第一章主要是绪论内容，主要介绍一下本书的大纲

1. 如何架构工业级生产用游戏引擎
2. 现实中的游戏开发团队是怎么样组织和运作的
3. 有哪些主要子系统和设计模式不断出现在几乎所有的游戏引擎中
4. 每个主要子系统的典型需求
5. 有哪些子系统与游戏类型或具体游戏无关，有哪些子系统是为某游戏类型或者具体游戏而设计的
6. 引擎和游戏的边界位于何处

**1-绪论**

**1.1-Game Studio的组成**

**Engineer**：主要负责设计和实现软件。一般根据职能可以分为两种，一种是运行时程序员，负责制作引擎和游戏本身，另一种是工具程序员，负责制作离线工具，供整个团队使用。

**Artist**：艺术家可以细分为以下几类：

1. Concept Artist：一般是美术指导，主要负责通过素描或者绘画，让团队了解游戏的预设最终面貌。一般而言，游戏成品的屏幕截图会十分贴近concept art(概念艺术图)
2. 3D Modeler：负责3D建模，一般可以分为前景建模师(foreground modeler)和背景建模师(background modeler又称关卡设计师)。前者负责制作物体角色载具武器等移动或着可交互的对象，后者则是主要制作静态的背景几何模型，比如地形建筑桥梁
3. Texture Modeler：一般是制作纹理的二维影像，这些纹理可以被用于贴附于3d模型上，使得模型的细节更丰富，更具有真实感
4. Lighting Artist：灯光师主要负责布置静态和动态光源，通过颜色亮度光源方向等属性调整场景的美感和情感
5. Animator：动画师主要为游戏中的角色和物体加入动作。
6. Motion Capture Actor:用来提供原始的动作数据。这些数据经过动画师的调整后置于游戏中
7. Sound Designer：主要是制作和混合游戏中的音效和音乐
8. Voice Actor：为游戏角色配音
9. Composer：为游戏创作音乐

**Game Designer**:游戏设计师负责设计玩家体验的互动部分，这部分一般称为游戏性。不同种类的游戏设计师从事不同细致程度的工作。一些资深的游戏设计师一般是在宏观的层面上设定故事的主线、整体的章节或关卡顺序、玩家的高层次目标。其他的游戏设计师(关卡设计师)则是在虚拟游戏世界的个别关卡或者地域上工作。其他的游戏设计师则会在非常技术性的层面上和游戏性工程师(gameplay engineer)合作

**Producer**：一般而言，制作人是负责管理开发进度的，并同时承担人力资源经理的职责。还有些制作人主要做的是资深游戏设计师的工作。

**Others**：其余的工作人员主要是支持团队，比如行政管理团队以及市场策划团队，还有负责为团队采购配置软硬件的IT支持。

**Publisher**：游戏的市场策划、制造以及分销通常由发行商负责。第一开发商是指游戏工作室直接隶属于游戏主机生产商

**1.2-游戏是什么**

游戏通常而言是一种互动体验，为玩家提供一连串渐进式挑战，玩家最终能够通过学习而精通该游戏。

**电子游戏作为软实时模拟**：

大部分二维或三维的电子游戏，会被计算机科学家称为软实时互动基于代理的计算机模拟(soft real-time interactive agent-based computer simulation)的例子。

数学模型是现实或虚拟世界的模拟。近似化(approximation)和简化(simplification)是游戏开发者最有力的两个工具。若能巧妙地运用它们,就算是一个被大量简化的模型,也能非常接近现实,难辨真假,而能带来的乐趣也比现实更多。

基于代理的模拟是指,模拟中多个独立的实体(称为代理)一起互动。此术语非常符合三维电子游戏的描述,游戏中的载具、人物角色、火球、豆子等都可视为代理。所以多数游戏采用面向对象(object-oriented)编程语言,或较宽松的基于对象(object-based)编程语言。

