = GetNumberOfDiagonals(m);
    for (int i = 0; i < numDiagonals; i++) {
        // Test if the i-th diagonal can be removed without creating
        // a concave vertex; if so, remove the diagonal from the mesh
        if (DiagonalCanBeRemoved(m, i))
            RemoveDiagonal(m, i);
        }
    // The mesh is now a convex decomposition of the polygon. Return it
    return m;
}
```

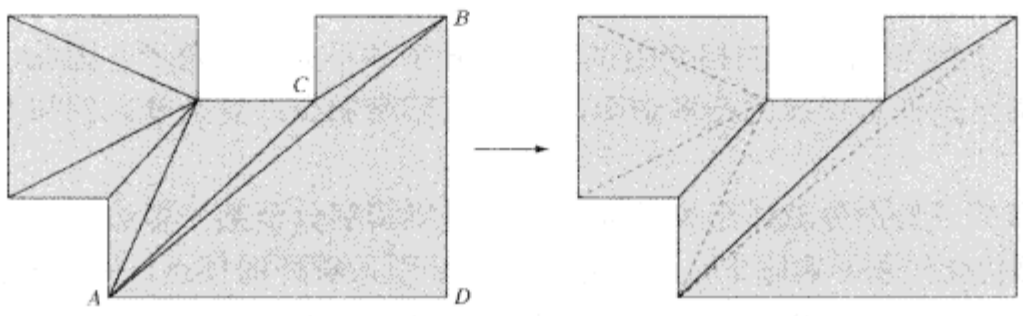


图 12.21　Hertel-Mehlhorn 算法通过尽可能多地消除对角线以将三角剖分转换为凸剖分

这里，无须采取单独的对角线删除步骤，该删除操作已合并于当前三角剖分计算中。由于耳式三角形已被剪裁，因而可对先前的已切割多边形执行测试。另外，如果将切割的三角形与原多边形进行合并操作后最终将生成一个凸多边形，则可将其视为合法的合并操作。否则，输出原多边形，且已切割的耳式多边形记为当前的“原多边形”。通过测试三角形-原多边形中的各顶点，其复杂度可达到常量级。

不难看出，对于不包含冗余边的凸划分，其结果最多 4 倍于最小凸面片数量。一般情况下，在实际应用中，Hertel-Mehlhorn 算法的划分结果将接近于最优结果。

通常，对三角剖分算法稍加改动即可得到较好的划分结果。例如，对角线应至少连接一个凹顶点，而不选择连接两个凸顶点的对角线。如图 12.21 所示，可采用对角线 *CD* 替代 *AB*，这将使凸剖分结果生成 3 个凸面片而非 4 个。

12.5.3　多面体的凸剖分

正如凸多边形可实现快速的相交测试，凸多面体也适用于高效的相交计算。因此，非凸多面体也需要剖分为相应的凸多面体，进而可采用第 9 章所描述的相关算法进行碰撞检测。

总体而言，多面体的剖分过程较为复杂：某些对象无法生成较好的剖分结果或根本无法剖分。例如，碗形结构的内部表面皆为凹面且无法实现相应的剖分。同样，圆环结构也会出现类似的情况。

首先，可以采用与三角剖分类似的方案对多面体进行操作，称之为四面体剖分——将多面体划分为相应的四面体。虽然多边形皆可实现相应的三角剖分，但某些多面体无法实现四面体剖分。其中，较为典型的例子是 Schönhardt 多面体。该多面体源于三棱柱结构且在两三角形之间加以扭曲，则三棱柱中的 3 个矩形面均被“褶皱”为具有一条公共凹边的两个三角形（如图 12.22 所示）。考查该多面体上的 4 个顶点时，其中两个顶点必定位于某一凹边的两侧。由于不存在构造四面体的 4 个相关顶点，因此无法实现四面体剖分。

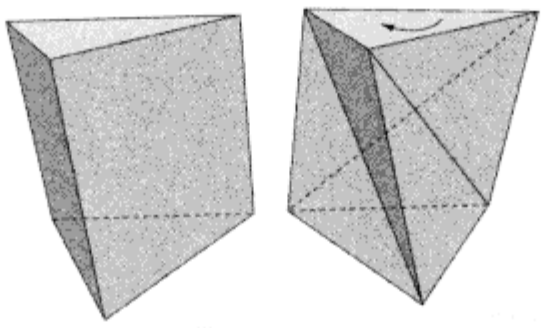


图 12.22　在三棱柱中的两个三角形之间进行扭曲操作后的 Schönhardt 多面体。在对该三棱柱的垂直面进行三角剖分时，将生成 3 条对角线凹边

然而，可在 Schönhardt 多面体内部添加 Steiner 顶点以实现其四面体剖分。另外，具有 n 个顶点的多面体一般需要 $O(n^2)$级的 Steiner 顶点以实现四面体剖分（如 Chazelle 多面体）。另外，若不使用附加的 Steiner 顶点，确定多面体的四面体剖分可行性则属于 NP-完全问题。在实际应用中，这意味着与 Hertel-Mehlhorn 算法等价的 3D 多面体剖分方案是不可行的。需要注意的是，凸多面体通常可实现相应的四面体剖分。确定凸多面体的最小四面体剖分操作则属于 NP-hard 问题。

一种较为有效的方法是利用穿过一条或多条凹边的分割面对多面体实施递归划分。每一次划分操

作将减少一条凹边，算法结束后将返回一个凸体集（且无凹边）。

若划分目标是生成尽可能少的凸四面体，则可采用同一分割方式尽可能地处理更多的凹边，这也意味着将减少所需的分割次数以生成较少的凸体数量。图 12.23 显示了采用两条凹边的划分过程。

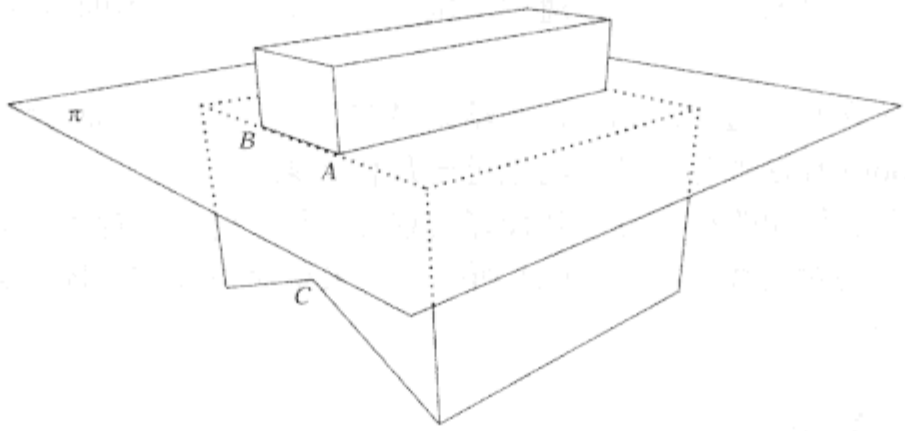


图 12.23　一种较好的启发式方法是利用一次分割处理两条或多条凹边。
这里，除凹边 C 之外，分割面 π 穿越凹边 A 和 B

若只可处理一条凹边，则仅保留一个自由度即可确定分割面——相对于凹边的旋转角。此时，可选择另一条（凸）边以构造一个分割面，或者选择与该边关联的某一数据面的支撑面作为分割面，这两种方案如图 12.24 所示。由于未必存在另一条边可与凹边形成平面，因此方案一的可行性并不确定。方案二则无须搜索相应边即可执行相关计算。然而，与多边形数据面对齐的分割面往往是（健壮性）问题的根源，这类似于 11.3.2 节中所描述的直线-平面相交问题。

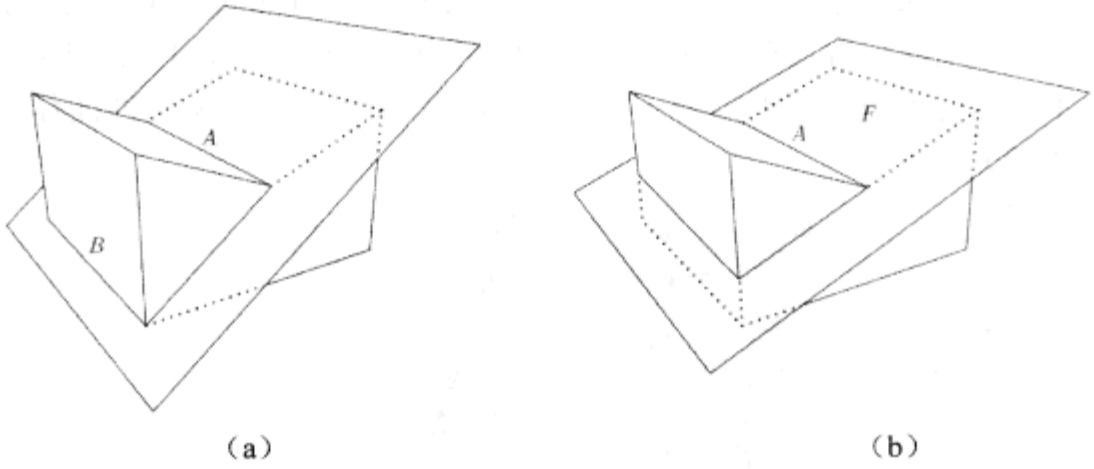


图 12.24　针对凹边 A 的两种解决方案。（a）选择穿越 A 和另一边 B 的平面进行分割。
（b）选择 A 的邻接数据面 F 作为支撑面进行分割

非凸多面体的平面分割结果往往会生成多个多面体，而非两个。随后，剖分过程将对每一个多面体进行递归划分。这里，为了限制剖分数量，往往只对多面体进行局部剖分，如图 12.25 所示。其中，多面体中标记为 A 的部分将被分割面切割。另外，若分割面穿越整个多面体，则标记为 B 的部分也将被切割。

跨越分割面的多边形将形成两个分离的轮廓线，且分别局部于 A 和 B。可将分割操作限制在与原凹边连通的轮廓线处，以获取正确的分割操作（即只对 A 进行分割操作）。待分割完毕后，位于分割面正面和背面的几何体均出现轮廓线形状的孔，且需要进行必要的填充操作。通过构造轮廓线形状的

多边形并针对几何体的其余部分实施必要的连接操作，相应的孔最终将被填充。

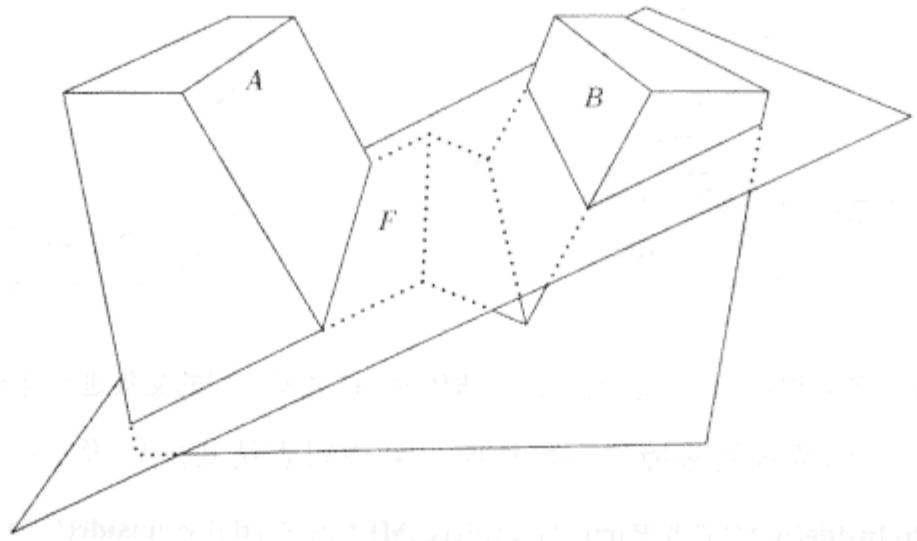


图 12.25　对 A 进行切割（这里，选择数据面 F 的支撑面 π 作为分割面，并与所选凹边邻接）将会影响到 B

可利用下列方法计算凹边：令 F_1 和 F_2 分别表示两个数据面且交于边 E。这里，如果 F_2 中的某一顶点（且不在边 E 上）位于 F_1 支撑面的正面，则边 E 为凹边。特别地，对于图 12.26 中的两个三角形 ABC 和 ADB，如果 $\mathbf{n} \cdot (D - A) > 0$ 且 $\mathbf{n} = (B - A) \times (C - A)$，则边 AB 为凹边。

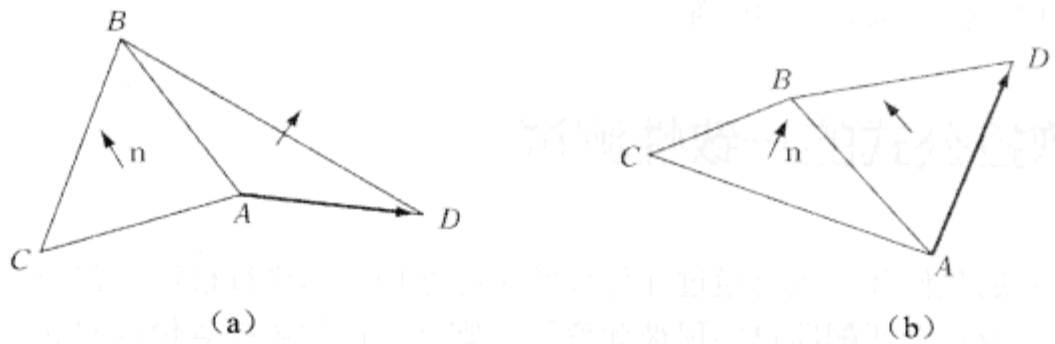


图 12.26　（a）$\mathbf{n} \cdot (D - A) < 0$，因而边 AB 视为凸边。（b）$\mathbf{n} \cdot (D - A) > 0$，因而边 AB 视为凹边

[Belov02]中也给出了凸剖分的相关算法。另外，[Yong00]和[Ribelles01]中均探讨了分割操作，[Ribelles01]中还进一步讨论了孔填充操作。最后，采用 BSP 树结构的凸剖分的相关内容可参见[Joshi03]。

12.5.4　不可剖分的凹几何体

如 12.5.3 节所述，并非所有的凹几何体都可剖分为凸体。此处，考查一个半球形碗状结构并采取多边形建模。由于碗形内部皆为凹面，因此剖分操作需要分别对各数据面进行分割。一种方法是将碗形结构视为多边形汤，并作为一个数据面集执行边界几何体的碰撞检测。因此，该方法可使用任何一种碰撞算法加以解决。但其缺陷在于：算法将丢失体空间信息。这也意味着，将很难检测到某一对象是否完全位于该碗形几何体结构的内部。

另一种方法是将碗形结构作为一类 CSG 对象加以建模。例如，可按如下方式对碗形结构 D 实现 CSG 表达：轴对齐包围盒 A 与两球体 B、C 之差的交集，即 $D = A \cap (B - C)$，如图 12.27 所示。

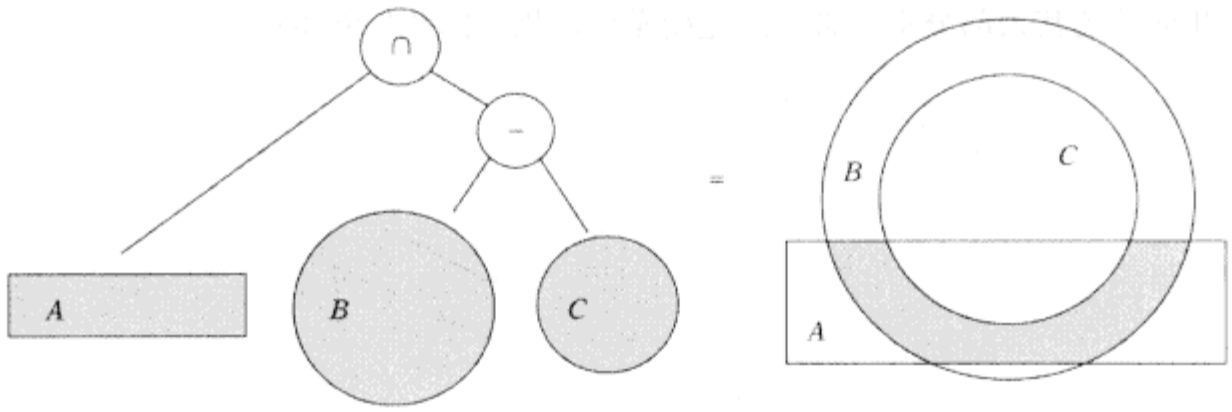


图 12.27　作为 CSG 对象并根据轴对齐包围盒 A 与球体 B、C 之差之间的交集进行建模的半球形碗状模型

因此，对象 M 与该碗形结构的碰撞状态可按照下列逻辑表达式加以确定：