所有互动电子游戏都是时间性模拟的(temporal simulation)，即游戏世界是动态的(dynamic)——随着游戏事件和故事的展开,游戏世界的状态随着时间改变。游戏也必须回应人类玩家的输入,这些输入是游戏本身不可预知的,因而也说明游戏是互动时间性模拟的(interactive temporal simulation)。最后,多数游戏会描绘游戏的故事,并实时回应玩家的输入,这使游戏成为互动实时模拟的(interactive real-time simulation)。显著的反例是一些回合制游戏,如计算机化的象棋及非实时策略游戏,尽管如此,这些游戏通常也会向用户提供某种形式的实时图形用户界面(graphical user interface, GUI)。

时限(deadline)是所有实时模拟的核心概念。在电子游戏中,明显的例子是需要屏幕每秒最少更新24次,以制造运动的错觉。(大部分游戏会以每秒30帧或60帧的频率渲染画面,因为这是 NTSC 制式显示器刷新率1的倍数。)当然,电子游戏也有其他类型的期限。例如物理模拟可能需要每秒更新120次以保持稳定。一个游戏角色的人工智能系统可能每秒最少要“想一次”才能显得不呆。另外,也可能需要每1/60秒调用一次声音程序库,以确保音频缓冲有足够的声音数据,避免发出一些短暂失灵声音。

“软”实时系统是指一些系统,即使错过期限也不会造成灾难性后果。因此,所有游戏都是软实时系统(soft real-time system)——如果帧数不足,人类玩家在现实中不会因此而死亡。与此相比,硬实时系统(hard real-time system)错过期限可能会导致操作者损伤甚至死亡。直升机的航空电子系统和核能发电厂的控制棒(control rod)2系统便是硬实时系统的例子。

模拟虚拟世界许多时候要用到数学模型。数学模型可分为分析式(analytic)或数值式(nu-merical)。分析式模型可为其自变量(independent variable)设任何值来求值。可是,大部分数学问题并没有闭型解。在电子游戏中,用户输入是不能预知的,因此不应期望可以对整个游戏完全用分析式建模，而可以用类似微分积分的方式进行逼近。

**1.3-游戏引擎是什么**

“游戏引擎”这个术语在20世纪90年代中期形成,与第一人称射击游戏(first person shooter, FPS)如id Software公司开发的非常受欢迎的游戏《毁灭战士》(Doom)有关。《毁灭战士》的软件架构被相当清楚地划分成核心软件组件(如三维图形渲染系统、碰撞检测系统和音频系统等)、美术资产(art asset)、游戏世界、构成玩家游戏体验的游戏规则(rule of play)。这么划分是很有价值的,若另一个开发商取得这类游戏的授权,只要制作新的美术、关卡布局、武器、角色、载具、游戏规则等,对引擎软件做出很少的修改,就可以把游戏打造成新产品。

数据驱动架构(data-driven architecture)或许可以用来分辨一个软件的哪些部分是引擎,哪些部分是游戏。若一个游戏包含硬编码逻辑或游戏规则,或使用特例代码去渲染特定种类的游戏对象,则复用该软件去制作新游戏就会变得困难甚至不可行。因此,这里说的“游戏引擎”是指可扩展的软件,而且不需要大量修改就能成为多款游戏软件的基础。

有些人可能以为游戏引擎能变成一个通用软件(像Apple QuickTime或微软的 WindowsMedia Player)，去运行几乎任何可以想象到的游戏内容。可是,这个设想至今尚未(或许永远不能)实现。大部分游戏引擎是针对特定游戏及特定硬件平台所精心制作及微调的。就算是一些较通用的游戏引擎,其实也只适合制作某类型的游戏,例如第一人称射击或赛车游戏。我们完全可以说,游戏引擎或中间件组件越通用,在特定平台运行特定游戏的性能就越一般。出现这种现象是因为设计高效的软件总是需要取舍的,而这些取舍是基于一些假设的,像一个软件会如何使用及在哪个硬件上运行等。

随着计算机硬件速度的提高及专用显卡的应用,再加上高效的渲染算法及数据结构,不同游戏类型的图形引擎差异已经缩小。例如,现在可以用第一人称射击引擎去做实时策略游戏。但是,通用性和最优性仍然需要取舍。按照游戏/硬件平台的特定需求及限制,经常可以通过微调引擎制作更精美的游戏。