```
bool collides = PartiallyInside(A,M) && PartiallyInside(B,M) && PartiallyOutside(C,M);
```

其中，若对象 **Y** 中的某一部分位于 **X** 的内部，则 **PartiallyInside(X,Y)**为真；相反，若对象 **Y** 中的某一部分位于 **X** 的外部，则 **PartiallyOutside (X,Y)**为真。这里，断言 **PartiallyInside()**只是一个常规的相交测试计算，其实现过程可参见第 5 章。而断言 **PartiallyOutside()**则是一个新出现的碰撞检测计算，但其实现过程也较为直观。例如，线段、三角形、多边形以及多面体部分位于球体之外的充分条件为：对象的一个或多个顶点位于该球体的外部。

12.6　采用欧拉公式的一致性测试

对于结构性较差的几何体，也可通过几何体的拓扑结构对其进行检测。首先，如果两个实体的表面边界无须执行分割或接合匹配即可实现彼此变形，则称其彼此拓扑等价或同胚。其中，多面体的球体同胚称作简单多面体。作为凸多面体，四面体和立方体都是简单多面体的具体实例。

简单多面体的顶点（V）、数据面（F）以及各边（E）的数量关系可简单地表示为欧拉公式，如下所示：

$$V + F - E = 2$$

如图 12.28（a）所示，由于 $V = 9$、$F = 9$、$E = 16$ 且有 $V + F - E = 9 + 9 - 16 = 2$，因此该多面体满足欧拉公式。

若多面体包含一个或多个孔，则归类于非简单多面体。其中，孔的数量则称之为多面体的亏格（genus）。亏格为 0（不含孔）的多面体与球体拓扑等价；包含一个孔的多面体与圆环拓扑等价；包含两个孔的多面体则与形如数字 8 的实体结构拓扑等价等。另一种考查方式是：全部多面体皆拓扑等价于包含 0 个或多个环柄（handle）的球体，其中，环柄的数量由亏格加以确定。欧拉公式的扩展版本为 Euler-Poincaré 公式，对于任意亏格（G）的多面体，有：

$$V + F - E = 2\,(1 - G)$$

考查图 12.28(b)所示的非简单多面体，其中，V=16，F=16，E=32，G=1。该多面体满足 Euler-Poincaré 公式。在 Euler-Poincaré 公式的实际使用过程中，全部多面体数据面需被一个连接顶点环包围，即与圆

盘结构同胚。同时，数据面不应为环状结构且不包含孔。另外，上述两个公式仅适用于流形几何体。因此，每一条边只连接两个数据面，每一条边仅包含两个顶点，每一个顶点至少连接 3 条边。

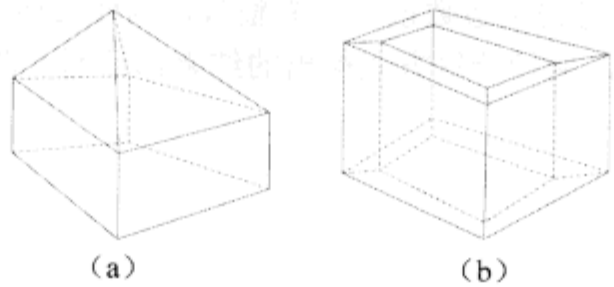


图 12.28　（a）简单多面体。（b）非简单多面体（亏格为 1）

由于将多面体压平至某一平面内并不会改变其顶点、数据面或边的数量，因此，一般来讲欧拉公式也适用于平面图，特别是多边形网格。例如，将一个立方体压平至某一平面后，立方体中的某一数据面将对应于位于压平对象外部的一个极大表面。因此，针对于平面图，在数据面计数时应添加该极大表面，或者适当地调整该公式以正确地处理该数据面。

某些常用表达式也源于欧拉公式。这里考查一个全封闭的三角形网格。由于每一个数据面均包含 3 条边，且每条边关联两个数据面，因此，有下列关系式：$E = 3F/2$。对于任意类型的封闭网格，每一个数据面至少包含 3 条边且每条边均关联两个数据面，因而上述表达式变为 $E \geqslant 3F/2$。分别求解 E 和 F，在进行相应的替换后，可得到 $F \leqslant 2V - 4$，$E \leqslant 3V - 6$。如果封闭网格仅包含三角形，则边数量与三角形（T）数量之间存在关系 $2E = 3T$，对应的公式则变为 $T = 2V - 4$，$E = 3V - 6$。若网格仅包含四边形，则边与四边形之间（Q）有 $2E = 4Q$，对应公式变为 $Q = V - 2$，$E = 2V - 4$。关于顶点、边以及数据面之间的关系可参见表 12.1。

表 12.1　采用欧拉公式时，封闭网格中边、顶点以及数据面之间的关系

封闭（流形）网格的图元构成	边（E）、顶点（V）以及数据面之间的关系
三角形（T）	$2E = 3T, T = 2V - 4, E = 3V - 6$
四边形（Q）	$2E = 4Q, Q = V - 2, E = 2V - 4$
三角形（T）和四边形（Q）	$2E = 3T + 4Q, T = 2V - 2Q - 4$
凸面（F）	$2E \geqslant 3F, F \leqslant 2V - 4, E \leqslant 3V - 6$

欧拉公式及其推导公式皆可用于检测网格是否具备良好的结构特征。需要注意的是，上述公式只可作为必要条件而非充分条件。因此，这一类公式可与其他约束条件结合使用，以实现完整的有效性检测。

作为一个有效性检测的实例，考查如图 12.29 所示的两个立方体，其模型均采用 9 个顶点加以实现。在图 12.29（a）中，立方体包含 6 个四边形以及两个三角形。而图 12.29（b）中的立方体则包含某些问题，其中，在上方数据面中出现了 t-连接（对应的缝隙采用灰色表示）。该立方体由 5 个四边形和 3 个三角形组成。

如果边数量已知，即图 12.9（a）中的立方体包含 15 条边，图 12.9（b）中的立方体包含 16 条边，则根据欧拉公式，左边的立方体有 $9 + (6 + 2) - 15 = 2$，右边的立方体有 $9 + (5 + 3) - 16 = 1$。由此可知，左边的立方体将满足欧拉公式，而右边的立方体则不满足该公式。在多数情况下，相关几何体并不会

直接采用边数量加以表示，而仅仅是数据面和顶点数量。此时，由于立方体仅包含三角形和四边形，因此边数量可采用推导公式 $E=(3T+4Q)/2$ 加以计算。针对左边的立方体，相应的计算结果为 15 条边，而右边的立方体则为 14.5 条边。考虑到边数量必须为整数，因而右边的立方体必定内含某些错误。因此结论为：无论边数量是否已知，两种计算方式得出的结果都证明了图 12.29（b）中的立方体存在建模错误。

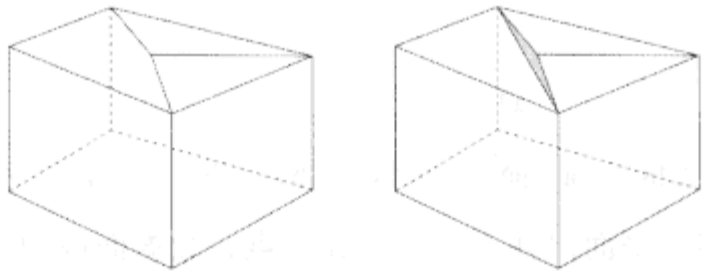


图 12.29　在以三角形和四边形表示的封闭流行网格中，边数量应满足 $E=(3T+4Q)/2$。针对左边的立方体，由两个三角形和 6 个四边形构成，因而包含 15 条边；右边的立方体则由 3 个三角形和 5 个四边形构成，相关计算公式将得出 14.5 条边——非整型结果。这表明：右边的立方体存在建模错误（其上方数据面中存在 t-连接）

另一种检测方法是：针对封闭的流形三角形网格，三角形的数量必须为偶数，即 $T=2V-4$ 的结果总为偶数。另外，对于由三角形和四边形构成的封闭流形网格，其三角形数量也应为偶数，即 $T=2V-2Q-4$ 的结果总为偶数。

12.7　小结

针对碰撞经检测系统，具有良好结构的输入几何体是十分重要的。否则，冗余顶点、非平面多边形、t-连接以及类似的健壮性问题都将导致测试系统出现严重的错误。例如，对象将会落入到几何实体的外部。

本章中所讨论的算法均采用不具备良好结构特征的多边形集合——多边形汤作为输入数据，算法最终将生成具有良好结构的网格。相应的消除操作体现了算法的通用性，并适用于多种场合。

本章还探讨了冗余顶点的焊接操作，以及顶点、边和数据面之间的邻接信息的计算方法，这将有助于实现邻接数据面间的缝隙和 t-连接的修复操作。另外，相关内容还介绍了如何将共面数据面合并为单一数据面，以及平面性和共面性的相关测试方案。由于多数碰撞检测系统一般只支持凸体输入数据，因此本章还详细阐述了凹体对象和数据面的凸（三角形）剖分算法。同时，针对某些无法实现剖分的数据对象（如碗形对象的内部表面结构为凹面），本章也提供了相应的解决方案。最后一部分内容则介绍了欧拉公式，针对封闭网格的有效性测试，该公式不失为一种简洁、有效的测试方案。

第 13 章 优 化 操 作

对高层算法加以改进可视为一种最为重要的优化方法，这一点已被人们广泛接受。然而，对某一任务选择合适的算法仅是众多因素之一，算法依赖于较好的底层数据结构，且该结构的执行效率将受限于目标机器的体系结构。针对高效的数据结构和算法，本章将详细讨论与目标体系结构相关的一些遗留问题，这些问题对于生成高质量的碰撞检测系统是至关重要的。

通过考查运算数量并乘以各运算符的执行时间，可对程序的执行效率进行有效的测试。因此，某些非算法（nonalgorithmic）的代码优化会试图减少执行指令的数量，并利用快速的等价指令代替原指令。虽然这一类优化操作仍具有某些参考价值，但当前的优化方案则更注重于内存优化和并行优化。

CPU 经过几代的演变，处理器的计算速度增长迅速，这远非改进内存的访问时间可比，同时，这一趋势仍将延续下去。因此，若处理器的计算速度停滞不前，内存访问这一桎梏很可能会得到较大的改观。

为了缓解这一瓶颈现象，CPU 制造商往往会在主存和 CPU 之间设置一级或多级高速缓冲存储器。这也体现了一种内存的层次结构，其中，若相关数据未出现于高速的下级缓存中，则只需访问慢速的上级缓存。

缓存的引入极大地改善了数据的访问机制，但该缓存机制本身仍无法完全满足存取速度上的要求，因此，对于性能要求较高的代码，应利用内存的层次结构特征对其加以特殊处理。这也要求程序员显式地实现相应的内存优化操作。这里，内存优化操作包括硬件缓存和软件缓存。

硬件缓存可通过内建于 CPU 的数据和指令缓存加以实现。硬件缓存通常自动生成，却会改变代码和数据在内存中的布局方式，因而可显著地改善代码的执行效率。该级别上的优化操作要求程序员对 CPU 的工作方式有着较为深刻的理解，并能够编写出相应的代码和数据结构。

软件缓存则出现于用户层上，且在应用中需要显式地对其加以实现和管理。必要时，还会涉及某些特定领域中的相关知识。软件缓存可在运行期内实现数据的重组，以获取具有空间一致性的数据。

第二种优化方式是针对代码和数据的并行优化操作。当前，多数 CPU 均提供了针对多个数据流实现并行操作的 SIMD 指令。另外，多数 CPU 均参照超标量体系结构加以设计，并可在一个时钟周期内执行多条指令（假设指令间无中断）。采用并行机制编写的特定代码其执行速度将超出非并行代码许多倍。

本章将首先介绍与 CPU 缓存结构相关的基本原理，并详细阐述内存优化操作。同时，还将对代码和数据进行调整以进一步改善底层缓存机制，并介绍数据的预取和预载操作，从而改善算法的计算性能。在本章的后面将给出相应的缓存优化数据结构的实例，并对软件优化这一概念进行进一步的解释。随后，本章将引入“数据别名”这一概念，详细探讨别名现象所导致的性能瓶颈，并具体阐述减少数据别名现象的多种解决方案。本章还介绍了如何利用 SIMD 并行机制以改善碰撞检测代码的测试性能，特别是底层几何图元的测试。最后，本章还将讨论分支结构对 CPU 指令管线所产生的不良影响及其处理方法。

需要注意的是，优化操作需有针对性。在进行优化操作之前，可先期考查某种替代算法，而后在此基础上进行优化。一般来讲，良好的算法设计更能提供显著的改善效果，这也是调试过程所不能比拟的。需要强调的是，在何处实施优化操作不可凭直觉加以判断，通常可采用代码优化检测工具予以辅助。另外，仅当代码的瓶颈位置已确定且辅以相应的算法和数据结构，方可执行后期的调试过程。由于某些建议的优化方案很难与特定的体系结构实现完美匹配，因此需要采取相应的测试方法对优化效率进行测算。

若不采用汇编代码，则在实现内存优化操作时，需对高级语言代码片段所生成的汇编代码类型存在某种直觉上的判断。高级语言代码中的某些细微变化可能会对汇编代码产生显著的影响。因此，应养成每隔一段时间研习汇编代码的良好习惯。

13.1 CPU 缓存

与主存中指令和数据的读取速度相比，CPU 执行指令的速度则要快很多。考虑到主存的存取时间，高速缓存的引入是十分必要的，其读取速度与 CPU 的指令执行速度不相上下。其中，缓存中的内容多是主存数据的备份，其位置处于 CPU 和主存之间。这里，较为常见的缓存方式包括存储指令的指令缓存和存储数据的数据缓存。

当 CPU 获取下一条指令或当前指令所需的数据时，将首先查看内存数据是否位于对应的缓存中。若是，则相应数据直接从缓存中获取；否则，将导致缓存数据读取失败，此时，CPU 将从主存中读取数据并将其存储于缓存中。二者相比，后者将比前者慢约 50～500 倍。最坏情况下，CPU 在读取数据时将处于等待状态且无数据可处理，这将导致 CPU 出现延迟现象或进入挂起（暂停）状态。另外，如果当前执行指令所引用的数据值需要加载，则仍然会导致 CPU 出现延迟。若数据无法保证位于缓存中，则计算过程将根据当前读取状态被推迟执行。

缓存区由多个 32 或 64 位的最小存储单元构成。位于主存和缓存之间的数据传输将在多个存储单元之间进行。因此，数据的读取开销将不再以字节记数，而是以访问的存储单元数量计算。

一般情况下，可采取直接映射策略将主存映射至缓存。直接映射是指，对于大小为 MKB 的一块缓存区域，主存中连续的 MKB 个字节区域将一一映射至缓存中。最终，两个各自独立的 MKB 字节块将映射至缓存中的同一位置。同时，为了能够有效区分缓存储单元中的相关数据，各缓存单元均包含相应的标识符，用以确定相关数据在主存中的原始地址。

由于两块各自独立的、大小为 MKB 的内存区域将会映射至同一缓存单元中，因此，内存间的交互操作将使缓存存储单元频繁地被刷新、重载。为了降低该问题（即缓存置换操作）所产生的负面影响，缓存间一般被设计成多组相联（多路相联）。对于 n-路相联的缓存结构，每一个逻辑缓存存储单元将分别与 n 个物理缓存存储单元相对应，可对各存储单元进行检测（通过标识符）以查看其是否已存储所需的相关数据。若该数据未曾出现，则包含最早数据的缓存单元将被刷新并被投入使用。在 n-路相联缓存结构中，可同时操作 n 个 MKB 的独立内存区域且不会产生缓存置换问题。通常情况下，缓存一般设计成 2-路、4-路或 8-路相联结构。图 13.1 显示了一个 2-路相联缓存的映射过程。

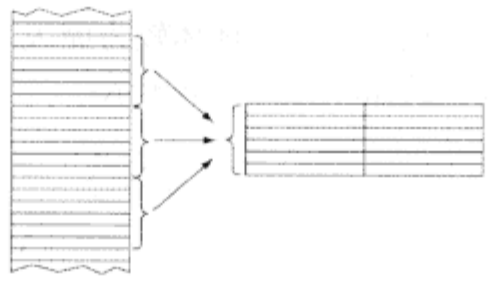


图 13.1　将主存映射为 2-路相联缓存

另外，某些 CPU 还具备多级缓存的能力。例如，1-级缓存或 L1 缓存的存取速度与 CPU 的速度不相上下。考虑到成本问题，这一类缓存的尺寸仅为若干 KB（在编写本书时，目前为 8～32KB），因而缓存失效问题时常发生。

另外，2-级缓存则位于 1-级缓存和主存之间。当 L1 缓存失效时，则立即对 L2 缓存进行检测：仅当 L2 缓存也失效，才对主存进行访问。

论及速度，L2 缓存则不及 L1 缓存，但仍远高于主存的存取速度。由于 L2 缓存的造价比 L1 缓存低廉，因而其容量较大，通常为 128KB～1MB 甚至更大。少数 CPU 在其内存结构中还具有第 3 级缓存。

缓存优化是指数据应尽可能地在缓存范围内操作，并降低缓存失效的次数。与代码或数据相关的缓存失效现象可归为下列 3 种形式（[Hill89]）：

- ☑ 空间失效。这一现象源于缓存空间有限，因而导致有效数据未被存储。该问题可通过适当地压缩数据、对数据进行重组以降低其扩展空间、删除不常访问的数据等措施加以解决（该问题将在 13.3.1 节“结构优化”中详细讨论）。
- ☑ 冲突失效。如果映射至同一缓存单元的被访内存块的数量大于缓存的级联数量，将会产生缓存的置换操作问题。通过压缩、重组数据并使其映射至不同的缓存存储单元，可相应地减少冲突失效的次数。
- ☑ 强制失效。如果内存数据第一次被读取，强制失效则无法避免。可将全部被访数据同步地置入同一缓存存储单元、使用小型数据或紧凑型数据以提高缓存的利用率，从而降低强制失效的次数。

关于缓存的组织方式、存储器结构的其他信息，可参考多种相关书籍，例如[Hennessy02]。

13.2　指令缓存优化

对缓存的利用率进行优化主要包括两部分，首先，需要减少缓存存储数据的尺寸；其次，可适当地增加数据的“局部性”特征，以使同时被访问的数据实现合并存储。在编写汇编代码时，程序员可直接控制上述两个参数；但在使用 C/C++这一类高级语言时，程序员在生成紧凑的机器码时多少会受到编译器的限制。当然，可采用多种手段以助于编译器生成高效的、基于缓存的代码。下列建议将有助于减少代码的尺寸：

- ☑ 应谨慎处理内联代码，内联函数将导致代码膨胀。因此，应仅对小型函数使用内联函数，且较小的内联代码段应与调用该函数的生成代码尺寸相仿。同时，仅内联函数可在.h 文件中加以定义。另外，还应意识到数据别名问题，相关内容将在 13.6 节中加以讨论。

☑ 将循环体分解为缓存尺寸的代码段。当循环体较为庞大且已无法与指令缓存尺寸实现良好匹配时，可将其分解为多个较小的循环体，如下所示：

```
for (i = 0; i < N; i++) {
        ...A...;
        ...B...;
        ...C...;
}
```

其中，**A**、**B**、**C** 分别对应较大的（独立）代码段，建议改写为：

```
for (i = 0; i < N; i++)
        ...A...;
for (i = 0; i < N; i++)
        ...B...;
for (i = 0; i < N; i++)
        ...C...;
```

此时，**A**、**B**、**C** 将分别与缓存尺寸匹配。在执行上述 3 个独立的循环体时，**A**、**B**、**C** 将驻留于缓存中。为了使代码能够驻留于缓存中，添加额外的循环操作还是值得的。对于与指令缓存不相匹配的循环体，则可采用相反的策略。可尝试缩小代码尺寸以使其与缓存尺寸匹配；合并循环体；显式地对重复项使用公共子表达式消元，而非依靠编译器来完成这一类工作；尽量使用简单、易于实现的算法（或具有类似效率的算法）替换复杂的算法。

☑ 避免无谓地展开循环操作。仅需展开较为简洁的最内层循环体，另外，考虑到指令延迟，循环体内的代码能够填补任一指令延迟时间即可，且无须实施过度的展开操作。

☑ 基于代码尺寸进行编译，而非执行速度。这里，将不采用最高级别的优化设置，可尝试开启某些编译选项（如 gcc 编译器中的-Os 选项）以生成较小尺寸的代码，并查看其最终的效果。

☑ 消除蠕变特性。应尽可能地保持函数功能的简洁性。如果某一函数提供了多项处理功能且一次只处理其中的某一项，该函数应改写为多个函数，其中每一个函数分别对应一项功能。

☑ 必要时使用汇编代码。对于某些用时较为苛刻的应用而言，编写汇编代码可能是最佳选择。优秀的底层程序员常能编写出良好的汇编代码，甚至胜过编译器。

有效地改善局部性特征促使程序员尽量编写功能单一的函数，虽然这有时会与传统的软件工程实践相悖。全功能代码将使函数变得过于繁琐。同时，简洁的代码将使该函数更加靠近调用点，从而使其与相应的调用函数位于同一缓存页中。另外，也可在文件间手工移动函数并在调用函数附近加以定义，从而实现被调函数的紧凑性。在某些系统中，通过定义多个代码段，函数在链接过程中可实现自动移动，并对应于代码空间内的一片邻接区域。编译器扩展项也可采用相应的代码段对函数加以标记，从而体现该函数的置入位置（例如，gcc 编译器提供了 **attribute((section ("xxx")))**功能扩展）。如果某一函数出现于多个位置，则有必要提供该函数的多个副本并与每一个调用点相关联。某些代码优化检测工具会提供相应的函数排序，以实现函数间的重组。

在函数内部，当某一语句的分支被跳过时，if 语句将会破坏相应的局域性特征。在某些情况下，两个分支路径完全可以合并为一条路径，一方面可以降低代码的尺寸，另一方面可以有效地改善代码的局部性特征。考查下列代码片段：

```
if (m > n) {
    for (int i = a; i < b; i++)
        c[i] = (big expression involving i) * d[i] + (second big expression involving i);
} else {
    for (int i = e; i < f; i++)
        g[i] = (big expression involving i) * h[i] + (second big expression involving i);
}
```

如代码中所示，将两个 **big expression** 作为独立函数分离出来并不是一种较好的解决方案，对其进行调用将会导致缓存的置换问题。可按照下列方式重写代码：

```
if (m > n) {
    lo = a; hi = b; out = c; in = d;
} else {
    lo = e; hi = f; out = g; in = h;
}
for (int i = lo; i < hi; i++)
    out[i] = (big expression involving i) * in[i] + (second big expression involving i);
```

这一变化使得代码尺寸缩减为原来的一半。

多种体系结构均提供了一条或多条缓存控制指令用以改善指令缓存的应用效果。特别地，这些指令可指示 CPU 将位于一个或多个缓存单元中的代码置入当前指令缓存中。另外，通过发布缓存指令以提前调用函数地址，该函数在调用时将出现在缓存内。然而，在实际应用中很难体现这些指令的应用价值。可采用相应的优化检测操作以对最终效果加以评估。高效地利用指令缓存则需要读者深入了解编译器生成的机器码以进一步判断程序的执行效率，并能够清醒地认识到高级语言代码的改变如何影响到生成的机器码。

13.3 数据缓存优化

数据缓存的利用率也可通过降低数据尺寸、提高空间和时间局域性特征得到改善。虽然减少数据尺寸可视为一种压缩编码方案，但这里很少使用相应的编码算法，例如 Huffman 编码、Lempel-Ziv 编码以及算术编码（[Sayood00]）。这一类算法更适用于具有较高延迟时间的大型数据集，例如二级存储设备或网络连接。若一次仅在主存和缓存之间传输若干字节，则算法的压缩时间甚至会远远大于其节省的时间。这里，数据压缩算法首先对存储数据进行评估，以查看该数据是否有必要加以存储从而防止数据被重复存储，以及是否可采用较小的数据类型实现存储。其中，相对简单的行程编码算法可视为一类有效的解决方案。

通过重新设计算法和数据结构并以一种更具预测性的、线性或部分线性的方式访问数据，可有效地改善数据局域性特征这一问题。该过程通常会涉及分块（blocking）操作：将较大的数据分解为较小的数据块以便与缓存尺寸相匹配。

在底层层面上，该方案可将缓存存储单元的操作数量降为最低，并对数据加以组织以实现同时读取缓存单元中的全部或多数字数据。例如，可从渲染数据中分离出碰撞检测数据，即除去 RGB 颜色数

据、纹理坐标数据以及其他与碰撞检测不相关的数据，并交替排列顶点数据以及其他相关数据，从而尽可能地消除数据的局部性特征。

13.3.1 结构优化

针对数据缓存的改进措施，首先考查结构和类的优化方案，包括下列 3 点：

- ☑ 降低结构的整体尺寸。使用恰当的位数表示数字，从而可降低缓存单元中的匹配数据量以及结构中相关字段的内存读取次数。
- ☑ 在结构内适当地重组字段。结构内的相应字段通常按照其自身含义加以分类，但从本质上讲应以访问模式加以分组，以使字段的访问-存储过程相吻合。
- ☑ 根据数据的使用频率划分结构。由于结构内部相关字段的访问频率不同，可将那些不经常访问的字段置入单一结构中，从而增加频繁受访数据缓存间的一致性，特别在该结构是数组或其他聚合型数据结构的某一部分时。

许多平台上的对齐操作将会迫使编译器对相关结构实施填充操作，即 *N* 字节字段将实现 *N* 字节对齐（例如，32 位整型数内存对齐地址均可被 4 整除）。由于填充结构的对齐尺寸往往是该结构中最大 *N* 字节字段的某一倍数，因此会浪费一部分内存空间。考虑到填充操作，按降序排序成员变量将在一定程度上节省内存空间。另外，某些编译器会提供相应的警告，以助于检测当前内存空间的消耗状态（例如，在 gcc 编译器中，可开启-Wpadded 选项）。

基于 MIPS 平台的体系结构往往有相应的对齐限制。其中，若定义了下列数据结构：

```
struct X {                  struct Y {                  struct Z {
    int8 a;                     int8 a, pad_a[7];           int64 b;
    int64 b;                    int64 b;                    int64 e;
    int8 c;                     int8 c, pad_c[1];           float f;
    int16 d;                    int16 d, pad_d[2];          int16 d;
    int64 e;                    int64 e;                    int8 a;
    float f;                    float f,pad_f[1];           int8 c;
};                          };                          };
```

则多数编译器将会生成如下结果：**sizeof(X) == 40,sizeof(Y) == 40,sizeof(Z) == 24**（这里，假设 float 类型为 4 字节）。与排序后的结构 **Z** 相比，结构 **X** 的填充数据占据了全部内存空间的 40%（如结构 **Y** 所示）。其他缩减结构尺寸的相关方法如下所示：

- ☑ 从已存储值可轻易计算得到的数据值则不要存储于结构中。例如，若三角形的顶点数据已从内存中读取，则可通过该顶点数据直接推导出法线和平面方程，而无须从该三角形数据结构中获取相关数据，以避免额外的缓存读取操作。
- ☑ 可能的话，应尽量使用小尺寸整型数据。这里，不必总是使用默认的整型数据（通常大于 32 位），可根据具体情况采用 16 位甚至是 8 位数据。例如，若玩家的生命值定义为整型数据且范围是 0～100，则应存储为 8 位的字节数据而非 32 位整型数据。
- ☑ 采用位域表示法而非布尔值。尽量不采用 **bool** 类型的标志变量，可将全部标志封装至某一位域中。**bool** 变量的尺寸是依赖于机器环境的，其范围从一个字节至长整型数据不等。将多个标志存储于位格式中，可将所需内存空间至少减至 8 倍，通常为 32 倍或更多。

- ☑ 采用偏移量而非用于索引数据结构的指针。指针通常为 4 字节（或更多）固定尺寸变量，而偏移量的尺寸却可根据索引项的数量产生变化。例如，2 字节偏移量可索引包含 216 个对象的数组（如果偏移量根据当前对象设置，则相应数量会更多）。当然，如此数量的索引操作一般很少出现。

由于缓存失效将导致计算开销急剧增加，因此必须谨慎处理压缩/非压缩数据。13.3.2 节和 13.4 节分别给出了基于数据压缩的相关实例。

需要指出的是，在某些系统结构中，上述方案可能会带来额外的 CPU 计算开销。例如，附加指令要求针对小尺寸整型数据或位域中的某一位进行操作。随着指令运行数量的增加，缓存压力也将会随之增长，进而会逐渐抵消缓存优势。因此，对代码进行优化测试不失为一种较好的解决方案——理想情况下，可对变化前后的代码分别进行测试，从而采取相应的改进方案。

对相关字段加以记录实现起来并非易事。在某些情况下，多个字段将存储于同一缓存单元中，尤其是多个字段被同时访问时。考查下列情况，位置、速度以及加速度字段经常被同时访问，因而对其进行合并存储将会增加执行效率。多数情况下，确定字段的最佳排列方式其过程往往比较复杂。理想情况下，编译器将会参考前次运行信息并在后续编译过程中自动地执行相应的字段记录。然而，相应的语言标准很可能并不包含字段记录这一功能，且当前基本不存在相应的编译器能够支持该优化操作。一种较好的字段重组方案是通过访问函数间接地获取结构中的字段。这里，可临时性地设置这些函数，以测试字段的访问频率并记录哪些字段被同时访问。另外，在某些情况下，对结构进行填充操作将会有效地改善程序的计算性能，具体体现为：这将使被访的两个邻接字段位于同一缓存内。

考虑到结构字段的访问频率，可将其划分为两个不同的独立结构：高频被访字段结构和低频被访字段结构，应分别对其分配内存空间并加以存储。另外，各个高频字段结构中均嵌入低频字段结构的链接指针（如图 13.2 所示）。此处，虽然只能对低频字段结构实现间接访问并增加了些许计算开销，但相应的主结构（高频被访字段结构）将变得更加紧凑，并可实现高效的数据缓存。另外，由于结构的访问方式不同，不必一定采用链接指针。例如，使用单独的索引即可同时访问高频被访字段数组和低频被访字段数组。

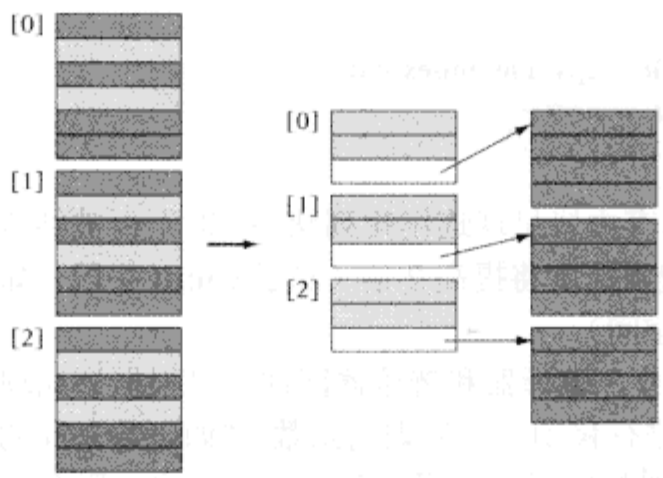


图 13.2　定义 3 个实例化结构数组（其中，高频访问字段采用浅灰色表示），高频/低频划分方案将对数据实施重组，并对低频访问字段单独存储，即结构数组中只包含高频受访字段（辅以低频受访字段对应的链接）

下面的示例显示了结构的划分过程且不使用链接指针。这里，相关代码用于遍历结构数组，当某

元素具有最大值 **value** 时，则增加该元素的 **count** 字段值。

```
struct S {
    int32 value;
    int32 count;
    ...
} elem[1000];

// Find the index of the element with largest value
int index = 0;

for (int i = 0; i < 1000; i++)
    if (elem[i].value > elem[index].value) index = i;
// Increment the count field of that element
elem[index].count++;
```

如果上述基于结构数组的搜索过程视为核心操作，则应将结构 **S** 划分为高频受访字段结构（**S1**）和低频受访字段结构（**S2**）。上述代码可改写为：

```
// Hot fields of S
struct S1 {
    int32 value;
} elem1[1000];

// Cold fields of S
struct S2 {
    int32 count;
    ...
} elem2[1000];

// Find the index of the element with largest value
int index = 0;
for (int i = 0; i < 1000; i++)
    if (elem1[i].value > elem1[index].value) index = i;
// Increment the count field of that element
elem2[index].count++;
```

因此，全部 **value** 字段在内存内皆呈现连续排列状态，即不再被结构 **S** 中的 **count** 字段和其他字段分隔。因此，最大值 **value** 的搜索速度将提高 2 倍（除了 **count** 字段，如果还能析取出其他低频受访字段，搜索速度还有更大的提升空间）。

类似于结构的低频/高频划分，编译器和链接器同样也可根据该原理对代码进行重组，因而可对调用频率较低的函数或部分函数进行移动，降低调用函数-被调函数之间的冲突，将会直接减少指令缓存中的操作次数。一类优化算法则是使用了“缓存着色技术”这一概念并对代码（或数据）进行排列，进而将受访频率较高的数据项映射至不同的缓存单元中（各缓存单元可视为彼此不同的颜色值），相关内容可参见[Hashemi97]。但目前为止，还很少有相应的编译器能够实现缓存着色技术。虽然该方案可实现手工处理，但一旦代码产生变化，全部过程也需要重新处理。另外，[Chilimbi99a]中也对数据的缓存布局问题进行了详细的讨论。关于结构划分和字段重组问题，还可参见[Chilimbi99b]。

13.3.2　顶点数据的量化操作和压缩操作

针对碰撞检测代码，除去数据结构自身之外，顶点的表达方式也是十分重要的，因为它们构成了数据的主要组成部分。虽然顶点的 x、y、z 值常采用浮点数加以存储，但也可对其实施量化操作。例如，可使用 3 个 16 位的整型值加以表示。相应的量化计算可有效地实现网格顶点的对齐操作，因此须谨慎处理以避免四边形或其他大型碰撞测试图元产生共面现象。

在某些情况下，还可以对顶点数据实现进一步的编码。例如，对于小型场景世界，顶点可编码为 32 位整型数据，其中顶点的 x、y 值分别占据 11 位，z 值则包含 10 位（如图 13.3 所示）。

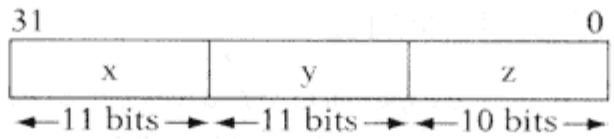


图 13.3　量化为 32 位整型数据的顶点值。其中，x 和 y 值分别包含 11 位数据，z 值包含 10 位数据

当采用相对于前一顶点的偏移量来表示其他顶点时，顶点数据将会占用更少的内存空间。例如，在计算由顶点表示的 AABB 时，可将其中心点作为固定原点，并根据该原点计算其他顶点的偏移距离。这里，额外开销仅为采用全精度表示的原点，其他顶点则通过较少的几个数据位即可实现。

例如，考查下列 5 个 16 位整型随机顶点：A = (30086,15857,9385)，B = (30189,15969,9285)，C = (30192,15771,9030)，D = (29971,15764,9498)，E = (30304,15888,9133)。上述顶点也可以采用基于顶点(30138, 15866, 9264)的 16 位偏移量加以表示：(−52, −9, 121)、(51, 103, 21)、(54, −95, −234)、(−167, −102, 234)、(166, 22, −131)。这里，量化顶点可采用 9 位数值加以存储，相对于原点格式，这节省了大量的内存空间。

若原点可在顶点间浮动，则还能够进一步地节省内存空间。亦即，可通过前一顶点计算偏移量，因而能够有效地实现数据压缩。该方案适用于索引网格，其中，顶点顺序可实现随机混合，这将有助于获取较好的压缩率。综上所述，可对顶点 A～E 按照 D、A、B、E、C 这一顺序加以重组，首顶点表示为(29971, 15764, 9498)，且后续顶点分别为(115, 93, −113)、(103, 112, −100)、(115, −81, −122)、(−112, −117, −73)。因此，顶点分量可采用 8 位数据加以表示。

由于叶节点中的数据一般仅跨越场景世界中的某一小部分，因而空间划分算法可与量化格式的叶节点数据结合使用。另外，局部数据的量化操作不但可以节省内存空间，还能够实现坐标值的全范围表示。如果相关叶节点数据待压缩后实现缓冲存储，其额外的压缩消耗通常可以分摊的方式加以抵消。令叶节点数据相对于某一原点（该原点表示父节点的空间位置）实现存储，则可增加该处理过程的健壮性。此时，相关计算将围绕此原点进行，从而可实现较高的精确度。

13.3.3　预取和预载操作

非缓存数据的加载访问所导致的延迟将大大地降低执行速度，因此，多数 CPU 均提供了相关机制以避免这一情况的出现。其中一类机制称为预取指令，它允许程序员在 CPU 连续执行指令时指示 CPU 加载某一特定缓存单元中的内容至缓存中。通过提前发布预取指令，数据将在程序需要时加载至缓存

中，则 CPU 可实现零延迟操作。但这一操作过程也存在着相应的风险：若预取指令过早发布，在 CPU 计划读取相关数据时，该数据可能已从缓存中清除。相反，如果预取指令发布迟缓，则该操作的优势将会丧失殆尽。理想状态下，预取指令应适当地提前发布，从而有效地解决数据加载时的内存延迟问题。

针对线性结构，预取指令不失为一种高效且易于使用的操作方式。例如，考查下列简单的循环处理：

```
// Loop through and process all 4n elements
for (int i = 0; i < 4 * n; i++)
    Process(elem[i]);
```

适当地将代码展开，上述内容可利用预取操作加以改写。这里，假设缓存单元匹配于 4 个连续的 **elem** 值，下列代码定义了一种较好的循环体结构：

```
const int kLookAhead = 4;                  // Experiment to find which fixed distance works best
Prefetch(&elem[0]);                        // Prefetch first four elements
for (int i = 0; i < 4 * n; i += 4) {       // Unrolled. Process and step 4 elements at a time
    Prefetch(&elem[i + kLookAhead]);       // Prefetch cache line a few elements ahead
    Process(elem[i + 0]);                  // Process the elements that have already
    Process(elem[i + 1]);                  // been fetched into memory on the
    Process(elem[i + 2]);                  // previous iteration
    Process(elem[i + 3]);
}
```

如果函数 **Process()**的执行时间不小于内存延迟时间的 1/4，则下一个 **elem** 值的缓存单元将在下一次循环迭代中出现于内存当中（若函数执行时间仍存在提升空间，数据预取操作将会提前两个缓存单元完成）。

对于非线性结构，预取操作则具有一定的技巧性。最为直观的方法是对全部可能分支以及潜在最佳方案实施贪婪预取操作。例如，二叉树的前序遍历即可视为一类贪婪预取操作，如下所示：

```
void PreorderTraversal(Node *pNode)
{
    Prefetch(pNode->left);              // Greedily prefetch left traversal path
    Process(pNode);                     // Process the current node
    Prefetch(pNode->right);             // Greedily prefetch right traversal path
    PreorderTraversal(pNode->left);     // Recursively visit left subtree
    PreorderTraversal(pNode->right);    // then recursively visit right subtree
}
```

贪婪预取操作实现起来较为简单，但并非最为高效的算法。另外，在代码返回至树型结构被访部分之前，预取区域中的数据一般不会从缓存中清除。当预取操作与线性操作联合使用时，其执行效率将得到大大提升（参见 13.5 节）。

针对内存读取延迟问题，某些 CPU 则采用了非阻塞载入机制。非阻塞载入机制在出现缓存失效后 CPU 将继续执行后续指令，仅当寄存器中未出现相关数据时出现延迟。需要注意的是，除非硬件自身支持多个载入操作，否则，若当前加载操作尚未结束且出现某一后续加载操作，CPU 仍会出现延迟，并导致第二次缓存失效。在加载操作执行过程中，通常会采取“缺失命中”操作，即后续的缓存数据

加载操作将会消除 CPU 延迟。

针对采用非阻塞机制的、缺失命中加载操作类型的 CPU，前期循环数据可较好地实现交叉写入，并在读取下一个缓存单元时，相关处理过程将呈现零延迟操作。

```
Elem a = elem[0];
for (int i = 0; i < 4 * n; i += 4) {        // Load the cache line of the first four array elements
    Elem e = elem[i + 4];                   // Cache miss. Fetches next cache line using nonblocking load
    Elem b = elem[i + 1];                   // These following three values are in cache and...
    Elem c = elem[i + 2];                   // ...are loaded as hit-under-miss; no stalls
    Elem d = elem[i + 3];
    Process(a);                             // Process the data from the cache line that...
    Process(b);                             // ...was fetched in the previous iteration
    Process(c);
    Process(d);
    a = e;                                  // e now in cache, and so is b, c, and d of next iteration
}
```

上述代码将读取紧邻数组 **elem** 尾部的某一外部元素，且代码并未对其实施进一步的处理。若数组外部数据读取过程中出现相关问题，可相应地析取出最后一次循环，并对其实施单独处理。

13.4　基于缓存感知的数据结构和算法

设计者应具备相应的缓存优化意识。针对特定平台所设计的相关算法和数据结构称为缓存感知（cache aware）或缓存敏感（cache conscious）型设计。

下面将分别给出相应的代码示例，以进一步说明缓存对代码设计所产生的影响。首先将讨论具有高效缓存特征的紧凑型 *k*-d 树；其次，将分析 AABB 树结构的紧凑型表现方式，进而阐述如何采用量化操作以降低存储需求。

针对需跨平台运行的代码，一种多平台缓存版本的解决方案是实现缓存参数无关的代码，且该类代码在众多传统的缓存结构中均可实现高效的缓存方案。关于参数无关的缓存机制还将在 13.4.3 节中加以讨论。

13.4.1　紧凑型静态 *k*-d 树

通常情况下，*k*-d 树中的节点须具备相应的类型标识，以表明为内部节点还是叶节点。如果节点属于内部节点，则该节点应至少包含两项值：分割轴以及该轴向上的分割点坐标值。如果节点表示为叶节点，则该节点中须包含相关数据的数组指针或索引。这里为了便于使用，分割数据值将定义为单精度浮点数。

这里，改变浮点尾数的低 23 位对数值的影响并不明显。针对 *k*-d 树的分割面而言，这一变化则不存在实际意义。因此，尾数的两个低位可用于存储其他数据。特别地，可利用这两个数据位标识节点类型，且不会产生任何副作用。例如，使用 00、01、10 分别表示 *k*-d 树内部节点分割轴的 *x*、*y*、*z* 轴。

而值 11 则可将 *k*-d 树节点标识为叶节点。针对叶节点，其余 30 位数据将存储一个索引值，并用于查询叶节点中的最终数据。基于该思想，*k*-d 树节点可按照下列方式加以表示：

```
union KDNode {
    float splitVal_type;            // nonleaf, type 00 = x, 01 = y, 10 = z-split
    int32 leafIndex_type;           // leaf, type 11
};
```

其中，将节点类型位定义为浮点数的低二位，则 **splitVal_type** 则可作为绑定了类型位的浮点数加以使用。如前所述，对分割面稍加移动即可恢复其原始状态。综上所述，*k*-d 树节点可采用 32 位值加以描述。

这里，假设缓存单元位宽为 64 字节，在宽度优先的情况下，单一缓存单元可容纳全部 4 层、15 个节点的 *k*-d 树（因此，索引由 0 开始且为 n 的某一节点，其子节点的位置为 $2n+1$ 和 $2n+2$），其全貌如图 13.4 所示。

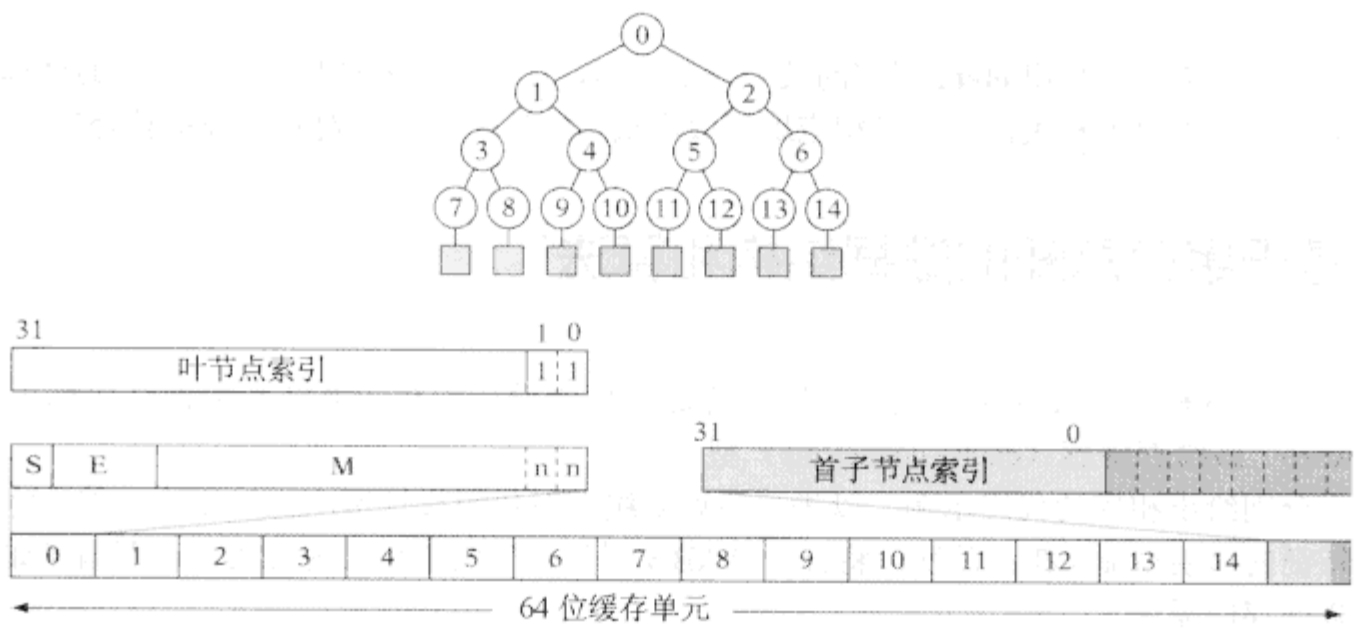


图 13.4　*k*-d 树的高效表示方法。其中，大小为 64 字节的缓存单元将在前 60 个字节中装载包含 15 个节点的 4 层 *k*-d 树数据；缓存单元的最后 4 个字节则分别表示 8 棵子树及其相应位置

因此，可以 4 层为单位对树型结构实施剪裁，并按照宽度优先的方式输出包含 15 个节点的子树，从而实现对完全树的存储操作。针对节点 **KDNode**，给定其指针 **pNode**，则可快速确定指向子节点的各指针且无须对内存实施进一步的访问。

```
// Align tree root to start of cache line (64-byte aligned)
void ComputeChildPointers(KDNode *pRoot, KDNode *pNode, KDNode **pLeft, KDNode **pRight)
{
    int32 nodeAddress = (int32)pNode;
    int32 nodeIndex = (nodeAddress & 0x3f) >> 2;          // node index within cache line (0-14)
    int32 leftAddress, rightAddress;
    if (nodeIndex < 7) {
        // Three out of four, children are at 2n+1 and 2n+2 within current cache line
        leftAddress = nodeAddress + (nodeAddress & 0x3f) + 4;
        rightAddress = leftAddress + 4;
```

```
    } else {
        // The children are roots of subtrees at some other cache lines
        int32 rootAddress = (int32)pRoot;
        int32 cacheLineFromStart = (nodeAddress - rootAddress) >> 6;
        int32 leftCacheLine = 16 * cacheLineFromStart + 1;
        int32 bitIndex = nodeIndex - 7;          // (0-7)
        leftAddress = rootAddress + leftCacheLine * 64 + bitIndex * 2 * 64;
        rightAddress = leftAddress + 64;         // at next consecutive cache line
    }
    *pLeft = (KDNode *)leftAddress;
    *pRight = (KDNode *)rightAddress;
}
```

对于深度为 d 的树型结构，在最坏情况下，单次根节点-叶节点前序遍历将访问 d 次缓存单元，而现在只需读取 $d/4$ 次缓存单元。

即使每个节点仅分配 4 个字节，存储一棵完全树仍然成本高昂。这里，针对树型结构的深度值，每棵子树将不再囊括全部 16 个子节点，而只是对当前活动子树加以存储。缓存单元中余留的 4 个尾字节可用于链接其他子树，以体现一种较为紧凑的表达方式。令前 3 个字节表示为当前已知子树数组的索引值，从而确定首子树的起始位置；且最后一个字节的 8 位数据可用来判断 8 组子节点中的某一组节点。其中，位于第 4 层的子树节点可能包含两个子节点或无子节点。另外，在计算 3 字节索引值的相关偏移量（定位某一组子树）时，与前述树型结构对应的位数据将分别执行加法操作以及乘法操作（乘以 128）。综上所述，下列代码显示了 k-d 树左、右子树指针的计算过程：

```
// Align tree root to start of cache line (64-byte aligned)
void ComputeChildPointers(KDNode *pRoot, KDNode *pNode, KDNode **pLeft, KDNode **pRight)
{
    int32 nodeAddress = (int32)pNode;
    int32 nodeIndex = (nodeAddress & 0x3f) >> 2;          // node index within cache line (0-14)
    int32 leftAddress, rightAddress;
    if (nodeIndex < 7) {
        // Three out of four children are at 2n+1 and 2n+2 within current cache line
        leftAddress = nodeAddress + (nodeAddress & 0x3f) + 4;
        rightAddress = leftAddress + 4;
    } else {
        // The children are roots of subtrees at some other cache lines
        int32 rootAddress = (int32)pRoot;
        int32 bitIndex = nodeIndex - 7;                   // (0-7)
        // Last word on cache line specifies linking of subtrees
        int32 linkWord = *((int32 *)(nodeAddress | 0x3c));
        assert(linkWord & (1 << bitIndex));               // must be set
        int32 offset = PopCount8(linkWord & ((1 << bitIndex) - 1));
        leftAddress = rootAddress + ((linkWord >> 8) + offset * 2) * 64;
        rightAddress = leftAddress + 64;                  // at next consecutive cache line
    }
    *pLeft = (KDNode *)leftAddress;
    *pRight = (KDNode *)rightAddress;
}
```

函数 **PopCount8()**用于计算字节内被设定的位数，并采用循环并行的方式加以实现。首先，对全部奇数位实施掩码操作，并与偶数位相加以构造（连续的奇、偶位值组合）部分和。随后，该部分和将以同样的方式进行掩码操作和加法操作，直至获得最终结果。

```
inline int32 PopCount8(int32 n){
    n = n - ((n & 0xaa) >> 1);          // Form partial sums of two bits each
    n = (n & 0x33) + ((n >> 2) & 0x33);  // Form partial sums of four bits each
    return (n + (n >> 4)) & 0x0f;        // Add four-bit sums together and mask result
}
```

另外，函数 **PopCount8()**也可采用小型查询表（4 位或 8 位宽度）的方式加以实现。与完全树存储方案相比，该方法使得子树地址的计算开销稍稍增加，但仍将低于缓存失效所导致的计算开销。

对上述方案稍作改变即可用于其他位宽值的缓存单元中。例如，对于 32 字节的缓存单元，需相应地降低节点数量以容纳 3 层树型结构而非 4 层。

本章所讨论的 *k*-d 树首次出现于[Ericson03]中。另外，[Szécsi03]也给出了一种类似的 *k*-d 树高效处理方案。

13.4.2 紧凑型 AABB 树

稍加观察即可得到紧凑格式的 AABB 树型结构：由于父 AABB 的 6 个面皆来源于子 AABB，因此，根据父 AABB 的边界范围，最多只需要 6 个附加面即可重构两个子 AABB，亦即，父 AABB 的最小 x 值应取左子 AABB 或右子 AABB 的最小 x 值。另外，还需要 6 个数据位用于确定继承于父 AABB 面的子 AABB，以及确定获取 6 个附加面之一的子 AABB。

这里，新增面将不采用浮点数表示，每一个面将分别占据一个字节，并定义为与父 AABB 面的比率格式（[Gomez01]）。下列代码显示了这一类节点的排列方式，以及基于父 AABB 和压缩 AABB 节点的重构方法。