**1.4-不同游戏类型的引擎差异**

通常在某种程度上，游戏引擎是为某游戏类型(genre)而设计的。当然，各种引擎也有很大的重叠部分

**第一人称射击游戏**：FPS是开发技术难度极高的游戏类型之一。能与之相比的或许只有第三人称射击/动作/平台游戏,以及大型多人在线游戏。这是因为FPS要让玩家面对一个精细而超1. 现实的世界时感到身临其境。

1. FPS游戏常会注重技术,例如:
2. 高效地渲染大型三维虚拟世界。
3. 快速反应的摄像机控制及瞄准机制。
4. 玩家的虚拟手臂和武器的逼真动画。·各式各样的手持武器。
5. 宽容的玩家角色运动及碰撞模型,通常使游戏有种“漂浮”的感觉。
6. 非玩家角色(NPC,如玩家的敌人及同盟)有逼真的动画及智能。
7. 小规模在线多人游戏的能力(通常支持多至同时64位玩家在线),以及无处不在的死亡竞赛(death match)游戏模式。

FPS中使用的渲染技术几乎总是经过高度优化,并且按特定场景类型仔细调整过的。例如,室内“地下城爬行(dungeon crawl)”3游戏通常会利用二元空间分割树或基于入口的渲染系统。室外FPS游戏使用其他种类的渲染优化,例如遮挡剔除(occlusion culling)，或游戏在运行前预先把游戏世界分区化(sectorization)，以自动或手动方式设定每个分区是否能见到另一个分区。

当然，要让玩家在超现实游戏世界中有如身临其境,除了具有经优化的高质量图形技术,还需要具备更多条件。在FPS 中,角色动画、音效音乐、刚体物理、游戏内置电影及大量其他技术都必须是最前沿的。因此,这个游戏类型的技术需求是业界里最严格也最全面的。

**平台及其他第三人称游戏**：平台游戏(platformer)是指基于人物角色的第三人称游戏，在这类游戏中，主要的游戏机制是在平台之间跳跃。从技术上来说，平台游戏通常可以和第三人称设射击/动作/历险游戏类型一起考虑。

第三人称游戏和第一人称射击游戏有许多共通之处，但第三人称游戏比较看重主角的能力(ability)及运动模式(locomotion mode)”---运动模式一般指动物的运动，比如行走、跳跃、游泳以及飞行等。除此以外,这种类型游戏的主角化身(avatar)需要高度逼真的全身动画,相比起来,典型的 FPS里主角的“漂浮手臂”的动画是比较简单的。要注意,因为大部分FPS游戏都会有多人在线模式,所以除了第一人称的手臂外往往还需要渲染主角的全身动画。不过,在FPS游戏中,玩家化身的逼真程度一般并不及非玩家角色(NPC),更不能和第三人称游戏的玩家化身相比。

在平台游戏中,游戏主角通常是比较卡通而不是很真实或细腻的。但是,第三人称射击游戏通常使用非常真实的人形玩家角色。这两种类型都需要非常丰富的行为和动画。

第三人称游戏特别注重的技术如下所述。

1. 移动平台、梯子、绳子、棚架及其他有趣的运动模式。
2. 用来解谜的环境元素。
3. 第三人称的“跟踪摄像机”会一直注视玩家角色,也通常会让玩家用手柄右摇杆(在游戏主机上)或鼠标(在PC上)旋转摄像机(虽然在PC上有很多流行的第三人称射击游戏,但平台游戏类型几乎是游戏主机上独有的)。
4. 复杂的摄像机碰撞系统,以保证视点不会穿过背景几何物体或动态的前景物体。

**格斗游戏**：格斗游戏通常是两个玩家控制角色在一个擂台上对打，传统格斗类型游戏注重以下技术：

1. 丰富的格斗动画
2. 准确的攻击判定
3. 能侦测复杂按钮及遥感组合的玩家输入系统
4. 人群或相对静态的背景

由于这些游戏的三维世界比较小,而且摄像机一直位于动作的中心,以往这些游戏只有很少甚至不需要世界细分(world subdivision)或遮挡剔除。同样地,这些游戏不要求使用高阶的三维音频传播模型。