```
// Uncompressed AABB
struct AABB {
    float & operator [](int n) { return ((float *)this)[n]; }
    Point min;          // The minimum point defining the AABB
    Point max;          // The maximum point defining the AABB
};
// Compressed AABB node
struct PackedAABBNode {
    uint8 flags;        // Bits determining if new extent belong to left or right child
    uint8 newExtent[6]; // New extents, quantized within the parent volume
    ...                 // Potential links to children n
};

void ComputeChildAABBs(AABB &parent, PackedAABBNode &node, AABB *left, AABB *right)
{
    for (int i = 0; i < 6; i++) {
        int xyz = i & 3;
        // Compute the actual side value from the quantized newExtent[] value
```

```
            float min = parent.min[xyz], max = parent.max[xyz];
            float val = min + (max - min) * (float)node.newExtent[i] / 255.0f;
            // Test bits to see which child gets parent side value and which gets new value
            if (node.flags & (1 << i)) {
                (*left)[i] = parent[i];          // Left child gets parent's bound
                (*right)[i] = val;               // ...right child gets parent's bound
            } else {
                (*left)[i] = val;                // Left child gets computed bound
                (*right)[i] = parent[i];         // ...right child gets computed bound
            }
        }
}
```

标志字段中的两个剩余数据位可用于标识节点类型，即叶节点或内部节点。在[Gomez01]所提供的方案中，指向子节点的指针定义为两个 16 位索引值，从而使节点的整体尺寸变为 11 个字节。而对于静态树型结构，则可以省略指针项，从而使得节点空间更加紧凑。需要注意的是，在数据封装时需要进行正确的舍入操作，从而使量化后的子 AABB 能够正确地表示原子空间。

13.4.3　缓存参数无关性

缓存参数无关算法及其相关数据结构则可视为另一类解决方案（[Frigo99]）。该方案并不针对某一具体的体系结构进行优化操作，而是运行于包含各类缓存设备的、彼此各异的目标机器之上。相对于前述缓存感知算法，该方案的复杂度将不低于 $O(\log n)$。

针对缓存参数无关算法及其对应的数据结构，其背后思想包括：对缓存尺寸为 L，内存尺寸为 Z 的理想缓存模型进行优化操作，其中，L 和 Z 未知。由于 L 和 Z 可取任意值，因此，可对存储层次结构中的各级缓存使用优化的算法和数据结构。当然，这也包括各类 CPU 设备。

[Prokop99]中描述了一些缓存参数无关算法，如漏斗排序算法。其中，Prokop 概述了如何以参数无关的方式输出一棵二叉树，这一过程也称作 van Emde Boas 结构（[StøltingBrodal01]）。该树型结构通常以递归的方式加以定义，并与碰撞检测结构以及常见的游戏结构紧密相关。这里，定义一棵高度为 h 的树型结构 T，且根据其高度进行折半划分，即 T_0 表示为原树型结构 T 中小于$\lceil h/2 \rceil$部分的节点树。同时，$T_0 \sim T_k$ 则分别表示为以 T_0 叶节点为根节点的、原树型结构 T 中的各子树，且顺序为自左向右。因此，树型结构 T 的 van Emde Boas 表示法可归纳为：T_0 的 van Emde Boas 结构及其后续 $T_1 \sim T_k$ 的 van Emde Boas 结构。图 13.5 显示了一棵高度为 4 的二叉树的 van Emde Boas 结构。

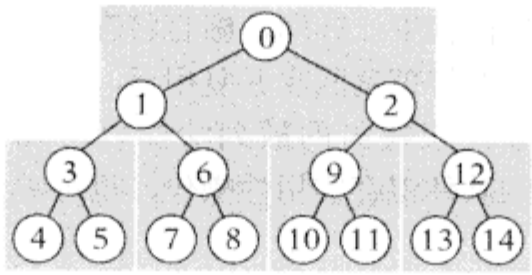


图 13.5　高度为 4 的完全二叉树的 van Emde Boas 结构。相关数字表示为线性数组中的各节点位置

在实际应用中，虽然缓存感知算法可能会优于缓存参数无关算法，但后者在跨平台计算时将具有

更大的优势。

13.5 软件缓存

在相关应用中，为了进一步提高 CPU 数据缓存的利用率，自定义的软件缓存方案也是一类较为有效的方法。这里，一个较为重要的概念是数据的线性化操作。所谓线性化操作是指：对并发访问的、分散于邻接数据块中的数据实现重组，藉以增加数据的空间局部性特征，从而改善代码的计算性能。因此，根据具体需求采用自定义的方式执行线性操作并对结果加以缓存，则可对分散存储的数据在运行期内实现相应的整合。例如，采用简单的查询体对树型结构的全部叶节点执行相交测试，以加速空间内的后续查询操作，即可视为一类缓存线性操作。

另外，软件缓存还可以保持碰撞数据的紧凑型。例如，树型结构的叶节点可在主数据结构中实现压缩存储，且可针对叶节点实现软件缓存。因此，当访问到某一叶节点时，将首先查看其缓存状态：若该节点未实现缓存，则对相关压缩数据进行解压缩操作，并缓存其叶节点 id 以作为键值。具体应用将在后续小节中加以详细讨论。

从另一角度上讲，采用软件缓存还可进一步节省内存空间。通常情况下，为了减少运行期内的计算，可对多数数据进行存储并根据现有相关数据在运行期内实现进一步的计算。其中，冗余数据可能包括叶节点中的 AABB 边界数据，以及与碰撞数据面绑定存储的面法线。

采用软件缓存后，上述数据可脱离于对应的存储数据。例如，面法线可根据顶点进行实时计算，并在缓存步骤中加以存储。另外，软件缓存还可用于记录某些开销较大的计算过程，如碰撞图元的转换操作。

考虑到多数碰撞查询所呈现的空间、时间一致性特征，数据的解压、推导过程以及后续的缓存操作所产生的计算开销将由多次查询操作加以分摊，从而使得每次查询的缓存消耗基本上可以忽略不计。由于只是针对查询的叶节点实施缓存操作，因此空间一致性将大幅削减相应的缓存数据。

软件缓存的另一个优点是缓存中的数据可参照所有节点采取统一的格式。因此，新增的数据结构可通过某种方式实现统一格式的数据缓存，进而添加到相应的系统中去。另外，格式的统一使得底层图元测试代码只需编写一次，从而避免了数据格式的改变对相关代码所产生的不良影响，最终降低了优化操作的内在风险。

根据高层缓存系统的实现方式，应向用户提供相应的选项以在按需调用的基础上开启或关闭软件缓存操作，因此，对于某一用户已知的查询操作，则避免了无谓的缓存时间开销。另一个较为有效的设计选项是令缓存操作处于“软件”内存中。亦即，缓存将存在于内存管理堆中的未用部分中，并且待该内存区域通过某次分配操作被内存管理器回收并复用后，对应缓存部分将变得不再有效。通常情况下，软件缓存可通过采用定尺寸元素数组的散列法加以实现。关于散列法的具体内容可参见[Knuth98]。另外，关于游戏环境中的软件缓存实用性操作，也可参见[Sauer01]。

13.5.1 缓存线性化操作实例

这里通过跟踪索引值、获取并缓存全部引用数据以移除全部间接操作。软件缓存将有助于实现数

据结构的线性化操作。与此同时，增加数据的紧凑性也将会显著地改善 CPU 的数据命中率，从而进一步提高整体计算性能。另外，线性化数据的预取操作和预载操作实现起来也更加方便（参见 13.3.3 节）。

为了进一步演示软件缓存方式的数据结构线性化操作过程，考查下列树型结构，其中，叶节点包含了一定数量的三角形图元。为了节省存储空间，叶节点三角形则采用 16 位索引数组方式加以存储。而各三角形则依次采用 3 个 16 位索引的顶点数组加以存储。该数据格式如图 13.6 所示且有下列代码：

```
// Array of all vertices, to avoid repeating vertices between triangles
Point vertex[NUM_VERTICES];

// Array of all triangles, to avoid repeating triangles between leaves
struct IndexedTriangle {
        int16 i0, i1, i2;
} triangle[NUM_TRIANGLES];

// Variable-size leaf node structure, containing indices into triangle array
struct Leaf {
        int16 numTris;          // number of triangles in leaf node
        int16 tri[1];           // first element of variable-size array, with one or more triangles
};
```

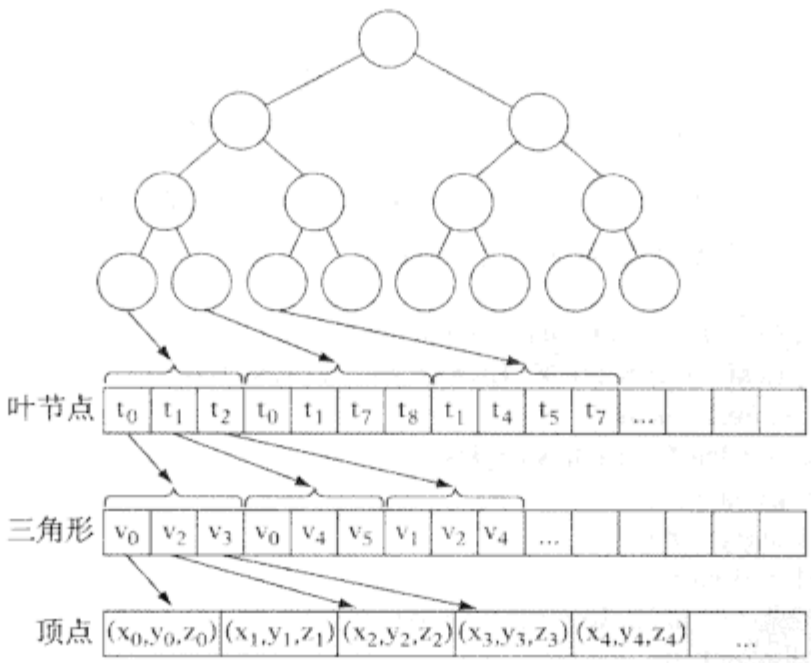


图 13.6　以紧凑格式存储的树型结构并包含诸多个间接操作：分别由叶节点索引和其中的三角形索引加以表示

叶节点可缓存于相应的叶节点数据结构中并作为一个连续的数据块包含全部相关信息，从而消除叶节点结构中固有的间接操作，如下所示：

```
// Explicitly declared triangle, no indirection
struct Triangle  {
        Point p0, p1, p2;
};

#define NUM_LEAFCACHE_ENTRIES (128-1)          // 2^N-1 for fast hash key masking w/'&'
```

```
// Cached leaf node, no indirection
struct CachedLeaf {
    Leaf *pLeaf;                                    // hash key corresponding to leaf id
    int16 numTris;                                  //number of triangles in leaf node
    Triangle triangle[MAX_TRIANGLES_PER_LEAF];      // the cached triangles
} leafCache[NUM_LEAFCACHE_ENTRIES];
```

图 13.7 显示了上述树型结构中叶节点的线性化数据缓存方案，并完全消除了三角形处理过程中的间接操作。

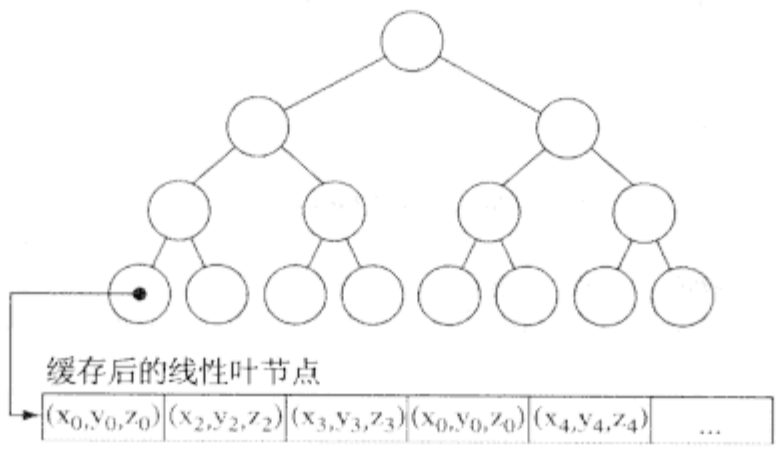


图 13.7　改进后的图 13.6 所显示的结构，并通过缓存方案实现线性化操作，从而消除间接操作以实现顶点的直接访问

因此，针对某一叶节点，可编写相应的代码以查看该节点是否已位于缓存中：若为否，则缓存该节点并随后对当前已缓存的节点数据实施相应的操作。

```
// Compute direct-mapped hash table index from pLeaf pointer (given as input)
int32 hashIndex = ((int32)pLeaf >> 2) & NUM_LEAFCACHE_ENTRIES;
// If leaf not in cache at this address, cache it
CachedLeaf *pCachedLeaf = &leafCache[hashIndex];
if (pCachedLeaf->pLeaf != pLeaf) {
    // Set cache key to be pLeaf pointer
    pCachedLeaf->pLeaf = pLeaf;
    // Fetch all triangles and store in cached leaf (linearized)
    int numTris = pLeaf->numTris;
    pCachedLeaf->numTris = numTris;
    for (int i = 0; i < numTris; i++) {
        Triangle *pCachedTri = &pCachedLeaf->triangle[i];
        IndexedTriangle *pTri = &triangle[pLeaf->tri[i]];
        pCachedTri->p0 = vertex[pTri->i0];
        pCachedTri->p1 = vertex[pTri->i1];
        pCachedTri->p2 = vertex[pTri->i2];
    }
}

// Leaf now in cache so use cached leaf node data for processing leaf node
int numTris = pCachedLeaf->numTris;
```

```
for (int i = 0; i < numTris; i++) {
    Triangle *pCachedTri = &pCachedLeaf->triangle[i];
    DoSomethingWithTriangle(pCachedTri->p0, pCachedTri->p1, pCachedTri->p2);
}
...
```

针对 64 位的缓存单元，在最坏情况下原代码对某一个三角形可能会执行 10 次缓存单元操作，包括：两次叶节点数据结构的缓存单元操作，两次索引三角形数据结构的缓存单元操作，以及两次各顶点的缓存单元操作。另外，即使各数据段实现了数据的填充操作以呈现数据单元的对齐特征，仍需多达 5 次的缓存单元访问操作。而针对缓存中的叶节点，经缓存后的代码基于每一个三角形最多访问两次缓存单元（且通常为一次）。

13.5.2　基于分摊机制的预测线性化缓存

线性化这一概念还可用于更高级的缓存方案中。例如，考查一棵静态树型结构且表示为某一场景世界，并在玩家附近设置一个球体以执行碰撞查询。这里，可将球体空间内的全部多边形数据按照线性化格式加以缓存，一旦玩家出现于球体内部，则只需测试缓存数据，即无须对全部树型结构进行遍历或测试操作。

上述计算过程可采取分摊机制加以实现，从而降低球体内部碰撞图元的线性化及缓存操作的计算开销。当玩家相对于当前缓存球体空间移动一段距离时，则会根据玩家的当前位置展开新的一轮缓存查询，并在几帧之内完成该计算过程（如图 13.8 所示）。待第二个缓存数据集全部计算完毕后，距离玩家最近的球体所对应的数据集将被设置为当前碰撞集合。随后，另一集合将被清零，待玩家相对于球体新原点位置移动一段距离后，将再次按同一方式重复计算上述过程。

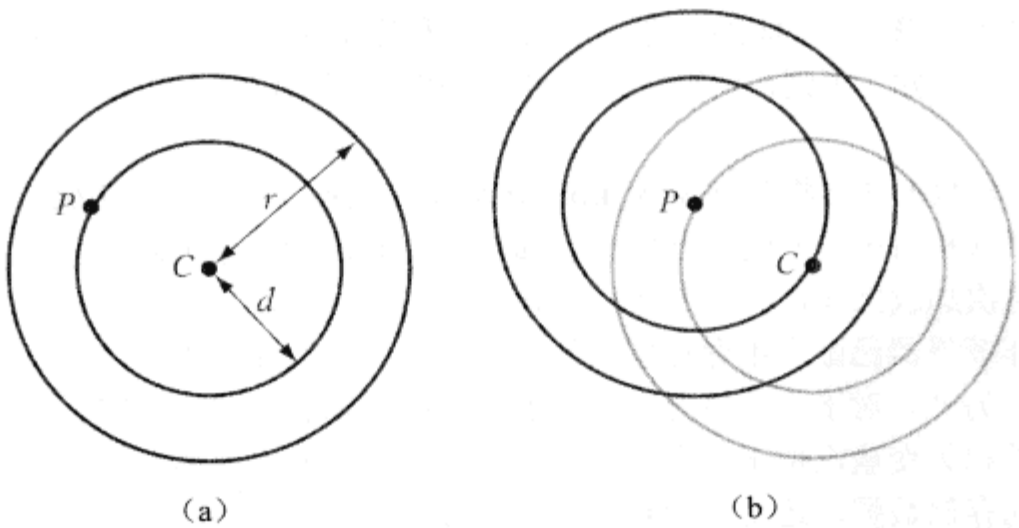


图 13.8　（a）球心位置为 c、半径为 r 的球体并缓存其内部几何体。同时，还需同时维护半径为 r 的同心球体的状态。（b）当玩家将小球体移至位置 P 时，对位于 P 且半径为 r 的球体内部的全部几何体执行分摊缓存操作。另外，缓存操作需在距离 $r-d$ 所占用的时间范围内完成，以确保最新的几何体集合为当前数据集

该方案适用于玩家以中速遍历场景世界这一情形。若玩家在两位置间呈现瞬间移动，则该方案并非最佳。

13.6 数据别名

在 C/C++语言中，令两个指针指向同一内存位置是十分常见的情形。当两个指针指向同一内存位置时，则二者互为别名关系。例如，在下列代码中，指针 **p1** 和 **p2** 即为整型变量 **n** 的别名：

```
int n;
int *p1 = &n;      // *p1 references the location of n
int *p2 = &n;      // *p2 also references the location of n
```

别名问题会对 C/C++编译器产生负面影响。当编译器无法确定二指针变量不为别名关系时，则（保守地）假设该别名关系成立——这一假设最终将阻止某些优操作，使得代码无谓地耗费相应的加载和存储指令，并在函数的堆栈帧中导致代码膨胀且破坏数据的紧凑性，因而间接地影响到代码和数据缓存的执行效率。另外，相应的加载和存储过程还将会妨碍指令的重排序以及循环的展开操作。

相反，其他语言（如 FORTRAN 77）则规定：两个标识符之间不允许出现彼此别名这一现象，除非使用 **EQUIVALENCE** 声明，从而有效地解决了别名问题。为了进一步说明这一现象，下列代码将计算顶点与一个 3 × 3 矩阵之间的转换操作：

```
void TransformPoint(Point *pOut, Matrix &m, Point &in)
{
    pOut->x = m[0][0] * in.x + m[0][1] * in.y + m[0][2] * in.z;
    pOut->y = m[1][0] * in.x + m[1][1] * in.y + m[1][2] * in.z;
    pOut->z = m[2][0] * in.x + m[2][1] * in.y + m[2][2] * in.z;
}
```

这里的编译器并不了解代码的相应功能，因此，从程序员角度来讲，虽然 **pOut** 和 **in** 并非别名关系，但编译器对此一无所知（否则，代码将无法实现正确的转换操作）。因此，编译器假定 **pOut** 和 **in** 将引用同一结构。

针对上述代码，多数编译器将分别加载 **in.x**、**in.y** 和 **in.z** 各 3 次，共计 9 次。而一类基于类型特征的别名分析型编译器（稍后讨论）则会分析出 **pOut->x** 和 **in.x** 为同名关系；**pOut->y** 和 **in.y** 为同名关系。因此只需执行两次加载操作，共计 5 次。而 **in** 中的各字段仍需各加载一次。

一般来讲，除非编译器已证实不存在别名问题，否则，假设该问题一直存在。但是，这一证明过程将是十分困难的。另外，对于相反情况（编译器假设存在别名问题，而实际结果却并非如此）最终结果也不容乐观：若相关变量已位于寄存器中，这将导致额外的内存读取操作。更为糟糕的是，若无法正确地移除无关的存储数据，还将会导致一系列的级联效应，并可能再次产生其他同名问题，从而阻止编译器消除无关的内存访问，依此类推。

这里需要强调的是，将输入变量声明为 **const** 并非是一个解决别名问题的有效方案。例如，将变量 **m** 声明为 **const Matrix &m** 只是表明：在“使用变量 **m**”进行访问时，**m** 所占据的相应内存区域不可改变，但这并不意味着该内存区域不可变，且仍可通过 **pOut** 写入。综上所述，**const** 变量对于编译器的代码优化操作并无太大帮助，其主要用途是辅助检测程序中的错误。

因此，对于相关问题，上述代码中可改写为：

```
void TransformPoint(Point *pOut, Matrix &m, Point &in)
{
    float tmpx, tmpy, tmpz;
    tmpx = m[0][0] * in.x + m[0][1] * in.y + m[0][2] * in.z;
    tmpy = m[1][0] * in.x + m[1][1] * in.y + m[1][2] * in.z;
    tmpz = m[2][0] * in.x + m[2][1] * in.y + m[2][2] * in.z;
    pOut->x = tmpx; pOut->y = tmpy; pOut->z = tmpz;
}
```

其中，输入数据将在内存写入之前得到访问，并且同一变量既可作为输入变量使用，也可作为输出变量使用，这就有效地避免了同名问题。另外，一款较好的编译器应该可对上述代码进行相关的优化操作。同时，输入数据使用在前、输出变量生成在后这一方式对于某些架构机可提供更好的计算方案。特别地，对于包含足够数量的多用途输入寄存器的类 RISC 机而言，尤是如此。

应尽可能地采用上述编程风格（使用局部变量加载全局变量、引用值以及临时变量）以辅助编译器的相关优化操作，亦即，可显式地帮助编译器识别已排除的同名问题。然而，在某些情况下，该方案并非总是能够得以顺利执行。例如，基于 C++语言的类数据封装、抽象概念以及模板机制，代码通常会被分解为较小的通用子模块，并隐藏了相应的内部结构和操作。因此，诸如在数据结构访问过程中生成一个缓冲指针的局部变量备份，或者将恒等表达式移出循环体之外这一类的简单操作，在数据结构封装的环境下或仅通过展开模板实例一般难以实现。

因此，数据抽象机制往往会对优化操作带来一定的负面影响，尤其是别名问题。这一现象也称作抽象惩罚问题，进一步的讨论可参见[Stepanov]、[Robison96a]以及[Robison96b]。另外，针对 C++语言抽象惩罚所导致的计算性能问题，特别是与向量和矩阵表达式相关的计算开销问题，可参考[Veldhuizen95]中给出的模板表达式技术。

13.6.1 基于类型的别名分析

某些现代 C/C++编译器则采用了基于类型的别名分析机制以辅助解决别名问题，这一类编译器遵循下列相关语言标准：仅同一兼容类型的两个对象之间彼此允许出现别名现象，并且每一块内存区域在其生存期内只可与一种类型的变量相关联。

例如，在上述标准规定之下，浮点数和整数则为互不兼容的数据类型。考虑到数据块的移动以及类似操作的正确执行，有符号和无符号字符型数据则全兼容于其他类型的数据。根据上述兼容规则，一些添加了基于类型的别名分析机制的高级编译器能够对下列内容进行有效的判断：

```
void Foo(float *v, int *n)
{
    for (int i = 0; i < *n; i++)
        v[i] += 1.0f;
}
```

即 **v[i]**和***n** 之间不存在别名问题并且可进一步优化为：

```
void Foo(float *v, int *n)
{
    int t = *n;
    for (int i = 0; i < t; i++)
        v[i] += 1.0f;
}
```

二者间的差异并不大，但这一差异却十分重要。第一段代码未曾使用基于类型的别名分析，因此，***n** 将根据循环次数进行多次内存读取操作；在第二段代码中，***n** 则移出了循环体，因此只需读取一次。需要注意的是，根据 C/C++语言标准，下列类型转换代码是合法的：

```
uint32 i;
float f;
i = *((int32 *)&f);    // Illegal way of getting the integer representation of the float f
```

该操作将 **f** 所占据的内存区域视为两种不同类型的数据块。针对上述代码，支持基于类型的别名分析的编译器其生成代码将变得不确定。由于编译器一般能够较好地解决联合结构中的别名问题，因此多数编译器将其视为一种有效的访问途径，如下所示：

```
union {
    uint32 i;
    float f;
} u;
u.f = f;
uint32 i = u.i;
```

然而，为了更好地配合相关语言标准并生成正确的代码，该浮点数应通过 **char** 数组加以访问，并在其元素中辅以 **uint32** 类型数据。

13.6.2 restrict 指针

ANSI/ISO C（1999）语言标准新增了一种关键字 **restrict**，并允许程序员部分地辅助编译器处理程序中的别名问题。其结果为，编译器可执行进一步的代码优化操作。另外，某些 C++编译器也提供了相应的 **restrict** 关键字作为其语言扩展。利用 **restrict** 修饰符声明一个指针（或引用）意即向编译器表明：在当前指针作用域内，其指向的目标区域只能通过该指针（或其备份）加以访问。采用受限指针或引用后，上述顶点转换代码可按如下方式加以改写：

```
void TransformPoint(Point * restrict pOut, Matrix & restrict m, Point & restrict in)
{
    pOut->x = m[0][0] * in.x + m[0][1] * in.y + m[0][2] * in.z;
    pOut->y = m[1][0] * in.x + m[1][1] * in.y + m[1][2] * in.z;
    pOut->z = m[2][0] * in.x + m[2][1] * in.y + m[2][2] * in.z;
}
```

针对该函数，一款较好的编译器将能够生成较为优化的代码：该编译器能够判断出写入 **pOut** 不会对输入数据产生任何影响。由于忽略 **restrict** 关键字并不违背语言标准，因此在使用该关键字之前检测

目标编译器是否真正支持该功能提示以实现代码优化将是一个不错的选择。这里，为了进一步演示 **restrict** 指针的优点，考查两个复数的乘法操作$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$，其直接实现方式如下所示：

```
void ComplexMult(float *a, float *b, float *c)
{
    a[0] = b[0]*c[0] - b[1]*c[1];      // real part in a[0] (b[0] and c[0])
    a[1] = b[0]*c[1] + b[1]*c[0];      // imaginary part in a[1] (b[1] and c[1])
}
```

上述函数可按下列 3 种方式加以调用：

```
float f[3];
ComplexMult(&f[0], &f[0], &f[0]);    // a[0] aliases b[0], c[0]; a[1] aliases b[1], c[1]
ComplexMult(&f[1], &f[0], &f[0]);    // a[0] aliases b[1], c[1]
ComplexMult(&f[0], &f[1], &f[1]);    // a[1] aliases b[0], c[0]
```

显然，**a[0]**和 **a[1]**将会与 **b[0]**、**b[1]**、**c[0]**或 **c[1]**产生别名问题。另外，**a[0]**的赋值操作将使得全部 4 个输入值在第二行代码中再次被读取。为了较好地演示该问题，下列编译代码（这里，采用了 gcc 编译器并生成 MIPS 汇编代码）解释了这一情况：

```
ComplexMult(float *, float *, float *)
lwc1    f3,0x0000(a2)     ; f3 = c[0]
lwc1    f2,0x0004(a2)     ; f2 = c[1]
lwc1    f1,0x0000(a1)     ; f1 = b[0]
lwc1    f0,0x0004(a1)     ; f0 = b[1]
mul.s   f1,f1,f3          ; f1 = f1 * f3
mul.s   f0,f0,f2          ; f0 = f0 * f2
sub.s   f1,f1,f0          ; f1 = f1 – f0
swc1    f1,0x0000(a0)     ; a[0] = f1
lwc1    f2,0x0004(a1)     ; f2 = b[1] (reloaded)
lwc1    f3,0x0000(a2)     ; f3 = c[0] (reloaded)
lwc1    f0,0x0000(a1)     ; f0 = b[0] (reloaded)
lwc1    f1,0x0004(a2)     ; f1 = c[1] (reloaded)
mul.s   f2,f2,f3          ; f2 = f2 * f3
mul.s   f0,f0,f1          ; f0 = f0 * f1
add.s   f0,f0,f2          ; f0 = f0 + f2
jr      ra
swc1    f0,0x0004(a0)     ; a[1] = f0
```

通过对变量 **a** 采用 **restrict** 修饰（为了使编译器进一步发挥其别名分析功能，这里也对 **b** 和 **c** 采用了 **restrict** 限制），将提示编译器 **b** 和 **c** 均不会与 **a** 产生别名问题，如下所示：

```
void ComplexMult(float * restrict a, float * restrict b, float * restrict c)
{
    a[0] = b[0]*c[0] - b[1]*c[1];      // real part
    a[1] = b[0]*c[1] + b[1]*c[0];      // imaginary part
}
```

因此，4 个输入变量均读取一次。参见下列函数编译代码以确定这一结论：

```
ComplexMult(float *, float *, float *)
lwc1    f2,0x0004(a1)     ; f2 = b[1]
lwc1    f0,0x0004(a2)     ; f0 = c[1]
lwc1    f1,0x0000(a1)     ; f1 = b[0]
mul.s   f4,f2,f0          ; f4 = f2 * f0
lwc1    f3,0x0000(a2)     ; f3 = c[0]
mul.s   f0,f1,f0          ; f0 = f1 * f0
mul.s   f2,f2,f3          ; f2 = f2 * f3
mul.s   f1,f1,f3          ; f1 = f1 * f3
add.s   f0,f0,f2          ; f0 = f0 + f2
sub.s   f1,f1,f4          ; f1 = f1 – f4
swc1    f0,0x0004(a0)     ; a[1] = f0
jr      ra
swc1    f1,0x0000(a0)     ; a[0] = f1
```

同时，该编译代码还显示了源代码的重组方式，即对相关的两组计算采用了指令交错。考虑到指令延迟，通常无法即刻获取相应的指令结果，因此，指令的重新排序将有助于降低相关指令间的时间延迟。

13.6.3 避免别名问题

由于别名问题通常会对程序的性能产生较大的负面影响，因此有必要对代码及其编写风格进行相应的调整以减少这一问题的出现。下列内容可供读者参考：

- ☑ 将全局变量、函数形式参数中的指针和引用变量缓存至局部变量中。在 C++语言中，由于 **this** 指针所导致的别名问题，即使是类成员也会受惠于这一类局部缓存方式。从优化角度来讲，类成员的行为更接近于全局变量（仍存在一定的作用域），因而也可采用同样的方式加以处理。
- ☑ 尽量避免使用局部变量的地址（**&**符号）。这将消除局部变量的指针，因而也就不会出现别名问题。
- ☑ 对于小型数值，应按值传递参数而非按指针或引用的方式传递。这里，小型数值是指能够较好地配合机器寄存器的数据。在某些情况下，甚至可优先按值传递结构（这类结构通常较小且只包含少数字段）。
- ☑ 将指针或引用声明为 **restrict** 格式。从理论上讲，仅限制输出指针已经足够了。在实际应用中，由于可随意地在指针间进行赋值操作，编译器有时可能会无法及时地跟踪指针数据，即出现指针丢失这一情况。在这种情况下，除非两个指针皆采用 **restrict** 加以修饰限定，否则编译器假设其针对同一目标区域将会出现别名问题。因此，限定全部指针将会简化相应的别名分析过程，并更有可能生成最优结果。需要注意的是，仅当确保不会出现别名问题或编译代码无法正确地运行时，才可考虑采用上述方案。
- ☑ 变量的声明和使用之间应尽量靠近（即尽量推迟声明）。在声明和使用之间的代码越少，别名问题以及编译器别名分析失败的几率也就越小。另外，应及时定义新的临时变量而不要在同一作用域内继续使用原临时变量。
- ☑ 按内联方式定义小型函数，特别是这一类函数使用引用参数时。
- ☑ 应尽量从 **if** 语句和循环体（包括循环条件）中提取出某些公共子表达式，编译器通常不会自

动完成这一类工作。

☑ 应避免过度的抽象，从而将抽象惩罚降低至最低点。

关于别名问题、抽象惩罚问题、**restrict** 指针以及其他类似问题，其有价值的信息并不多见。[Robison99]和[Mitchell00]对此进行了相关探讨。

13.7 采用 SIMD 优化的并行操作

当前，CPU 均包含相应的多重处理单元。通常情况下，CPU 内建运算逻辑单元（ALU）和浮点处理单元，分别执行整数运算以及浮点运算。这一类处理单元一般趋于并行操作，即 ALU 在执行整型指令的同时，FPU 将执行浮点指令。某些 CPU 实现为 **n**-路超标量体系结构，这意味着 CPU 将包含 **n** 个并行执行单元（通常为 ALU），并在一个时钟周期内最多处理 **n** 条指令。

另外，当前众多微处理器均在其基本指令集中涵盖了某种类型的 SIMD 扩展指令（单指令，多数据），这一类 SIMD 指令通常定义于宽寄存器之上——包含 4 位、8 位甚至是 16 位标量整型数据或浮点值——并以并行方式实现多数据单指令计算。因此，相关指令集中的 SIMD 指令可潜在提升 4 倍（8 倍或 16 倍）的代码计算性能。

目前，SIMD 扩展指令包括 Intel 的 SSE 和 SSE2，AMD 的 3DNow!以及 Motorola 和 IBM 的 AltiVec。SONY 公司的 PlayStation2 则将并行机制提升到一个新的水平：基于其 MIPS CPU 核心（两个 ALU 单元和一个 FPU 单元）提供了多媒体扩展指令；同时，还包含两个独立的向量协处理器且实现了与 CPU 的并行操作。另外，二者均支持并行指令流。综上所述，PlayStation2 平台可同时执行多达 7 条不同的指令。诸如 PlayStation3 和 Xbox2 这一类新型的游戏机系统则提供了更强大的并行机制。下面的内容将集中探讨 SIMD 指令的具体应用，并从两方面对顺序执行的代码进行优化处理：

☑ 指令级并行操作。通过确定类似的操作并以并行方式计算，可对现有算法实现优化操作。例如，点积计算中的乘法操作可通过单一的 SIMD 乘法指令实现并行计算。考虑到计算过程的诸多因素，处理过程中的某些部分有时则无法采用指令级的并行机制实现相应的优化操作，例如点积计算中部分乘积结果的求和操作。

☑ 数据级并行操作。SIMD 指令还可针对完全不同的数据实现并行操作。例如，可并行实现 4 个点积计算。考虑到 SIMD 的命名含义，从本质上讲，数据级并行机制正好体现了 SIMD 指令的实际操作过程。

SIMD 优化操作的一个有效应用示例是在交互光线跟踪计算中对光线相交测试进行加速计算。这里将采用 SIMD 指令以并行的方式针对一个三角形测试 4 条光线（[Wald01]）。其中，4 条光线（2×2 光线簇）将同步遍历一棵 *k*-d 树。同时，遍历方案也将采取 SIMD 指令以并行方式执行。如果至少一条光线与子树定义的空间相交，则该子树将被访问。虽然该遍历方案将会带来某些额外的计算开销，但考虑到 2×2 光线簇之间的一致性特征，这类计算开销在实际应用中是无法避免的。然而，这一类簇查询方案一般并不适用于跨度较大的碰撞检测计算。与非 SIMD 的 C 语言测试版本相比，SIMD 三角形测试将在计算时间上带来 3.5～3.7 倍的提升，而包含 4 条光线的簇遍历方案将会再次将计算速度提升约 2 倍。

编译器对 SIMD 指令的支持尚待进一步开发。这里，编译器对 SIMD 指令的支持主要由相关内建

函数构成（映射为对应的汇编指令）。因此，有效地使用 SIMD 指令仍然离不开汇编语言的支持。

为了进一步演示 SIMD 指令在碰撞检测计算中的应用，后续小节将逐一探讨球体-球体、球体-AABB 以及 AABB-AABB 之间的测试。其中，假设 SIMD 机制可在同一寄存器中加载 4 个（浮点）数据值，并可对其实现单指令同步操作。另外，关于光线-盒体以及光线-三角形测试，可参见[Knittel00]和[Bonnedal02]。

需要注意的是，对于缺少 SIMD 指令扩展支持的机器而言，同样能够实现数据级的并行操作，相关实例可参见 13.4.1 节中的 **PopCount8()**函数。

13.7.1 4 球体-4 球体 SIMD 测试

这里，分别定义 8 个球体 S_i，$1 \leqslant i \leqslant 8$ 且每一个球体均按(x_i,y_i,z_i,r_i)4 元格式加以定义。其中，(x_i,y_i,z_i)为球心坐标，r_i 为半径。4 球体测试如下所示：

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 \leqslant (r_i + r_j)^2$$

其中，$1 \leqslant i \leqslant 4$，$j = i + 4$。该测试可采用 SIMD 指令实现并行操作。首先，定义 8 个 SIMD 寄存器。例如，寄存器 **PX** 包含 4 个值 x_1，x_2，x_3，x_4。同理，其他 7 个寄存器如下所示：

```
PX = x1 | x2 | x3 | x4        QX = x5 | x6 | x7 | x8
PY = y1 | y2 | y3 | y4        QY = y5 | y6 | y7 | y8
PZ = z1 | z2 | z3 | z4        QZ = z5 | z6 | z7 | z8
PR = r1 | r2 | r3 | r4        QR = r5 | r6 | r7 | r8
```

4 球体测试可按下列 SIMD 汇编伪代码实现并行操作：

```
SUB T1,PX,QX        ; T1 = PX - QX
SUB T2,PY,QY        ; T2 = PY - QY
SUB T3,PZ,QZ        ; T3 = PZ - QZ（T1-3 为球心之差）
ADD T4,PR,QR        ; T4 = PR + QR（T4 为半径之和）
MUL T1,T1,T1        ; T1 = T1 * T1
MUL T2,T2,T2        ; T2 = T2 * T2
MUL T3,T3,T3        ; T3 = T3 * T3（T1-3 为球心之间的平方距离）
MUL R2,T4,T4        ; R2 = T4 * T4（R2 为半径和的平方）
ADD T1,T1,T2        ; T1 = T1 + T2
SUB T2,R2,T3        ; T2 = R2 - T3
LEQ Result,T1,T2    ; Result = T1 <= T2
```

最终代码仅需要 11 条指令。考虑到一条指令在 1/4 时钟周期内的输出/延迟，最终结果可在 16/19 个时钟周期内获得，因此，针对单一球体-球体测试，代码计算则只需 4 个时钟周期。

13.7.2 4 球体-4AABB SIMD 测试

根据上述内容，球体-AABB 之间的测试也可采用类似的方式加以计算。如前所述，球体按(x_i,y_i,z_i,r_i)4 元格式加以定义；AABB 则根据对角最小值($min[i]x$, $min[i]y$, $min[i]z$)和最大值($max[i]x$, $max[i]y$, $max[i]z$)加以定义。另外，SIMD 寄存器按如下方式定义：

```
MINX  = min1x | min2x | min3x | min4x     SX  = x1 | x2 | x3 | x4
MINY  = min1y | min2y | min3y | min4y     SY  = y1 | y2 | y3 | y4
MINZ  = min1z | min2z | min3z | min4z     SZ  = z1 | z2 | z3 | z4
MAXX  = max1x | max2x | max3x | max4x     SR  = r1 | r2 | r3 | r4
MAXY  = max1y | max2y | max3y | max4y
MAXZ  = max1z | max2z | max3z | max4z
```

该测试的伪代码如下所示：

```
MAX TX,SX,MINX        ; TX = Max(SX,MINX)      计算 AABB 之上/内的、距球心 S 的最近顶点
MAX TY,SY,MINY        ; TY = Max(SY,MINY)      T = (TX, TY, TZ)。可将球心限定在 AABB 的
MAX TZ,SZ,MINZ        ; TZ = Max(SZ,MINZ)      范围之内。
MIN TX,TX,MAXX        ; TX = Min(TX,MAXX)
MIN TY,TY,MAXY        ; TY = Min(TY,MAXY)
MIN TZ,TZ,MAXZ        ; TZ = Min(TZ,MAXZ)
SUB DX,SX,TX          ; DX = SX - TX           D = S - T 表示 S 和球心 T 之间的向量。
SUB DY,SY,TY          ; DY = SY - TY
SUB DZ,SZ,TZ          ; DZ = SZ - TZ
MUL R2,SR,SR          ; R2 = SR * SR           最终，Result = Dot(D, D) <= SR^2，其中，SR 为球
MUL DX,DX,DX          ; DX = DX * DX           心半径（为了降低点积中两个连续加法操作产生的
MUL DY,DY,DY          ; DY = DY * DY           延迟，可将 DZ^2 移至不等式的右侧，并与 SR^2
MUL DZ,DZ,DZ          ; DZ = DZ * DZ           进行减法操作）。
ADD T1,DX,DY          ; T1 = DX + DY
SUB T2,R2,DZ          ; T2 = R2 - DZ
LEQ Result,T1,T2      ; Result = T1 <= T2
```

上述代码包含了 16 条指令且一条指令的输出/延迟为 1/4 个时钟周期，则最终结果耗时 23/26 个时钟周期。

13.7.3　4AABB-4AABB SIMD 测试

对于 AABB-AABB 之间的并行计算，可通过最小值($min[i]x$, $min[i]y$, $min[i]z$)和最大值($max[i]x$, $max[i]y$, $max[i]z$)定义 8 个 AABB，$1 \leqslant i \leqslant 8$。SIMD 寄存器如下所示：

```
MIN1X  = min1x | min2x | min3x | min4x   MIN2X  = min5x | min6x | min7x | min8x
MIN1Y  = min1y | min2y | min3y | min4y   MIN2Y  = min5y | min6y | min7y | min8y
MIN1Z  = min1z | min2z | min3z | min4z   MIN2Z  = min5z | min6z | min7z | min8z
MAX1X  = max1x | max2x | max3x | max4x   MAX2X  = max5x | max6x | max7x | max8x
MAX1Y  = max1y | max2y | max3y | max4y   MAX2Y  = max5y | max6y | max7y | max8y
MAX1Z  = max1z | max2z | max3z | max4z   MAX2Z  = max5z | max6z | max7z | max8z
```

该测试的伪代码如下所示：

```
MAX AX,MIN1X,MIN2X      ; AX = Max(MIN1X,MIN2X)      计算两个包围盒的相交空间：
MIN BX,MAX1X,MAX2X      ; BX = Min(MAX1X,MAX2X)      利用最小范围内的最大值和最
MAX AY,MIN1Y,MIN2Y      ; AY = Max(MIN1Y,MIN2Y)      大范围内的最小值。
MIN BY,MAX1Y,MAX2Y      ; BY = Min(MAX1Y,MAX2Y)
MAX AZ,MIN1Z,MIN2Z      ; AZ = Max(MIN1Z,MIN2Z)
MIN BZ,MAX1Z,MAX2Z      ; BZ = Min(MAX1Z,MAX2Z)
```

```
LEQ T1,AX,BX        ; T1 = AX <= BX      如果存在有效空间（两个 AABB 相交），
LEQ T2,AY,BY        ; T2 = AY <= BY      则经计算后，最小范围定小于最大范围。
LEQ T3,AZ,BZ        ; T3 = AZ <= BZ
AND T4,T1,T2        ; T4 = T1 && T2
AND Result,T3,T4    ; Result = T3 && T4
```

上述代码包含 11 条指令且一条指令的输出/延迟为 1/4 个时钟周期，则最终结果耗时 16/19 个时钟周期。

13.8 分支结构

对于 CPU 而言，分支结构同样能够导致性能问题。一条运行中的指令将分为多个步骤加以执行，其原因仍归结于并行机制。下列抽象体系结构包含了 4 个相应步骤：

1. 指令读取。在该步骤中，将实现指令的内存读取操作。
2. 解码操作。此时，将检测指令中的相关数据位并将相应控制发往该指令的实际执行位置。
3. 执行指令。执行指令所规定的实际操作。
4. 存储结果。指令结构将存储至相关寄存器或内存中。

其中，各执行步骤均消耗一个时钟周期，因此，一条指令执行完毕后将消耗 4 个周期。待前一条指令执行完毕、后一条指令开始之时，指令间将间隔 4 个时钟周期。然而，由于指令的执行过程包含了多个处理阶段，因此可将各操作阶段彼此重叠以实现流水线作业。因此，可每隔一个时钟周期开始执行一条新增指令。图 13.9 比较了多个指令间的非流水线和流水线操作（I1～I8），其中，非流水线处理器执行了一条指令，与此同时，流水线处理器则执行了 4 条指令。

图 13.9　非流水线指令的执行过程（上），以及流水线指令的执行过程（下）

显然，若将指令细分解为多个快速处理阶段，则流水处理器将能够以较高的速率处理更多条指令。考虑到速度提升所带来的收益，某些 PC 中的 CPU 内建了多达 20 级的流水线。

分支结构将会对基于流水线操作的 CPU 体系结构带来某些负面影响。如图 13.10 所示，其中，若

不考虑分支结构 I1 后的 I2 指令且已知 I3 为另一分支结构，则须停止后续指令的分支目标，直至当前分支结构执行完毕。分支结构产生的延迟将会在流水线中产生“气泡”效应，从而降低流水线的计算性能。

时钟周期	1	2	3	4	5	6	7	8
取指操作	I1	I2	—	—	I3	I4	I5	I6
解码	...	I1				I3	I4	I5
执行	...	...	I1				I3	I4
存储	...	...	...	I1				I3

图 13.10　针对分支指令 I1，消除指令 I2 后所导致的流水线延迟

为了保证流水线的持续运行，CPU 将会适当地预测分支目标（在当前分支处考查之前的分支历史，并通过特定的分支缓存进行跟踪记录）并假定结果正确。如果预测失效，则清空该流水线（清空全部预测分支目标）并重启流水线。对于一条包含 20 级的流水线，将容纳约 20 条指令并随后填充该流水线的各级，因此，流水线将再次满负载运行。

应尽量避免流水线的清空操作，从而改善流水线的计算性能。首先，应对相关代码重新加以设计，以减少不必要的分支结构；其次，还应对代码进行适当的重组（特别是 if 语句及其他判断条件），以使控制流更具预测性。

对代码重新设计常涉及布尔值的移位计算。同时，还包括强制编译器生成某些特定的 CPU 预测指令。当然，这些指令的执行状况将依赖于先前计算控制标志的状态。当前，多数 CPU 均在其指令集中支持预测指令。另外，预测指令的使用将因编译器而异，因此，有必要对相关生成代码进行检测以了解其发布环境。

关于消除分支结构的详细讨论则超出了本书的范围。这里，仅给出一个消除分支结构的相关示例。下列代码将判断某值是否为前 5 个素数之一：