**竞速游戏**：竞速游戏包括所有以在赛道上驾驶车辆或者其他载具为主要任务的游戏，一般而言可以分为几个子类别。

1. 着重模拟的竞速游戏力求模仿真实的驾驶体验
2. 街机竞速游戏偏好娱乐性多于真实感
3. 街头竞速类型，采用可改装车辆
4. 卡丁车类型

竞速游戏通常是非常线性的,这比较像旧式的FPS游戏。但移动速度一般比FPS游戏快许多。因此,这类游戏经常使用非常长的走廊式赛道和环形赛道,有时候加入一些可选分支或捷径。竞速游戏把图形的细节集中在载具(vehicle)、赛道及近景上。但是,卡丁车游戏还需要将足够的渲染及动画资源投放到驾驶角色上。

典型竞速游戏有以下技术特性：

1. 使用多种“窍门”去渲染遥远的背景,例如使用二维纸板形式的树木、山岳和山脉。
2. 赛道通常切开成较简单的二维区域,称为分区(sector)。这些数据结构用来实现渲染优化、可见性判断(visibility determination)，帮助非人类玩家操控车辆的人工智能及路径搜寻,以及解决很多其他技术问题。
3. 第三人称视角摄像机通常追随在车辆后面,第一人称摄像机有时候会置于驾驶舱里。
4. 如果赛道经过天桥底下及其他狭窄空间,必须花精力防止摄像机和背景几何物体碰撞。

**实时策略游戏**：在这类游戏中,玩家在一个广阔的场地里,利用兵工厂策略部署作战单位(battle unit)来试图压倒对手。游戏世界通常会以斜面俯视(oblique top-down view)'的视角显示。

RTS通常不允许玩家改变视角以观看不同距离的景物。这个限制使开发者能在RTS渲染引擎上采用各种优化。

RTS游戏的惯用手法如下：

1. 每个作战单位使用解析度相对较低的模型,使游戏能支持同时显示大量单元
2. 游戏的设计和进行多是在高度场地形(height field terrain)画面上展开的。
3. 除了部署兵力,游戏通常准许玩家在地形上兴建新的建筑物。
4. 用户的互动方式通常为单击及以范围选取单位,再加上包含指令、装备、作战单位种类建筑种类等的菜单及工具栏。

**大型多人在线游戏**：MMOG定义为能同时支持大量玩家(由数千至数十万个)在非常大的持久世界(persistent world)里进行的游戏(持久世界是指其状态能持续一段很长的时间,比特定玩家每次玩的时间长很多)。除了同时在线人数和持久性外,MMOG的游戏体验和小型的多人游戏是相似的。MMOG的子类型有MMO角色扮演游戏(MMORPG)、MMO实时策略游戏(MMORTS)及 MMO第一人称射击游戏(MMOFPS)。

MMOG的核心是一组非常强大的服务器。这些服务器维护游戏世界的权威状态，管理用户的登入登出，也会提供用户间文字对话或者IP电话等服务。几乎所有的MMOG都要求用户定期支付服务费用，也可能在游戏中或游戏外支持小额交易。这些都是开发商的主要收入来源，因此中央服务器最重要的角色可能是处理账单及小额交易。

因为MMOG的游戏场景规模和玩家数量很大，所以MMOG里的图形逼真程度通常稍低于其他游戏

**玩家创作内容**：简单来说就是玩家社区通过游戏本身制作各种不同玩法的mod

**其他游戏类型**：

1. 体育游戏，各主要体育项目是其子类型
2. 角色扮演游戏
3. 上帝模拟游戏
4. 环境或者社会模拟游戏
5. 解密游戏
6. 非电子游戏的移植，比如各种棋牌的移植
7. 基于网页的游戏
8. 其他游戏类型

各种游戏类型有其特殊的技术需求,因此传统上游戏引擎因游戏类型而有些差异。然而,不同游戏类型的技术需求也有很大的共通之处,尤其在单个硬件平台上,共通之处特别多。由于硬件性能的不断提升,因考虑优化而产生的游戏类型差异将会缩小。因此,现在把一项引擎技术应用于不同游戏类型,甚至不同硬件平台,变得越来越可行。

**1.5-游戏引擎概览**

**雷神之锤引擎家族**：一般认为，世界上首个三维第一人称射击游戏是德军总部(1992年)。这款游戏由id Software开发，随后又开发了毁灭战士、雷神之锤1-3.这些引擎在架构上非常相似。因此可以统称为雷神之锤引擎家族，v社的起源引擎也能追溯到雷神之锤的技术