```
uint32 SmallPrime(uint32 x)
{
    if ((x == 2) || (x == 3) || (x == 5) || (x == 7) || (x == 11))
        return 1;
    return 0;
}
```

针对上述代码，多数编译器将生成包含 5 个分支结构的汇编代码。例如，PC 上的 Microsoft Visual C 会将该函数编译为下列汇编代码：

```
00401150  mov   eax,dword ptr [esp+4]      ; fetch x
00401154  cmp   eax,2                      ; if (x == 2) goto RETURN_1
00401157  je    00401170
00401159  cmp   eax,3                      ; if (x == 3) goto RETURN_1
0040115C  je    00401170
0040115E  cmp   eax,5                      ; if (x == 5) goto RETURN_1
00401161  je    00401170
00401163  cmp   eax,7                      ; if (x == 7) goto RETURN_1
00401166  je    00401170
00401168  cmp   eax,11                     ; if (x == 11) goto RETURN_1
```

```
0040116B je      00401170
0040116D xor     eax,eax                    ; return 0
0040116F ret
00401170 mov     eax,1                      ; RETURN_1: return 1
00401175 ret
```

可以看出，上述代码中几乎每隔一行便出现一个分支结构，因此该代码的流水线特征较差。此处，可将短路 OR 操作符（||）替换为位 OR 操作符（|）以消除分支结构。但这样一来，最终生成的编译代码将变得十分冗长，因而简单地使用 OR 位操作符并不能完全解决相关问题。较好的方法是使用一个内置于某一常数的位查找表，其中，通过设定该常数的 2、3、5、7 以及 11 位，与 **x** 对应的相应位将包含正确的返回结果值。当采用 C/C++进行编码时，唯一复杂之处在于：当右操作数大于左操作数位数时，未定义相应的移位操作符。针对该问题，考查下列分支优化代码：

```
uint32 SmallPrime(uint32 x)
{
    const uint32 PRIME_BITS = (1 << 2) | (1 << 3) | (1 << 5) | (1 << 7) | (1 << 11);
    uint32 mask = (x < 32);
    return (PRIME_BITS >> x) & mask;
}
```

该函数的编译结果如下所示：