**虚幻引擎**：1998年epic通过unreal闯入fps世界，虚幻2是虚幻竞技场2004的基础。虚幻4则是其下一个进化阶段，它由方便且强大的图形用户界面去制作着色器，又有一个名为Kismet的图形用户界面供编写游戏逻辑。

**Source引擎**：起源是v社的自研引擎。

**DICE的寒霜引擎**：寒霜主要用于战地系列游戏的制作

**CryEngine**：Crytek公司的引擎，用于制作了far cry，战争前线等游戏

**索尼的PhyreEngine**：主要用于制作PlayStation上的游戏

**微软的XNA Game Studio**：主要基于C#以及CLR

**Unity**：主要的特点在于跨平台

**供非程序员使用的二维游戏引擎**：主要特点在于可以使用图形用户界面创作游戏而无需编程语言

**其他商业引擎**：eg.C4

**专有内部引擎**：比如EA的Sage

**开源引擎**：开源三维游戏引擎是由业余及专业开发者制作,并在网上免费发布的。开源(open source)通常意味着源代码可免费获得,并且其开发模式是全部公开的，即任何人都可以对代码做贡献。若有指明授权方式(licensing)，通常都使用GNU通用公共许可证(GNU General PublicLicense,GPL)或GNU 宽通用公共许可证(GNU Lesser General Public License, LGPL)3。GPL允许免费使用其代码,但其衍生作品也必须为GPL,即作品的代码也要免费供他人使用;后者则允许在盈利的产品中使用。此外还有其他免费或半免费授权模式的开源项目。

**1.6-运行时引擎架构**

游戏引擎通常由工具套件和运行时组件两部分组成，整个架构十分复杂，因此游戏引擎是大型软件系统。和所有软件系统一样，游戏引擎也是以软件层构建的。通常上层依赖下层，下层不依赖上层。当下层依赖上层时，成为循环依赖。在任何的软件系统中，都要极力避免循环依赖，不然会导致系统间复杂的耦合，也会使软件难以测试，并妨碍代码重用。本节先讨论运行时引擎架构。

**目标硬件**：它代表用来执行游戏的计算机系统或游戏主机。典型平台包括基于微软Windows、Linux或macOS的PC;移动平台如苹果的iPhone及iPad、谷歌的Android智能手机和平板电脑、索尼的PlayStation Vita及亚马逊的Kindle Fire等;游戏机如微软的Xbox/Xbox 360/Xbox One、索尼的PlayStation/PlayStation 2/PlayStation 3/PlayStation 4、任天堂的NDS/GameCube/Wii/Wii U。

**设备驱动程序**：设备驱动程序是由操作系统或硬件厂商提供的最低阶软件组件。驱动程序负责管理硬件资源，也隔离了操作系统及上层引擎，使上层的软件无须理解不同硬件版本的通信细节差异

**操作系统**：在PC上，OS是一直运行的。操作系统协调一台计算机上多个程序的执行，其中一个程序可能是游戏。操作系统如win，使用时间片方式，使得多个执行中的程序能够共享硬件，这成为抢占式多任务，这意味着PC游戏不能假设拥有硬件的所有控制权，PC游戏需要配合其他系统中的程序。而在游戏主机上，操作系统通常只是一个轻量级的库，链接到游戏的执行文档里。在游戏主机上，游戏通常拥有整台机器，不过现如今的主机的操作系统往往能够中断游戏的执行，接管某些系统资源以显示在线信息，或允许玩家暂停游戏以进入跨界导航菜单。因此，现在游戏机和PC开发的界限正在变窄。

**第三方软件开发包和中间件**：大部分游戏引擎都会借用第三方软件开发包(software development kit，SDK)及中间件(middleware)，SDK提供基于函数或基于类的接口，一般称为应用程序接口(application programming interface，API)。以下会介绍几个例子。

1. 数据结构以及算法

如同任何软件系统，游戏也非常依赖数据结构集合，以及操作这些数据的算法，比如说：

1. STL：C++标准模板库提供了很丰富的代码和算法去管理数据结构、字符串及基于流的输入输出
2. STLport：这是一个可移植的，经优化的STL实现
3. Boost：Boost是非常强大的数据结构及算法库，采用STTL的设计风格
4. Loki：Loki是强的泛型编程模板库

游戏开发者可分为两类:在他们的游戏引擎中使用STL模板库之类的,以及不使用的。一些开发者认为STL的内存分配模式(memory allocation pattern)不高效，会导致内存碎片问题,使STL不能在游戏中使用。一些开发者认为STL的强大和方便性超过它的问题,而且大部分问题实际上可以变通解决。STL在PC上一般可以无碍使用,因为PC上有高级的虚拟内存(virtual memory)系统,使必须谨慎地分配内存变得不那么重要(虽然开发者仍要非常谨慎)。在游戏主机上,只有有限的(甚至没有)虚拟内存功能,而且缓存命中失败(cache miss)的代价极高,因此游戏开发者最好编写自定义的数据结构,保证其是可预期和/或有限的内存分配模式。(在PC上做同样的事情肯定也错不了。)

**图形**：大多数游戏渲染引擎都是建立在硬件接口库上的，例如：

1. Glide是三维图形SDK，专门用于古老的Voodoo显卡。
2. OpenGL是获广泛使用的跨平台三维图像SDK
3. DirectX是微软的三维图像SDK，也是OpenGL的主要对手
4. Libgcm是索尼提供给PS图形硬件的低阶直接接口，在PS上比OpenGL更加高效
5. Edge是由顽皮狗和索尼为PS制作的强大高效渲染及动画引擎
6. Vulkan

**碰撞和物理**：碰撞检测和刚体动力学（在游戏开发社区里被简单称作物理）可由以下知名的SDK提供。

1. Havok是一个流行的工业级物理及碰撞引擎
2. PhysX是另一个流行的工业级物理及碰撞引擎，NVIDIA提供免费下载
3. Open Dynamics Engine（ODE）是知名的开源物理及碰撞引擎包

**角色动画**：市面上有许多商用的角色动画包

1. Granny是Rad Game Tools公司的流行Granny工具套件包含健壮的三维模型导出器，其支持主要的三维建模及东华软件，例如Maya，3ds MAX等。Granny也包括负责读取及操作导出模型和动画数据的运行时库,以及强大的运行时动画系统。
2. Haook Animation:因为游戏角色变得越来越真实,物理和动画之间的分界线变得越来越模糊。制作知名Havok 物理SDK 的公司,决定制作一个附送的动画SDK,使融合物理和动画变得十分容易。
3. Edge:为PS3而设的Edge库是由顽皮狗的ICE团队、美国索尼计算机娱乐(SCE)的工具及技术组、欧洲的索尼高阶技术组联合制作的。Edge包含强大及高效的动画引擎,以及为渲染而设的高效几何处理引擎。

**人工智能**：Kynapse:直至不久前,每个游戏都是以自有方式处理人工智能(artificial intelligence,AI)的。可是, Kynogon公司开发了一个名为Kynapse的中间件SDK1,提供低阶的AI构件,例如,路径搜寻(path finding)、静态和动态物体回避(avoidance)、空间内的脆弱点(vul-nerabilities)辨认(例如,一扇开着的窗可能会有埋伏),以及相当好的AI和动画间接口。

**生物力学角色模型**：Endorphin和Euphoria这两个动画套件，利用真实人类运动的高阶生物力学模型去产生角色动作

如前面提及的,物理和动画之间的分界线开始变得模糊。软件包,如Havok Animation,尝试用传统方式结合动画和物理,先由动画师利用Maya之类的工具制作基本动作,再在执行时利用物理去扩充那些动作。生物力学模型同时考虑了角色的中心，体重分布，以及人类的在地心引力及其他作用力影响下，会如何平衡及运动

**平台独立层**：大多数游戏引擎需要运行于不同的平台上。像艺电、Activision Blizzard这样的公司,经常需要游戏支持多个目标平台,从而覆盖最大的市场。通常，只有第一方工作室，例如索尼的顽皮狗和Insomniac工作室,无须为每个游戏同时支持两个或以上的目标平台。因此,大部分游戏引擎的架构都有一个平台独立层(platform independence layer)，如图1.20所示。平台独立层在硬件、驱动程序、操作系统及其他第三方软件之上,以此把其余的引擎部分和大部分底层平台隔离。