```
00401180 mov     ecx,dword ptr [esp+4]      ; fetch x
00401184 mov     edx,8Ach                   ; PRIME_BITS = ...
00401189 cmp     ecx,32                     ; x < 32
0040118C sbb     eax,eax
0040118E shr     edx,cl                     ; edx = PRIME_BITS >> "x"
00401190 neg     eax                        ; eax = mask
00401192 and     eax,edx                    ; return (PRIME_BITS >> x) & mask
00401194 ret
```

与前一结果相比，上述汇编代码更加简洁并体现了一种较好的流水线特征，从而有效地消除了分支结构。对于时间要求严格的代码而言，可通过不同方案消除多数分支结构，这里所探讨的内容便是一例。

13.9 小结

本书各个章节主要讨论了碰撞检测系统（以及其他相关领域）中相关数据结构和算法的高效实现方案，除此之外，直接对碰撞检测系统运行平台的硬件体系结构进行限定、调整同样可获得巨大的性能提升。针对现代体系结构，本章集中讨论了两类性能提升方案——内存优化方案以及并行优化方案。

与内存读取速度相比，CPU 在数据处理速度方面明显占优。因此，通常在 CPU 和主存之间设置多级缓存单元，以加速数据的传输速度。适当地利用缓存机制，可将数据的传输速度提高 1～3 个数量级。本章讨论了多种代码和数据的缓存优化方案，包括基于缓存感知数据结构和算法的代码以及数据重组方案。同时，本章阐述了 C/C++代码中别名现象对内部优化操作所产生的影响——这将导致编译代码出

现冗余的内存访问操作。本章还详细分析了消除别名问题的多种方法。

本章关于并行机制优化操作的讨论主要集中于 SIMD 扩展指令方面。目前，多数 CPU 均支持 SIMD 扩展指令，因此，指令可以并行方式操作于 4、8 甚至是 16 位数据格式之上。基于此，相关测试（如图元-图元之间的相交测试）可分别获得约 4、8 或者 16 倍的性能提升。

本章最后还介绍了分支结构。对于包含 20 级的 CPU 操作流水线而言，无效的预测分支最终将导致大量的计算开销。对性能要求严格的代码而言，应在内部循环中避免这一类分支结构的出现，这一问题可采用相应的位操作技巧加以解决。

参 考 文 献

[Abrams98] Abrams, Stephen. Wonjoon Cho. Chun-Yi Hu. Takashi Maekawa. Nicholas Patrikalakis. Evan Sherbrooke. XiuziYe."Efficient and Reliable Meth-ods for Rounded Interval Arithmetic," MIT Design Laboratory Memorandum 97-9. 1998.

[Acton96] Acton, Forman. Real Computing Made Real. Princeton University Press, 1996.

[Akenine-Möller01] Akenine-Möller, Tomas. "Fast 3D Triangle-box Overlap Testing,"Journal of Graphics Tools, vol. 6, no. 1, pp. 29～33, 2001.

http://www.ce.chalmers.se/staff/tomasm/pubs/tribox.pdf

[Akenine-Möller02] Akenine-Möller, Tomas. Eric Haines. Real-Time Rendering(2d ed.) A K Peters, 2002.

http://www.realtimerendering.com/

[Amanatides87] Amanatides, John. Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing." Proceedings of Eurographics 1987, pp. 3～10, 1987.

http://www.cs.yorku.ca/～amana/research/grid.pdf

[Amanatides97] Amanatides, John. Kia Choi."Ray Tracing Triangular Meshes." Proceedings of the Eighth Western Computer Graphics Symposium, pp. 43～52, April 1997.

http://www.cs.yorku.ca/～amana/research/mesh.pdf

[Andrew79] Andrew, Alex. "Another Efficient Algorithm for Convex Hulls in Two Dimensions." Information Processing Letters, vol. 9, no. 5, pp. 216–219, 1979.

http://dx.doi.org/10.1016/0020-0190(79)90072-3

[Anton00] Anton, Howard. Elementary Linear Algebra (8th ed.). John Wiley & Sons, 2000.

[Arvo90] Arvo, James. "Transforming Axis-aligned Bounding Boxes," in Andrew Glassner (ed.), Graphics Gems, Academic Press, pp. 548～550, 1990.

http://www.graphicsgems.org/

[Arvo91] Arvo, James (ed.). Graphics Gems II, Academic Press, 1991.

http://www.graphicsgems.org

[Avnaim95] Avnaim, Francis. Jean-Daniel Boissonnat. Olivier Devillers. Franco Preparata. Mariette Yvinec. "Evaluation of a New Method to Compute Signs of Determinants," Proceedings of the Eleventh Annual Symposium on Computational Geometry, pp. 416～417, 1995.

http://www-sop.inria.fr/prisme/publis/abdpy-enmcs-95.ps.gz

[Baciu99] Baciu, George. Wingo Wong. Hanqiu Sun. "RECODE: An Image-based Collision Detection Algorithm," Journal of Visualization and Computer Animation, vol. 10, no. 4, pp. 181～192, 1999.

[Baraff92] Baraff, David."Dynamic Simulation of Non-penetrating Rigid Bodies," Ph.D. thesis,

Technical Report 92-1275, Computer Science Department, Cornell University, 1992.

http://www.pixar.com/companyinfo/research/deb/

[Barber96] Barber, C. Bradford. David Dobkin. Hannu Huhdanpaa. "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, vol. 22, no. 4, pp. 469～483, 1996.

http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/

[Barequet96] Barequet, Gill. Bernard Chazelle. Leonid Guibas. Joseph Mitchell. Ayellet Tal. "BOXTREE: A Hierarchical Representation for Surfaces in 3D,"Computer Graphics Forum, vol. 15, no. 3, pp. 387～396, 1996.

ftp://ftp.ams.sunysb.edu/pub/geometry/boxtree.ps.gz

[Barequet97] Barequet, Gill. Subodh Kumar."Repairing CAD Models,"Proceedings of the 8th Conference on Visualization 1997 (VIS 97), pp. 363～370, October 1997.

ftp://ftp.cs.technion.ac.il/pub/barequet/papers/unv-vis97.ps.gz

[Barequet99] Barequet, Gill. Sariel Har-Peled. "Efficiently Approximating the Minimum-volume Bounding Box of a Point Set in Three Dimensions,"Proceedings of theTenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 82～91,1999.

http://valis.cs.uiuc.edu/～sariel/research/papers/98/bbox.pdf

[Barrera03] Barrera, Tony. Anders Hast. Ewert Bengtsson. "A Fast and Simple All-integer Parametric Line,"in Jeff Lander (ed.), Graphics Programming Methods. Charles River Media, 2003.

[Baumgart72] Baumgart, Bruce. "Winged Edge Polyhedron Representation,"Artificial Intelligence Project Memo AIM-179 (CS-TR-72-320), Computer Science Department, Stanford University, 1972.

ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/72/320/CS-TR-72-320.pdf

[Bekker01] Bekker, Henk. Jos Roerdink. "An Efficient Algorithm to Calculate the Minkowski Sum of Convex 3D Polyhedra,"in Computational Science – ICCS 2001, Lecture Notes in Computer Science 2073, Springer-Verlag, pp. 619～628, 2001.

http://www.cs.rug.nl/～roe/publications/ccs3.ps.gz

[Belov02] Belov, Gleb. "A Modified Algorithm for Convex Decomposition of 3D Polyhedra,"Technical report MATH-NM-03-2002, Institut für Numerische Mathematik, Technische Universität, Dresden. March 2002.

http://www.math.tu-dresden.de/～belov/cd3/cd3.ps

[Bentley75] Bentley, Jon."Multidimensional Binary SearchTrees Used for Associative Searching," Communications of the ACM, vol. 18, no. 9, pp. 509～517,1975.

http://www.acm.org/pubs/articles/journals/cacm/1975-18-9/p509-bentley/p509-bentley.pdf

[Berg00] de Berg, Mark. Marc van Kreveld. Mark Overmars. Otfried Schwarzkopf. Computational Geometry (2d ed.). Springer Verlag, 2000.

[Bergen97] van den Bergen, Gino. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees,"Journal of Graphics Tools, vol. 2, no. 4, pp.1～14, 1997.

http://www.win.tue.nl/～gino/solid/jgt98deform.pdf

[Bergen99] van den Bergen, Gino. "Collision Detection in Interactive 3D Computer Animation," Ph.D.

thesis, Eindhoven University of Technology, Eindhoven, 1999.

[Bergen03] van den Bergen, Gino. Collision Detection in Interactive 3D Environments. Morgan Kaufmann Publishers, 2003.

[Bergen04] van den Bergen, Gino. "Ray Casting against General Convex Objects with Application to Continuous Collision Detection." Submitted to Journal of Graphics Tools, 2004.

http://www.dtecta.com/papers/jgt04raycast.pdf

[Binstock96] Binstock, Andrew. "Hashing Rehashed," Doctor Dobbs Journal, April 1996.

[Bonnedal02] Bonnedal, Rasmus. Magnus Pettersson."SIMD Accelerated Ray Tracing," Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, 2002.

http://www. ce.chalmers.se/undergraduate/Theses/Exjobbsrapporter/973_2002.pdf

[Botkin] Botkin, Nikolai. "Randomized Algorithm for Solving Quadratic Programs."

http://citeseer.ist.psu.edu/108615.html

[Botkin95] Botkin, Nikolai. "Randomized Algorithms for the Separation of Point Sets and for Solving Quadratic Programs,"Applied Mathematics and Optimization, vol. 32, pp. 195～210, 1995.

http://citeseer. ist.psu.edu/87315.html

[Boyd04] Boyd, Stephen. Lieven Vandenberghe. Convex Optimization. Cambridge University Press, 2004.

http://www.stanford.edu/～boyd/cvxbook.html

[Boyles99] Boyles, Michael. Shiaofen Fang. "Slicing-based Volumetric Collision Detection," Journal of Graphics Tools, vol. 4, no. 4, pp. 23～32, 1999.

http://www.avl.iu.edu/～mjboyles/papers/vcd.ps

[Capens01] Capens, Nicolas. "Smallest Enclosing Spheres," Flipcode Code of the Day Forum, June 29, 2001.

http://www.flipcode.com/cgi-bin/msg.cgi?showThread=COTD-SmallestEnclosingSpheres&forum=cotd &id=-1 and ftp://ftp.flipcode.com/cotd/MiniBall.zip

[Cardoso98] Cardoso, Jean-François."Blind Signal Separation: Statistical Principles,"Proceedings of IEEE, vol. 86, no. 10, pp. 2009～2025, 1998.

http://citeseer.ist.psu.edu/cardoso98blind.html

[Carr02] Carr, Nathan. Jesse Hall. John Hart. "The Ray Engine," Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware, pp. 37～46, 2002.

http://basalt.cs.uiuc.edu/～jch/papers/rt2/RayEngine-gh02.pdf

[Cazals95] Cazals, Frédéric. George Drettakis. Claude Puech."Filtering, Clustering, and Hierarchy Construction: A New Solution for Ray Tracing Very Complex Environments,"Computer Graphics Forum (Eurographics 1995 Proceedings), vol. 14, no. 3, pp. 371～382, 1995.

http://w3imagis.imag.fr/Publications/1995/CDP95/index.fr.html

[Chien03] Chien,Yu-Ren. Jing Sin Liu."Improvements to Ellipsoidal Fit Based Collision Detection," Technical Report TR-IIS-03-001, Institute of Information Science, Academia Sinica,Taiwan, 2003.

http://www.iis.sinica.edu.tw/LIB/TechReport/tr2003/tr03001.pdf

[Chilimbi99a] Chilimbi, Trishul. Mark Hill. James Larus. "Cache-conscious Structure Layout," Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI 99), pp. 1～12, May 1999.

http://research.microsoft.com/～trishulc/papers/layout_distr.pdf

[Chilimbi99b] Chilimbi, Trishul. Bob Davidson. James Larus. "Cache-conscious Structure De-finition," Proceedings of the ACM SIGPLAN 1999 Confer-ence on Programming Language Design and Implementation (PLDI 99), pp. 13～24, May 1999.

http://research.microsoft.com/～trishulc/papers/definition_distr.pdf

[Chin92] Chin, Norman."Partitioning a 3-D Convex Polygon with an Arbitrary Plane,"in David Kirk (ed.), Graphics Gems III,Academic Press, pp. 219～222, 1992.

http://www.graphicsgems.org/

[Choi02] Choi, Yi-King. Wenping Wang. Myung-Soo Kim. "Exact Collision Detection of Two Moving Ellipsoids Under Rational Motions,"Technical Report TR-2002-15, Department of Computer Science and Information Systems, Hong Kong University, 2002.

http://www.csis.hku.hk/research/techreps/document/TR-2002-15.pdf

[Chung96] Chung, Kelvin. Wenping Wang."Quick Collision Detection of Polytopes in Virtual Environments," Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 96), Mark Green (ed.), pp. 125～131, 1996.

http://www.geocities.com/kelchung220/download/exactcol.tar.gz

[Chung98] Chung, Adrian. "Re: OBB Trees," comp.graphics.algorithms usenet newsgroup article, Message-ID <6le9nu$ef9$1@magpie.doc.ic.ac.uk>, June 7,1998.

http://groups.google.com/groups?selm=6le9nu$ef9$1@magpie.doc.ic.ac.uk

[Chvatal83] Chvatal, Vasek. Linear Programming. W. H. Freeman & Co, 1983.

[Cleary88] Cleary, John. Geoff Wyvill. "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision," The Visual Computer, vol.4, no. 2, pp. 65～83, July 1988.See also Technical report 1987-282-30,Department of Computer Science, University of Calgary.

http://pharos.cpsc.ucalgary.ca/Dienst/Repository/2.0/Body/ncstrl.ucalgary_cs/1987-282-30/pdf

[Cohen95] Cohen, Jonathan. Ming Lin. Dinesh Manocha. Madhav Ponamgi. "I-COLLIDE: An Interactive and Exact Collision Detection System for Largescale Environments,"Proceedings of the 1995 Symposium on Interactive 3D Graphics, pp. 189～196, 1995.

http://www.cs.jhu.edu/～cohen/Publications/icollide.pdf

[Coutinho01] Coutinho, Murilo. Dynamic Simulations of Multibody Systems, Springer-Verlag, 2001.

[Cormen90] Cormen, Thomas. Charles Leiserson. Ronald Rivest. Introduction to Algorithms, MIT Press, 1990.

[Cromwell94] Cromwell, Robert. "Efficient Eigenvalues for Visualization," in Paul Heckbert (ed.), Graphics Gems IV, Academic Press, pp. 193～198, 1994.

http://www.graphicsgems.org/

[Crosnier99] Crosnier, André. Jarek Rossignac. "Tribox Bounds for Three-dimensional Objects,"

Computers & Graphics, vol. 23, no. 3, pp. 429～437, 1999.

http://www.gvu.gatech.edu/～jarek/papers/Tribox.pdf

[Cychosz92] Cychosz, Joseph."Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations,"in David Kirk (ed.), Graphics Gems III, Academic Press, pp. 284～287, 1992.

[Cychosz94] Cychosz, Joseph. Warren Waggenspack Jr. "Intersecting a Ray with a Cylinder," in Paul Heckbert (ed.),Graphics Gems IV, Academic Press, pp. 356～365, 1994.

http://www.graphicsgems.org/

[Cyrus78] Cyrus, Mike. Jay Beck. "Generalized Two- and Three-dimensional Clipping,"Computers and Graphics, vol. 3, no. 1, pp. 23～28, 1978.

[Deutsch01] Deutsch, Frank. Best Approximation in Inner Product Spaces. Springer-Verlag, 2001.

[Devillers02] Devillers, Olivier. Philippe Guigue. "Faster Triangle-Triangle Intersection Tests," Technical Report 4488, INRIA, June 2002.

http://www.inria.fr/rrrt/rr-4488.html

[Dinerstein02] Dinerstein, Jonathan. Parris Egbert. "Practical Collision Detection in Rendering Hardware for Two Complex 3D Polygon Objects,"Proceedings of the 20th Eurographics UK Conference, pp. 129～135, 2002.

[Dobkin90] Dobkin, David. David Kirkpatrick."Determining the Separation of Preprocessed Polyhedra: A Unified Approach,"Proceedings of the 17th International Colloquium on Automata, Languages and Programming, vol. 443 of Lecture Notes in Computer Science, Springer-Verlag, pp. 400～413, 1990.

http://citeseer.ist.psu.edu/58807.html

[Duff92] Duff,Tom."Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry," Computer Graphics (SIGGRAPH 1992 Proceedings), vol. 26, no. 2, pp. 131～138, July 1992.

http://doi.acm.org/10.1145/133994.134027

[Eberly00] Eberly, David. "Intersection of Cylinders,"Technical Report, Magic Software, November 21, 2000.

http://www.magic-software.com/Documentation/IntersectionOfCylinders.pdf

[Eberly01] Eberly, David. 3D Game Engine Design. Morgan Kaufmann Publishers, 2001.

[Eberly02] Eberly, David. "Intersection of a Sphere and a Cone," Technical Report, Magic Software, March 8, 2002.

http://www.magic-software.com/Documentation/IntersectionSphereCone.pdf

[Eberly03] Eberly, David. "Polyhedral Mass Properties (Revisited)," Technical Report, Magic Software, January 25, 2003.

http://www.magic-software.com/Documentation/PolyhedralMassProperties.pdf

[Eberly04] Eberly, David. Game Physics. Morgan Kaufmann Publishers, 2004.

[Edelsbrunner87] Edelsbrunner, Herbert. Algorithms in Combinatorial Geometry. Springer-Verlag, 1987.

[Ehmann00] Ehmann, Stephen. Ming Lin. "SWIFT: Accelerated Proximity Queries Between Convex

Polyhedra by Multi-Level Voronoi Marching," Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

http://www.cs.unc.edu/～geom/SWIFT/EhmannLinTR00.pdf

[Erickson97] Erickson, Jeff. "Plücker Coordinates," Ray Tracing News, vol. 10, no. 3, December 1997.

http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art11

[Ericson03] Ericson, Christer. "Memory Optimization,"Presentation at Game Developer's Conference, San José, CA, 2003.

http://www.gdconf.com/archives/2003/Ericson_Christer.ppt

[Erikson95] Erikson, Carl. "Error Correction of a Large Architectural Model:The Henderson County Courthouse,"Technical Report TR95-013, Department of Computer Science, University of North Carolina at Chapel Hill, 1995.

http://www.cs.unc.edu/～walk/papers/eriksonc/courthouse.ps.gz

[Everitt02] Everitt, Cass. Mark Kilgard."Practical and Robust Stenciled Shadow Volumes for Hardware-accelerated Rendering,"White paper, NVIDIA Corporation, March 2002.

http://arxiv.org/ftp/cs/papers/0301/0301002.pdf

[Fischer03] Fischer, Kaspar. Bernd Gärtner. "The Smallest Enclosing Ball of Balls: Combinatorial Structure and Algorithms,"Proceedings of 19th Annual ACM Symposium on Computational Geometry (SCG), pp. 292～301, 2003.

http://www.inf.ethz.ch/personal/gaertner/texts/own_work/ballsofballs.pdf

[Floater04] Floater, Michael. Kai Hormann. Géza Kós. "A General Construction of Barycentric Coordinates over Convex Polygons,"to appear in Advances in Computational Mathematics, 2004.

http://vcg.isti.cnr.it/～hormann/papers/coordinates.pdf

[Fortune93] Fortune, Steven. Christopher Van Wyk. "Efficient Exact Arithmetic for Computational Geometry," Proceedings of the 9th ACM Symposium on Computational Geometry, pp. 163～172, May 1993.

http://cm.bell-labs.com/cm/cs/who/sjf/eeacg.ps.gz

[Freeman75] Freeman, Herbert. Ruth Shapira. "Determining the Minimum-area Encasing Rectangle for an Arbitrarily Closed Curve,"Communications of the ACM, vol. 18, no. 7, pp. 409～413, 1975.

[Friedman77] Friedman, Jerome. Jon Bentley. Raphael Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time," ACM Transactions on Mathematical Software, vol. 3, no. 3, pp. 209～226, September 1977.

ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/75/482/CS-TR-75-482.pdf

[Frigo99] Frigo, Matteo. Charles Leiserson. Harald Prokop. Sridhar Ramachandran. "Cache-oblivious Algorithms," Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 99), pp. 285～297, October 1999.

ftp://theory.lcs.mit.edu/pub/cilk/FrigoLePr99.ps.gz

[Fuchs80] Fuchs, Henry. Zvi Kedem. Bruce Naylor. "On Visible Surface Generation by a Priori Tree Structures," *Computer Graphics (SIGGRAPH 1980 Proceedings), vol. 14, no. 3, pp. 124～133, July 1980.*

http://portal.acm.org/citation.cfm?id=807481

[Fuchs83] Fuchs, Henry. Gregory Abram. Eric Grant."Near Real-time Shaded Display of Rigid Objects," Computer Graphics (SIGGRAPH 1983 Proceedings), vol. 17, no. 3, pp. 65～72, 1983.

http://portal.acm.org/citation.cfm?id=801134

[Fünfzig03] Fünfzig, Christoph. Dieter Fellner. "Easy Realignment of k-DOP Bounding Volumes," Proceedings of Graphics Interface 2003, pp. 257～264, 2003.

http://www.graphicsinterface.org/cgi-bin/DownloadPaper?name=2003/117/paper117.pdf

[Garcia99] Garcia, Miguel Angel. Angel Domingo Sappa. Luis Basañez. "Efficient Generation of Object Hierarchies from 3D Scenes," IEEE International Conference on Robotics and Automation, Detroit, USA, pp. 1359～1364, May 1999.

http://www.etse.urv.es/recerca/rivi/pub/3d/icra99.pdf

[Gärtner99] Gärtner, Bernd. "Fast and Robust Smallest Enclosing Balls," Proceedings 7th Annual European Symposium on Algorithms (ESA99), Lecture Notes in Computer Science vol. 1643, Springer-Verlag, pp. 325～338, 1999.

http://www.inf.ethz.ch/personal/gaertner/texts/own_work/esa99_final.pdf

[Gärtner00] Gärtner, Bernd. Sven Schönherr."An Efficient, Exact, and Generic Quadratic Programming Solver for Geometric Optimization,"Proceedings 16th Annual ACM Symposium on Computational Geometry (SCG), pp. 110～118, 2000.

http://www2.inf.ethz.ch/～gaertner/cv/publications.html

[Gilbert88] Gilbert, Elmer. Daniel Johnson. S. Sathiya Keerthi. "A Fast Procedure for Computing the Distance Between Complex Objects in Three-dimensional Space," IEEE Journal of Robotics and Automation, vol. 4, no. 2, pp. 193～203, 1988.

[Gilbert90] Gilbert, Elmer. Chek-Peng Foo."Computing the Distance Between General Convex Objects in Three-dimensional Space," IEEE Transactions on Robotics and Automation, vol. 6, no. 1, pp. 53～61, 1990.

[Glassner89] Glassner, Andrew (ed.). An Introduction to Ray Tracing. Academic Press, 1989.

[Glassner90] Glassner, Andrew (ed.). Graphics Gems. Academic Press, 1990.

http://www.graphicsgems.org/

[Goldberg91] Goldberg, David."What Every Computer Scientist Should Know About Floating-point Arithmetic,"ACM Computing Surveys, March 1991.

[Goldman90] Goldman, Ronald. "Intersection of Three Planes," in Andrew Glassner (ed.), Graphics Gems, Academic Press, p. 305, 1990.

[Goldsmith87] Goldsmith, Jeffrey. John Salmon."Automatic Creation of Object Hierarchies for Ray Tracing,"IEEE Computer Graphics and Applications, vol. 7, no. 5, pp. 14～20, May 1987.

[Golub96] Golub, Gene. Charles van Loan. Matrix Computations (3d ed.). Johns Hopkins University Press, 1996.

[Gomez99] Gomez, Miguel."Simple Intersection Tests for Games,"Gamasutra, October 1999.

http://www. gamasutra.com/features/19991018/Gomez_1.htm

[Gomez01] Gomez, Miguel. "Compressed Axis-aligned Bounding Box Trees,"in Mark DeLoura (ed.), Game Programming Gems 2. Charles River Media, pp. 388～393, 2001.

[Gottschalk96] Gottschalk, Stefan. Ming Lin. Dinesh Manocha. "OBB-Tree:A Hierarchical Structure for Rapid Interference Detection,"Computer Graphics(SIGGRAPH 1996 Proceedings), pp. 171～180, 1996.

http://www.cs.unc.edu/～walk/papers/gottscha/sig96.pdf

[Gottschalk00] Gottschalk, Stefan."Collision Queries Using Oriented Bounding Boxes,"Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill, 2000.

http://www.cs.unc.edu/～geom/theses/gottschalk/main.pdf

[Govindaraju03] Govindaraju, Naga. Stephane Redon. Ming Lin. Dinesh Manocha. "CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware," Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, San Diego, CA, pp. 25～32, 2003.

http://gamma.cs.unc.edu/CULLIDE/

[Green95] Green, Daniel. Don Hatch. "Fast Polygon-Cube Intersection Testing,"in Alan Paeth (ed.), Graphics Gems V, Academic Press, pp. 375～379, 1995.

http://www.graphicsgems.org/

[Greenspan96] Greenspan, Michael. Nestor Burtnyk. "Obstacle Count Independent Real-Time Collision Avoidance."Proceedings of the 1996 IEEE International Conference on Robotics and Automation (ICRA 96), pp. 1073～1080, April 1996.

http://www.ece.queensu.ca/hpages/faculty/greenspan/MyWebPage/papers/grebur96.pdf

[Gu01] Gu, Hong. Thomas Chase. Douglas Cheney. Thomas Bailey. Douglas Johnson."Identifying, Correcting, and Avoiding Errors in Computer-aided Design Models Which Affect Interoperability," Journal of Computing and Information Science in Engineering, vol. 1, no. 2, pp. 156～166, June 2001.

http://dx.doi.org/10.1115/1.1384887

[Guha03] Guha, Sudipto. Shankar Krishnan. Kamesh Munagala. Suresh Venkatasubramanian. "The Power of a Two-sided Depth Test and Its Application to CSG Rendering and Depth Extraction," Proceedings of ACM Siggraph Symposium on Interactive 3D Graphics (to appear), 2003.

http://www.research.att.com/～krishnas/MY_PAPERS/i3d03_csg.pdf

[Guibas85] Guibas, Leonidas. Jorge Stolfi."Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,"ACM Trans actions on Graphics, vol. 4, no. 2, pp. 74～123, April 1985.

http://doi.acm.org/10.1145/282918.282923

[Guibas99] Guibas, Leonidas. David Hsu. Li Zhang. "H-Walk: Hierarchical Distance Computation for Moving Convex Bodies," Proceedings of the Fifteenth Annual Symposium on Computational Geometry, pp. 265～273, 1999.

http://citeseer.ist.psu.edu/guibas99hwalk.html

[Guibas03] Guibas, Leonidas. An Nguyen. Li Zhang."Zonotopes as Bounding Volumes." Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 803～812, 2003.

http://www.hpl.hp.com/personal/Li_Zhang/papers/zonotopes.ps

[Haines88] Haines, Eric. "Automatic Creation of Object Hierarchies for Ray Tracing,"Ray Tracing News,

vol. 1, no. 6, April 1988.

http://www.acm.org/tog/resources/RTNews/html/rtnews3a.html#art6

[Haines91a] Haines, Eric. "Efficiency Improvements for Hierarchy Traversal in RayTracing."in James Arvo (ed.), Graphics Gems II,Academic Press, pp. 267～272, 1991.

[Haines91b] Haines, Eric."Fast Ray-convex Polyhedron Intersection,"in James Arvo (ed.), Graphics Gems II, Academic Press, pp. 247～250, 1991.

http://www.graphicsgems.org/

[Haines94] Haines, Eric. "Point in Polygon Strategies,"in Paul Heckbert (ed.), Graphics Gems IV, Academic Press, pp. 24～46, 1994.

http://www.erichaines.com/ptinpoly/

[Haines99] Haines, Eric. "Quicker Grid Generation via Memory Allocation," Ray Tracing News, vol. 12, no. 1, June 1999.

http://www.acm.org/pubs/tog/resources/RTNews/html/rtnv12n1.html#art6

[Hansen92] Hansen, Eldon. Global Optimization Using Interval Analysis. Marcel Dekker, 1992.

[Hashemi97] Hashemi, Amir. David Kaeli. Brad Calder. "Efficient Procedure Mapping Using Cache Line Coloring," Proceedings of the SIGPLAN Confer-ence on Programming Language Design and Implementation, pp. 171～182, June 1997.

http://www.cs.ucsd.edu/users/calder/papers/PLDI97-Color.pdf

[Hausner98] Hausner, Melvin. A Vector Space Approach to Geometry. Dover Publications, 1998.

[Havran99] Havran,Vlastimil. Filip Sixta. "Comparison of Hierarchical Grids," Ray Tracing News, vol. 12, no. 1, June 1999.

http://www.acm.org/tog/resources/RTNews/html/rtnv12n1.html#art3

[He99] He, Taosong."Fast Collision Detection Using QuOSPO Trees,"Proceed-ings of the 1999 Symposium on Interactive 3D Graphics, pp. 55～62, 1999.

http://citeseer.ist.psu.edu/262510.html

[Heckbert94] Heckbert, Paul (ed.).Graphics Gems IV. Academic Press, 1994.

http://www.graphicsgems.org/

[Held97] Held, Martin. "ERIT: A Collection of Efficient and Reliable Intersection Tests,"Journal of Graphics Tools, vol. 2, no. 4, pp. 25～44, 1997.

http://www.cosy.sbg.ac.at/～held/papers/jgt98.ps.gz

[Held01] Held, Martin. "FIST: Fast Industrial-strength Triangulation of Polygons,"Algorithmica, vol. 30, no. 4, pp. 563～596, August 2001.

http://citeseer.ist.psu.edu/held00fist.html

[Hennessy02] Hennessy, John. David Patterson. David Goldberg. Computer Architecture: A Quantitative Approach (3d ed.). Morgan Kaufmann Publishers,2002.

[Hertel83] Hertel, Stefan. Kurt Mehlhorn. "Fast Triangulation of Simple Polygons," Proceedings of the 1983 International FCT Conference on Fundamentals of Computation Theory, pp. 207～218, 1983.

[Hickey01] Hickey,Timothy. Qun Ju. Maarten van Emden."Interval Arithmetic:From Principles to

Implementation,"Journal of the ACM, vol. 48, no. 5, pp. 1038～1068, September 2001.

http://csr.uvic.ca/～vanemden/Publications/hickeyJuvE.ps

[Hill89] Hill, Mark. Alan Smith."Evaluating Associativity in CPU Caches,"IEEE Transactions on Computers, vol. 38, no. 12, pp. 1612～1630, December 1989.

[Hinker93] Hinker, Paul. Charles Hansen."Geometric Optimization,"Proceedings of the 4th Conference on Visualization 1993, pp. 189～195, October 1993.

http://www.ccs.lanl.gov/ccs1/projects/Viz/pdfs/93-vis.pdf

[Huang97] Huang, Chong-Wei. Tian-Yuan Shih."On the Complexity of Point-in-Polygon Alg-orithms," Computers & Geosciences, vol. 23, no. 1, pp. 109～118,1997.

[Hubbard95] Hubbard Philip. "Collision Detection for Interactive Graphics Applications,"IEEE Transactions on Visualization and Computer Graphics, vol. 1, no. 3, pp. 218～230, September 1995. See also: Technical Report CS-95-08, Department of Computer Science, Brown University.

http://www.cs.brown.edu/publications/techreports/reports/CS-95-08.html

[idSoftware01] id Software. Quake II source code.

ftp://ftp.idsoftware.com/idstuff/source/q2source-3.21.zip

[Imbert93] Imbert, Jean-Louis. "Fourier's Elimination: Which to Choose?" in Principles and Practice of Constraint Programming (PPCP), pp. 117～129, Newport, RI, 1993.

[Intel99] Intel."Applying Streaming SIMD Extensions to Collision Detection,"Intel Application Note AP-812, January 1999.

http://www.intel.com/cd/ids/developer/asmo-na/eng/19049.htm

[James99] James, Adam. "Binary Space Partitioning for Accelerated Hidden Surface Removal and Rendering of Static Environments,"Ph.D. thesis, University of East Anglia, August 1999.

http://www.sys.uea.ac.uk/～aj/PHD/Postscripts/bsp.pdf

[James00] James, Adam. Andy Day. "Conflict Neutralization on Binary Space Partitioning." Computer Graphics Forum, vol. 19, no. 2, pp. 153～163, June 2000.

http://citeseer.ist.psu.edu/james98conflict.html

[Jenkins97] Jenkins, Robert."Algorithm Alley,"Doctor Dobbs Journal, September 1997.

http://burtleburtle.net/bob/

[Jevans89] Jevans, David. Brian Wyvill. "Adaptive Voxel Subdivision for Ray Tracing."Proceedings of Graphics Interface 1989, pp. 164～172, 1989.

[Jolliffe02] Jolliffe, Ian. Principal Component Analysis (2d ed.). Springer Verlag, 2002.