平台独立层包装了常用的标准C语言库、操作系统调用及其他基础API,确保包装了的接口在所有硬件平台上均为一致。这是必需的,因为不同平台间有不少差异,即使所谓的“标准”库,如标准C语言库,也有平台差异。

**核心系统**：游戏引擎以及其他大规模复杂C++应用软件,都需要一些有用的实用软件(utility)，本书把这类软件称为核心系统(core system)。以下是核心系统层的一些常见功能。

1. 断言(assertion):断言是一种检查错误的代码。断言会插入代码中捕捉逻辑错误或找出与程序员原来假设不符的错误。在最后的生产版本中,一般会移除断言检查。
2. 内存分配:几乎每个游戏引擎都有一个或多个自定义的内存分配系统,以保证高速的内存分配及释放,并控制内存碎片所造成的负面影响
3. 数学库:游戏本质上就是高度数学密集的。因此,每个游戏引擎都有一个或多个数学库,提供矢量(vector)、矩阵(matrix)、四元数(quaternion)旋转、三角学(trigonometry)、直线/光线/球体/平截头体(frustum)等的几何操作、样条线(spline)操作、数值积分(numericalintegration)、解方程组,以及其他游戏程序员需要的功能。
4. 自定义数据结构及算法:除非引擎设计者想完全依靠第三方软件包，如STL,否则引擎通常要提供一组工具去管理基础数据结构(链表、动态数组、二叉树、散列表等)，以及算法(搜寻、排序等)。这些数据结构及算法有时需要手工编码,以减少或完全消除动态内存分配,并保证在目标平台上的运行效率为最优。

**资源管理器**：每个游戏引擎都有某种形式的资源管理器,提供一个或一组统一接口,去访问任何类型的游戏资产及其他引擎输入数据。有些引擎使用高度集中及一致的方式(例如虚幻的包、OGRE的ResourceManager类)。其他引擎使用专设(ad hoc)方法，比如让程序员直接读取文件这些文件可能来自磁盘,也可能来自压缩文件(如雷神之锤引擎使用的是PAK文件)。

**渲染引擎**：在任何游戏引擎中，渲染引擎都是最大及最复杂的组件之一。渲染器有很多不同的架构方式，虽然没有单一架构方式，但是大多数现在的渲染引擎都有一些通用的基本设计哲学，这些哲学大部分是由底层三维图形硬件驱动形成的。

渲染引擎的设计通常采用分层架构(layered architecture)，以下会使用这些行之有效的方法。

1. 低阶渲染器(low-level renderer)包含引擎中全部原始的渲染功能。这一层的设计着重于高速渲染丰富的几何图元 (geometric primitive)集合,并不太考虑那些场景部分是否可见。这个组件可以拆分为几个子组件,以下分别讨论。
2. 图形设备接口：使用图形SDK,如 DirectX及OpenGL,都需要编写不少代码去枚举图形设备,初始化设备,建立渲染表面(如后台缓冲、模板/stencil 缓冲）等。这些工作通常由笔者称为图形设备接口(graphics device interface)的组件负责(然而各个引擎都有自己的术语)。在PC游戏中,程序员需编写代码把渲染器整合到Windows消息循环中。通常要编写“消息泵(message pump)”去处理等待中的Windows消息,其余时间则尽快不断地执行渲染循环。这样做,会使游戏的键盘轮询和渲染器的屏幕更新挂钩。这种耦合令人不快,我们可以再进一步使这种依赖最小化。
3. 其他渲染器组件：其他渲染器组件﹐低阶渲染器的其他组件一起工作,目的是要收集需提交的几何图元(geo-metric primitive,又称为渲染包, render packet)。几何图元包括所有要绘制之物,如网格(mesh) 、线表(line list)、点表(point list)、粒子(particle)、地形块(terrain patch)、字符串等。最后,把收集到的图元尽快渲染。
4. 场景图/剔除优化：低阶渲染器绘制所有被提交的几何图形,不太考虑那些图形是否确实为可见的,除了使用背面剔除(back-face culling)和摄像机平截头体的剪切平面。一般需要较高层次的组件,才能基于某些可视性判别算法去限制提交的图元数量。理论上，低阶渲染器无须知道其上层使用哪种空间分割或场景图。因此，不同的游戏团队可以重用图元提交代码,并为个别游戏的需求精心制作潜在可见集判别系统。开源渲染引擎OGRE'正是运用这一原则的好例子。OGRE提供即插即用的场景图架构。游戏开发者可以选择其中一个已实现的场景图设计,或是自定义一个。
5. 视觉效果：当代引擎支持广泛的视觉效果，包括：
6. 粒子系统，用作烟、火、水花等
7. 贴花系统，用作弹孔、脚印等
8. 光照贴图及环境贴图
9. 动态阴影
10. 全屏后期处理效果，在渲染三维场景至屏外缓冲后使用
11. 高动态范围(HDR)色调映射以及敷霜效果(bloom)
12. 全屏抗锯齿(FSAA)
13. 颜色矫正及颜色偏移效果，包括略过漂白、饱和度、去饱和度等