[Joshi03] Joshi, Bhautik. Sebastien Ourselin. "BSP-assisted Constrained Tetrahedralization," Proceedings of the 12th International Meshing Roundtable, Sandia National Laboratories, Albuquerque, NM, pp. 251～260, September 2003.

http://www.imr.sandia.gov/papers/imr12/joshi03.pdf

[Joy86] Joy, Kenneth."A Dissection of Fujimoto's Algorithm,"Technical Report CSE-86-6, Department of Computer Science, University of California, Davis, September 1986.

[Kalvin96] Kalvin, Alan. Russell Taylor."Superfaces: Polygonal Mesh Simplification with Bounded

Error,"IEEE Computer Graphics and Applications, vol. 16, no. 3, pp. 64～77, May 1996.

http://csdl.computer.org/comp/mags/cg/1996/03/g3064abs.htm

[Karasick89] Karasick, Michael. "On the Representation and Manipulation of Rigid Solids,"Ph.D. thesis, Department of Computer Science, Cornell University, NY, March 1989. See also Technical Report TR 89-976.

http://historical.ncstrl.org/tr/ps/cornellcs/TR89-976.ps

[Kay86] Kay, Timothy. James Kajiya. "Ray Tracing Complex Scenes,"Computer Graphics (SIGGRAPH 1986 Proceedings), vol. 20, no. 4, pp. 269～278, 1986.

http://doi.acm.org/10.1145/15922.15916

[Keil02] Keil, Mark. Jack Snoeyink. "On the Time Bound for Convex Decomposition of Simple Polygons," International Journal of Computational Geometry and Applications, vol. 12, no. 3, pp. 181～192, June 2002.

http://www.cs.unc.edu/～snoeyink/papers/convdecomp.ps.gz

[Kirk92] Kirk, David (ed.). Graphics Gems III. Academic Press, 1992.

http://www.graphicsgems.org/

[Klimaszewski97] Klimaszewski, Krzysztof. Thomas Sederberg. "Faster Ray Tracing Using Adaptive Grids," IEEE Computer Graphics and Applications, vol. 17, no. 1, pp. 42～51, January 1997.

[Klosowski98] Klosowski, James. Martin Held. Joseph Mitchell. Henry Sowizral. Karel Zikan. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs,"IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, pp. 21～36, 1998.

ftp://ftp.ams.sunysb.edu/pub/geometry/cd-tvcg-97.ps.gz

[Knittel00] Knittel, Günter."The UltraVis System,"HP Labs Tech Report HPL-2000-100, 2000.

http://www.hpl.hp.com/techreports/2000/HPL-2000-100.pdf

[Knott03a] Knott, David. Dinesh Pai. "CInDeR: Collision and Interference Detection in Real Time Using Graphics Hardware," Proceedings of Graphics Interface 2003.

http://www.graphicsinterface.org/cgi-bin/DownloadPaper?name=2003/139/paper139.pdf

[Knott03b] Knott, David. "CInDeR: Collision and Interference Detection in RealTime Using Graphics Hardware,"Master's thesis, Department of Computer Science, University of British Columbia, 2003.

http://www.cs.ubc.ca/grads/resources/thesis/Nov03/David_Knott.pdf

[Knuth97] Knuth, Donald. The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3d ed.). Addison-Wesley, 1997.

[Knuth98] Knuth, Donald. The Art of Computer Programming, Volume 3: Sorting and Searching (2d ed.). Addison-Wesley, 1998.

[Konečný97] Konečný, Petr. Karel Zikan. "Lower Bound of Distance in 3D,"Proceedings of WSCG 1997, vol. 3, pp. 640～649, 1997. See also Technical Report FIMU-RS-97-01, Faculty of Informatics, Masaryk University, Brno, Czech Republic.

http://ftp.fi.muni.cz/pub/reports/1997/pdf/FIMU-RS-97-01.pdf

[Konečný98] Konečný, Petr."BoundingVolumes in Computer Graphics,"Master's thesis, Faculty of

Informatics, Masaryk University, Brno, Czech Republic, 1998.

http://decibel.fi.muni.cz/HCILAB/thesis.ps.gz

[Krishnan98] Krishnan, Shankar. Amol Pattekar. Ming Lin. Dinesh Manocha. "Spherical Shell: A Higher-order BoundingVolume for Fast Proximity Queries,"Robotics: The Algorithmic Perspective: WAFR 1998, pp. 177～190, 1998.

http://www.research.att.com/～krishnas/MY_PAPERS/wafr98.pdf

[Lahanas00] Lahanas, Michael. Thorsten Kemmerer. Natasa Milickovic. Kostas Karouzakis. Dimos Baltas. Nikolaos Zamboglou. "Optimized Bounding Boxes for Three-dimensional Treatment Planning in Brachytherapy,"Medical Physics, vol. 27, no. 10, pp. 2333～2342, 2000.

http://www.mlahanas.de/Papers/preprint/bbox.pdf

[Laidlaw86] Laidlaw, David. William Trumbore. John Hughes. "Constructive Solid Geometry for Polyhedral Objects." Computer Graphics (SIGGRAPH 1986 Proceedings), vol. 20, no. 4, pp. 161～170, 1986.

http://doi.acm.org/10.1145/15922.15904

[Larcombe95] Larcombe,Mikki. "Re: Arbitrary Bounding Box Intersection," comp.graphics.algorithms usenet newsgroup article. Message-ID<1995Oct11.130545.22985@dcs.warwick.ac.uk>, October 11, 1995.

[Larsen99] Larsen, Eric. Stefan Gottschalk. Ming Lin. Dinesh Manocha. "Fast Proximity Queries with Swept Sphere Volumes," Technical Report TR99-018, Department of Computer Science, University of North Carolina, Chapel Hill, 1999.

ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/ssv.pdf

[Larsen00] Larsen, Eric. Stefan Gottschalk. Ming Lin. Dinesh Manocha. "Fast Distance Queries Using Rectangular Swept Sphere Volumes," Proceedings of IEEE International Conference on Robotics and Automation, 2000.

ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/icrassv00.pdf

[Lengyel02] Lengyel, Eric."The Mechanics of Robust Stencil Shadows,"Gamasutra, October 2002.

http://www.gamasutra.com/features/20021011/lengyel_01.htm

[Levine00] Levine, Ron. "Collisions of Moving Objects," gdalgorithms-list mailing list article, November 14, 2000.

[Li98] Li, Tsai-Yen. Jin-Shin Chen. "Incremental 3D Collision Detection with Hierarchical Data Structures,"Proceedings of ACM Symposium on Virtual Reality Software and Technology (VRST 98), pp. 139～144, November 1998.

http://www3.nccu.edu.tw/～li/Publication/

[Lin91] Lin, Ming. John Canny. "A Fast Algorithm for Incremental Distance Calculation," Proceedings of the 1991 IEEE International Conference on Robotics and Automation, pp. 1008～1014, April 1991.

[Luebke95] Luebke, David. Chris Georges. "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets," Proceedings of the 1995 Symposium on Interactive 3D Graphics, pp.105～106, 1995.

http://www.cs.virginia.edu/～luebke/publications/portals.html

[Luebke02] Luebke, David. Martin Reddy. Jonathan Cohen. Amitabh Varshney. Benjamin Watson. Robert Huebner. Level of Detail for 3D Graphics. Morgan Kaufmann Publishers, 2002.

[Lumelsky85] Lumelsky,Vladimir."On Fast Computation of Distance Between Line Segments," Information Processing Letters, vol. 21, no. 2, pp. 51～61, 1985.

[MacDonald90] MacDonald, David. Kellogg Booth."Heuristics for Ray Tracing Using Space Subdivision,"The Visual Computer, vol. 6, no. 3, pp. 153～166, 1990.

[Mäntylä88] Mäntylä, Martti. An Introduction to Solid Modeling. Computer Science Press, 1988.

[McConnell93] McConnell, Steve. Code Complete. Microsoft Press, 1993.

[McGuire00] McGuire, Max."Quake 2 BSP File Format,"June 7, 2000.

http://www.flipcode.com/tutorials/tut_q2levels.shtml

[Meisters75] Meisters, Gary. "Polygons Have Ears," American Mathematical Monthly, vol. 82, pp. 648～651, June/July 1975.

[Melax00] Melax, Stan. "Dynamic Plane Shifting BSP Traversal,"Proceedings of Graphics Interface 2000, pp. 213～220, May 2000.

http://www.graphicsinterface.org/proceedings/2000/140/

[Meyer02] Meyer, Mark. Haeyoung Lee. Alan Barr. Mathieu Desbrun. "Generalized Barycentric Coordinates on Irregular Polygons," Journal of Graphics Tools, vol. 7, no. 1, pp. 13～22, 2002.

http://www-grail.usc.edu/pubs/MHBD02.pdf

[Michelucci96] Michelucci, Dominique."Arithmetic Issues in Geometric Computations," Proce-edings of the 2nd Real Numbers and Computers Conference, Marseille, France, pp. 43～69. April 1996.

http://www.emse.fr/～micheluc/nro96.ps.gz

[Miettinen02a] Miettinen, Ville. "RE: A Better Method of Removing Duplicate Vertices…," gdalgorithms-list mailing list article, July 3, 2002.

http://sourceforge.net/mailarchive/message.php?msg_id=1777716

[Miettinen02b] Miettinen, Ville. "RE: Generating a (Near) Optimal OBB from a Set of Points," gdalgorithms-list mailing list article, December 5, 2002.

http://sourceforge.net/mailarchive/message.php?msg_id=2623767

[Mirtich96a] Mirtich, Brian. "Fast and Accurate Computation of Polyhedral Mass Properties," Journal of Graphics Tools, vol. 1, no. 2, pp. 31～50, 1996.

http://citeseer.ist.psu.edu/mirtich96fast.html

[Mirtich96b] Mirtich, Brian. "Impulse-based Dynamic Simulation of Rigid Body Systems," Ph.D. thesis, Department of Computer Science, University of California, Berkeley, 1996.

[Mirtich97] Mirtich, Brian. "Efficient Algorithms for Two-phase Collision Detection," in Practical Motion Planning in Robotics: Current Approaches and Future Directions, John Wiley & Sons, pp. 203～223, 1998. See also Technical Report TR97-23, Mitsubishi Electric Research Laboratories (MERL), 1997

http://www.merl.com/reports/TR97-23/

[Mirtich98] Mirtich, Brian. "V-Clip: Fast and Robust Polyhedral Collision Detection,"ACM Transactions on Graphics, vol. 17, no. 3, pp. 177～208, July 1998. See also Technical Report TR97-05, Mitsubishi Electric Research Laboratories (MERL), 1997.

http://www.merl.com/papers/docs/TR97-05.pdf

[Mitchell90] Mitchell, Don."Robust Ray Intersection with Interval Arithmetic," Proceedings on Graphics Interface 1990, Halifax, Nova Scotia, pp. 68～74, 1990.

[Mitchell00] Mitchell, Mark. "Type-based Alias Analysis," Dr. Dobb's Journal, October 2000.

[Möller97a] Möller, Tomas. Ben Trumbore. "Fast, Minimum-storage Ray/Triangle Intersection," Journal of Graphics Tools, vol. 2, no. 1, pp. 21～28, 1997.

http://www.graphics.cornell.edu/pubs/1997/MT97.html

[Möller97b] Möller, Tomas. "A Fast Triangle-Triangle Intersection Test,"Journal of Graphics Tools, vol. 2, no. 2, pp. 25～30, 1997.

http://www.ce.chalmers.se/staff/tomasm/pubs/tritri.pdf

[Moore66] Moore, Ramon. Interval Analysis. Prentice-Hall, 1966.

[Moore88] Moore, Matthew. Jane Wilhelms. "Collision Detection and Response for Computer Animation," Computer Graphics (SIGGRAPH 1988 Proceedings), vol. 22, no. 4, pp. 289～298, August 1988.

[Mortenson99] Mortenson, Michael. Mathematics for Computer Graphics Applications (2d ed.). Industrial Press, 1999.

[Motzkin53] Motzkin, Theodore. H. Raiffa. G. Thompson. R. Thrall."The Double Description Method," in H. Kuhn and A. Tucker (eds.), Contributions to the Theory of Games, Volume II, Volume 8 of Annals of Math Studies, pp. 51～73, Princeton University Press, 1953.

[Mulmuley94] Mulmuley, Ketan. Computational Geometry: An Introduction Through Randomized Algorithms. Prentice-Hall, 1994.

[Murali97] Murali, T. M. Thomas Funkhouser."Consistent Solid and Boundary Representations from Arbitrary Polygon Data," Proceedings of the 1997 Symposium on Interactive 3D Graphics, pp. 155～162, April 1997.

http://www.cs.princeton.edu/～funk/symp97.ps.gz

[Myszkowski95] Myszkowski, Karol. Oleg Okunev. Tosiyasu Kunii. "Fast Collision Detection Between Computer Solids Using Rasterizing Graphics Hardware," The Visual Computer, vol. 11, no. 9, pp. 497～511, 1995. See also Technical Report 94-1-047, University of Aizu, Japan, 1995.

ftp://ftp.u-aizu.ac.jp/u-aizu/doc/Tech-Report/1994/94-1-047.ps.Z

[Nagle02]Nagle, John. "Re: Collision Points with SOLID," comp.graphics. algorithms usenet newsgroup article, Message-ID <3C4089BC.80907@animats.com>, January 12, 2002.

http://groups.google.com/groups?selm=3C4089BC.80907%40animats.com

[Naylor93] Naylor, Bruce."Constructing Good Partitioning Trees."Proceedings of Graphics Interface 1993, pp. 181～191, May 1993.

http://www.graphicsinterface.org/pre1996/93-Naylor.pdf

[Nettle00] Nettle, Paul. Generic Collision Detection for Games Using Ellipsoids, Unpublished manuscript, 2000.

http://www.fluidstudios.com/publications.html

[Omohundro89] Omohundro, Stephen. "Five Balltree Construction Algorithms," Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA, November 1989.

http://www.icsi.berkeley.edu/ftp/global/pub/techreports/1989/tr-89-063.pdf

[O'Rourke85] O'Rourke, Joseph. "Finding Minimal Enclosing Boxes," International Journal of Computer and Information Sciences, vol. 14, no. 3, pp. 183～199, June 1985.

[O'Rourke98] O'Rourke, Joseph. Computational Geometry in C (2d ed.). Cambridge University Press, 1998.

[Ostgard03] Ostgard, Nathan. "Quake 3 BSP Collision Detection," March 28, 2003.

http://www.nathanostgard.com/tutorials/quake3/collision/

[O'Sullivan99] O'Sullivan, Carol. John Dingliana. "Real-time Collision Detection and Response Using Sphere-trees," Technical Report TCD-CS-1999-10, Computer Science Department, Trinity College, Dublin, February 1999.

http://www.cs.tcd.ie/publications/tech-reports/reports.99/TCD-CS-1999-10.pdf

[Paeth95] Paeth, Alan (ed.). Graphics Gems V. Academic Press, 1995.

http://www.graphicsgems.org/

[Paoluzzi89] Paoluzzi, Alberto. M. Ramella. A. Santarelli. "Boolean Algebra over Linear Polyhedra," Computer Aided Design, vol. 21, no. 8, pp. 474～484, August 1989.

[Preparata85] Preparata, Franco. Michael Shamos. Computational Geometry: An Introduction. Springer-Verlag, NewYork, 1985.

[Priest91] Priest, Douglas. "Algorithms for Arbitrary Precision Floating-point Arithmetic,"Pro-ceedings of the 10th Symposium on Computer Arithmetic, pp. 132～143, Grenoble, France, June 1991.

http://www.cs.cmu.edu/～quake-papers/related/Priest.ps

[Prokop99] Prokop, Harald. "Cache-oblivious Algorithms," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[Puppo97] Puppo, Enrico. Roberto Scopigno. "Simplification, LOD, and Multiresolution: Principles and Applications,"Eurographics 1997 Tutorial Notes PS97 TN4, 1997.

http://www.disi.unige.it/person/PuppoE/PS/EG97-tut.ps.gz

[Purcell02] Purcell, Timothy. Ian Buck. William Mark. Pat Hanrahan. "Ray Tracing on Prog-rammable Graphics Hardware," ACM Transactions on Graphics(Proceedings of ACM SIGGRAPH 2002), vol. 21, no. 3, pp. 703～712, 2002.

http://graphics.stanford.edu/papers/rtongfx/

[Quinlan94] Quinlan, Sean. "Efficient Distance Computation Between Nonconvex Objects," Proceedings of IEEE International Conference on Robotics and Automation, pp. 3324～3329, May 1994.

http://www.cs.bell-labs.com/who/seanq/icra94.pdf

[Redon02] Redon, Stéphane. Abderrahmane Kheddar. Sabine Coquillart."Fast Continuous Collision Detection Between Rigid Bodies," Computer Graphics Forum (Eurographics 2002 Proceedings), vol. 21, no. 3, September 2002.

http://www-rocq.inria.fr/i3d/i3d/publications/redoneg2002.pdf

[Reinhard00] Reinhard, Erik. Brian Smits. Charles Hansen. "Dynamic Acceleration Structures for Interactive Ray Tracing," Proceedings of the Eurographics Workshop on Rendering Techniques 2000, Brno,

Czech Republic, pp. 299～306, June 2000.

http://www.cs.utah.edu/～reinhard/papers/egwr2k.pdf

[Ribelles01] Ribelles, José. Paul Heckbert. Michael Garland. Tom Stahovich. Vinit Srivastava. "Finding and Removing Features from Polyhedra,"Proceedings of 2001 ASME Design Engineering Technical Conference (DETC 01), September 2001.

http://www.uiuc.edu/ph/www/garland/papers/polyfeat.pdf

[Rimon92] Rimon, Elon. Stephen Boyd. "Efficient Distance Computation Using Best Ellipsoid Fit,"Proceedings of the 1992 IEEE International Symposium on Intelligent Control, pp. 360～365, August 1992.

http://citeseer.ist.psu.edu/rimon92efficient.html

[Ritter90] Ritter, Jack. "An Efficient Bounding Sphere," in Andrew Glassner (ed.), Graphics Gems, Academic Press, pp. 301～303, 1990.

http://www.graphicsgems.org/

[Robison96a] Robison, Arch. "C++ Gets Faster for Scientific Computing," Computers in Physics, vol. 10, pp. 458～462, 1996.

[Robison96b] Robison, Arch. "The Abstraction Penalty for Small Objects in C++," The Parallel Object-Oriented Methods and Applications Conference(POOMA '96), 1996.

[Robison99] Robison, Arch. "Restricted Pointers Are Coming," C/C++ Users Journal, July 1999.

http://www.cuj. com/ documents/s=8050/cuj9907robison/

[Rockafellar96] Rockafellar, R. Tyrell. Convex Analysis. Princeton University Press, 1996.

[Rogers98] Rogers, David. Procedural Elements for Computer Graphics (2d ed.). McGraw-Hill, 1998.

[Rossignac92] Rossignac, Jarek. Abe Megahed. Bengt-Olaf Schneider. "Interactive Inspection of Solids: Cross-sections and Interferences," Computer Graphics (SIGGRAPH 1992 Proceedings), July 1992.

[Röttger02] Röttger, Stefan. Alexander Irion. Thomas Ertl. "Shadow Volumes Revisited," in Vaclav Skala (ed.), Proceedings of Winter School of Computer Graphics (WSCG), pp. 373～379, 2002.

http://www.vis.uni-stuttgart.de/～roettger/data/Papers/SHADOWS.PDF

[Rundberg99] Rundberg, Peter."An Optimized Collision Detection Algorithm,"MSc thesis, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, 1999.

http://www.ce.chalmers.se/staff/biff/exjobb/

[Ruspini] Ruspini, Diego. gilbert.c, a C Version of the Original Fortran Implementation of the GJK Algorithm.

ftp://labrea.stanford.edu/cs/robotics/sean/distance/gilbert.c

[Salesin92] Salesin, David. Filippo Tampieri. "Grouping Nearly Coplanar Polygons into Coplanar Sets,"in David Kirk (ed.), Graphics Gems III, Academic Press, pp. 225～230, 1992.

http://www.graphicsgems.org/

[Samet90a] Samet, Hanan. Applications of Spatial Data Structures. Addison-Wesley, 1990.

[Samet90b] Samet, Hanan. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.

[Sauer01] Sauer, Florian. "Dealing with Memory Constraints: Bringing Dr.Jones to the Infernal

Machine,"Presentation at Game Developer's Conference, San José, 2001.

http://www.gdconf.com/archives/2001/sauer.doc

[Sayood00] Sayood, Khalid. Introduction to Data Compression (2d ed.). Morgan Kaufmann Publishers, 2000.

[Schneider02] Schneider, Philip. David Eberly. Geometric Tools for Computer Graphics. Morgan Kaufmann Publishers, 2002.

[Schömer00] Schömer, Elmar. Jürgen Sellen. Marek Teichmann. Chee-Keng Yap. "Smallest Enclosing Cylinders," Algorithmica, vol. 27, no. 2, pp. 170～186, 2000.

http://www.mpi-sb.mpg.de/～schoemer/publications/ALGO00.pdf

[Schorn94] Schorn, Peter. Frederick Fisher. "Testing the Convexity of a Polygon,"in Paul Heckbert (ed.), Graphics Gems IV, Academic Press, pp. 7～15, 1994.

http://www.graphicsgems.org/

[Schroeder01] Schroeder, Tim."Collision Detection Using Ray Casting,"Game Developer, vol. 8, no. 8, pp. 50～56, August 2001. http://www.gdmag.com/src/aug01.zip

[Segal03] Segal, Mark. Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.5), October 2003.

http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf

[Seidel90] Seidel, Raimund. "Linear Programming and Convex Hulls Made Easy," Proceedings 6th Annual ACM Conference on Computational Geometry, pp. 211～215, Berkeley, CA, 1990.

[Shene94] Shene, Ching-Kuang."Computing the Intersection of a Line and a Cylinder," in Paul Heckbert (ed.), Graphics Gems IV, Academic Press, pp. 353～355, 1994.

http://www.graphicsgems.org/

[Shewchuk96a] Shewchuk, Jonathan. "Robust Adaptive Floating-point Geometric Predicates," Proceedings of the Twelfth Annual Symposium on Computational Geometry, pp. 141～150, 1996.

http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-predicates.ps

[Shewchuk96b] Shewchuk, Jonathan. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," First Workshop on Applied Computational Geometry, pp. 124～133, May 1996.

http://www-2.cs.cmu.edu/afs/cs/project/quake/public/papers/triangle.ps

[Shirley00] Shirley, Peter. Realistic Ray Tracing. A. K. Peters, 2000.

[Shoemake98] Shoemake, Ken. "Plücker Coordinate Tutorial," Ray Tracing News, vol. 11, no. 1, July 1998.

http://www.acm.org/tog/resources/RTNews/html/rtnv11n1.html#art3

[Skiena98] Skiena, Steven. The Algorithm Design Manual. TELOS (Springer-Verlag), 1998.

[Snyder93] Snyder, John. Adam Woodbury. Kurt Fleischer. Bena Currin. Alan Barr. "Interval Methods for Multi-point Collisions Between Time-dependent Curved Surfaces," Computer Graphics (SIGGRAPH 1993 Proceedings), pp. 321～334, 1993.

http://doi.acm.org/10.1145/166117.166158

[Stepanov] Stepanov, Alexander. Stepanov Benchmark Code.

http://www.physics.ohio-state.edu/～wilkins/computing/benchmark/stepanov.html

[Stewart02] Stewart, Nigel. Geoff Leach. Sabu John. "Linear-time CSG Rendering of Intersected Convex Objects,"The 10th International Conference in Central Europe on Computer Graphics Visualization and Computer Vision 2002 (WSCG 2002), vol. 2, pp. 437～444.

http://www.nigels.com/research/wscg2002.pdf

[StøltingBrodal01] Stølting Brodal, Gerth. Rolf Fagerberg. Riko Jacob. Cache Oblivious Search Trees via Binary Trees of Small Height, Technical Report BRICS RS-01-36, BRICS, University of Aarhus, Denmark, October 2001.

http://www.brics.dk/～rjacob/Publications/brics-rs-01-36.ps.gz

[Stone98] Stone, James. John Porrill. Independent Component Analysis and Projection Pursuit: A Tutorial Introduction. Unpublished Technical Report, April 1998.

ftp://ftp.shef.ac.uk/pub/misc/personal/pc1jvs/papers/ica_tutorial.ps.gz

[Sunday02] Sunday, Daniel. "Fast Polygon Area and Newell Normal Computation,"Journal of Graphics Tools, vol. 7, no. 2, pp. 9～13, 2002.

[Sutherland74] Sutherland, Ivan. Gary Hodgman. "Reentrant Polygon Clipping,"Communications of the ACM, vol. 17, no. 1, pp. 32～42, 1974.

[Szécsi03] Szécsi, László. "An Effective Implementation of the k-D Tree," in Jeff Lander (ed.), Graphics Programming Methods, Charles River Media, pp. 315～326, 2003.

[Tampieri92] Tampieri, Filippo. "Newell's Method for Computing the Plane Equation of a Polygon,"in David Kirk (ed.), Graphics Gems III, Academic Press, pp. 231～232, 1992.

http://www.graphicsgems.org/

[Teller92] Teller, Seth. "Visibility Computations in Densely Occluded Polyhedral Environments,"Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1992.

http://graphics.lcs.mit. edu/～seth/pubs/pubs.html

[Thibault87] Thibault, William. Bruce Naylor. "Set Operations on Polyhedra Using Binary Space Partitioning Trees," Computer Graphics (SIGGRAPH 1987 Proceedings), vol. 21, no. 4, pp. 153～162, 1987.

http://doi.acm.org/10.1145/37401.37421

[Toussaint83] Toussaint, Godfried. "Solving Geometric Problems with the Rotating Calipers," Proceedings of IEEE MELECON 1983, Athens, Greece, May 1983.

http://www-cgrl.cs.mcgill.ca/～godfried/publications/calipers.ps.gz

[Tzafestas96] Tzafestas, Costas. Philippe Coiffet. "Real-time Collision Detection Using Spherical Octrees: Virtual Reality Application," Proceedings 1996 IEEE International Workshop on Robot and Human Communication (ROMAN 96). Tsukuba, Japan, pp. 500～506, November 1996.

http://www.softlab.ntua.gr/～ktzaf/Publications/Tzafestas_ROMAN_96.zip

[Ulrich00] Ulrich, Thatcher. "Loose Octrees," in Mark DeLoura (ed.), Game Programming Gems. Charles River Media, pp. 444～453, 2000.

[Veldhuizen95] Veldhuizen, Todd. "Expression Templates," C++ Report, vol. 7, no. 5, pp. 26～31, June 1995.

http://osl.iu.edu/～tveldhui/papers/Expression-Templates/exprtmpl.html

[Vlack01] Vlack, Kevin. Susumu Tachi. "Fast and Accurate Spatio-temporal Intersection Detection with the GJK Algorithm,"Proceedings of the International Conference on Artificial Reality and Telexistence (ICAT 01), pp. 79～84, 2001.

http://sklab-www.pi.titech.ac.jp/～hase/ICATPHP/upload/30_camera.pdf

[Voorhies92] Voorhies, Douglas. "Triangle-cube Intersection," in David Kirk (ed.), Graphics Gems III, Academic Press, pp. 236～239, 1992.

http://www.graphicsgems.org/

[Wald01] Wald, Ingo. Philipp Slusallek. Carsten Benthin. Markus Wagner. "Interactive Rendering with Coherent Ray Tracing," Proceedings of the EUROGRAPHICS 2001, pp. 153～164, 2001.

http://graphics.cs.uni-sb.de/Publications/2001/InteractiveRenderingWithCoherentRayTracing.pdf

[Walker99] Walker, Robert. Jack Snoeyink. "Practical Point-in-Polygon Tests Using CSG Representations of Polygons,"Technical Report TR-99-12, Department of Computer Science, University of British Columbia, November 1999.

http://www.cs.ubc.ca/labs/imager/tr/pdf/walker.1999a.pdf

[Walster98] Walster, William. The Extended Real Interval System, 1998.

http://www.mscs.mu.edu/～globsol/Papers/extended_intervals.ps

[Wang01] Wang, Wenping. Jiaye Wang. Myung-Soo Kim. "An Algebraic Condition for the Separation of Two Ellipsoids,"Computer Aided Geometric Design, vol. 18, no. 6, pp. 531～539, July 2001.

http://3map.snu.ac.kr/mskim/ftp/ellipsoid.pdf

[Wang02] Wang, Wenping. Yi-King Choi. Bin Chan. Myung-Soo Kim. Jiaye Wang. Efficient Collision Detection for Moving Ellipsoids Based on Simple Algebraic Test and Separating Planes, Technical Report TR-2002-16, Department of Computer Science and Information Systems, University of Hong Kong, 2002.

http://www.csis.hku.hk/research/techreps/document/TR-2002-16.pdf

[Warren93] Warren, Michael. John Salmon. "A Parallel Hashed Oct-tree N-body Algorithm," Proceedings of Supercomputing 1993, pp. 12～21, 1993.

http://www.thesalmons.org/john/pubs/papers.html

[Weghorst84] Weghorst, Hank. Gary Hooper. Donald Greenberg. "Improved Computational Methods for Ray Tracing,"ACM Transactions on Graphics, vol. 3, no. 1, pp. 52～69, January 1984.

http://doi.acm.org/ 10.1145/357332.357335

[Weiler88] Weiler, Kevin. "The Radial Edge Structure: A Topological Representation for Non-manifold Geometric Modeling,"Geometric Modeling for CAD Applications, North-Holland, pp. 3～36, 1988.

[Weiler94] Weiler, Kevin. "An Incremental Angle Point in Polygon Test," in Paul Heckbert (ed.), Graphics Gems IV, Academic Press, pp. 16～23, 1994.

http://www.graphicsgems.org/

[Welzl91] Welzl, Emo. "Smallest Enclosing Disks (Balls and Ellipsoids)," in Hermann Maurer (ed.), New Results and New Trends in Computer Science, vol.555 of Lecture Notes in Computer Science, Springer-Verlag, pp. 359～370, 1991.

http://citeseer.ist.psu.edu/welzl91smallest.html

[Wu92] Wu, Xiaolin. "A Linear-time Simple Bounding Volume Algorithm," in David Kirk (ed.), Graphics Gems III, Academic Press, pp. 301～306, 1992.

http://www.graphicsgems.org/

[Xavier97] Xavier, Patrick. "Fast Swept-volume Distance for Robust Collision Detection,"Proceedings of the 1997 IEEE International Conference on Robotics and Automation, Albuquerque, New Mexico, April 1997.

[Yong00] Lu, Yong. Rajit Gadh. "Constrained and Aggregated Half Space Volume Decomposition: Generating Cutting Patches with 'Internal' and 'External' Extending,"Proceedings of the Eighth Pacific Conference on Computer Graphics and Applications (PG 00), pp. 262～271, 2000.

http://csdl.computer.org/comp/proceedings/pg/2000/0868/00/08680262abs.htm

[Zachmann95] Zachmann, Gabriel. "The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra," First Workshop on Simulation and Interaction in Virtual Environments (SIVE 95), University of Iowa, July 1995.

http://web.informatik.uni-bonn.de/II/ag-klein/people/zach/papers/

[Zachmann98] Zachmann, Gabriel. "Rapid Collision Detection by Dynamically Aligned DOP-trees," Proceedings of IEEEVirtual Reality Annual International Symposium (VRAIS 98), Atlanta, Georgia, pp. 90～97, March 1998.

http://web.informatik.uni-bonn.de/II/ag-klein/people/zach/papers/

[Zachmann00] Zachmann, Gabriel. "Virtual Reality in Assembly Simulation: Collision Detection, Simulation Algorithms, and Interaction Techniques,"Ph.D. thesis, Department of Computer Science, Darmstadt University of Technology, Germany, May 2000.

http://web.informatik.uni-bonn.de/II/agklein/people/zach/papers/zachmann_diss.pdf

[Zhang95] Zhang, Lixin. Jing Xiao. "Derivation of Contact States from Geometric Models of Objects," Proceedings of the IEEE International Conference Assembly and Task Planning, Pittsburgh, pp. 375～380, August 1995.

http://www.coe.uncc.edu/～xiao/contactd/ct.ps