游戏引擎常有效果系统组件,专门负责管理粒子、贴花及其他视觉效果的渲染需要。粒子和贴花系统通常是渲染引擎的独立组件,并作为低阶渲染器的输入端。另一方面,渲染引擎通常在内部处理光照贴图、环境贴图、阴影。全屏后期处理效果可以在渲染器内实现,或在运行于渲染器输出缓冲的独立组件内实现。

1. 前端：大多数游戏为了不同目的，都会使用一些二维图形去覆盖三维场景，这些目的包括：
2. 游戏的平视显示器(HUD)
3. 游戏内置菜单、主控台、其他开发工具
4. 游戏内置图形界面(GUI)让玩家操作角色装备，配置战斗单元。或完成其他复杂的游戏任务

前端图形这类二维图形通常会用附有纹理的四边形(quad)(一对三角形)结合正射投影(orthographic projection)来渲染。另一种方法是用完全三维的四边形公告板(bill-board)渲染,这些公告板能一直面向摄像机。

这一层也包含了全动视频(full-motion video,FMV)系统,该系统负责播放之前录制的全屏幕电影(可以用游戏引擎录制,也可以用其他渲染软件录制)。

另一个相关的系统是游戏内置电影(in-game cinematics,IGC)系统,该组件可以在游戏本身以三维形式渲染电影情节。例如,玩家走在城市中,两个关键角色的对话可用IGC实现。IGC可能包括或不包括玩家角色，它可以故意暂停游戏,在这期间玩家不能控制角色; IGC 也可悄悄地整合在游戏过程中,玩家甚至不会发觉有IGC在运行。

**剖析和调试工具**：游戏是实时系统,因此,游戏工程师经常要剖析游戏的性能,以便优化。此外,内存资源通常容易短缺,开发者也要大量使用内存分析工具(memory analysis tool)。调试功能包括调试用绘图、游戏内置菜单、主控台,以及能够录制及回放游戏过程的功能,方便测试和调试。多数游戏会加入自制的剖析及调试工具,常包括以下功能。

1. 手工插入测量代码,为某些代码计时。
2. 在游戏进行期间,在屏幕上显示性能统计数据。
3. 把性能统计写入文字或Excel文件。
4. 计算引擎及子系统所耗的内存,并显示在屏幕上。
5. 在游戏过程中或结束时,把内存使用率、最高使用率、泄漏等数据统计输出。
6. 允许在代码内布满调试用打印语句(print statement)，可以开关不同的调试输出种类,并设置输出的冗长级别(verbosity level)。
7. 游戏事件录制及回放的能力。这很难做得正确,倘若做对,便是追踪 bug 的非常宝贵的工具

PlayStation 4提供了一个强大的核心转储(core dump)设施去协助程序员调试崩溃的问题。PlayStation 4 会一直录制最后15秒的游戏视频,令玩家可以按下手柄上的分享按钮去分享游戏经历。因为有此特性, PS4的核心转储功能除了自动为程序员提供有关程序崩溃时的完整调用堆栈信息之外,还包括崩溃那一刻的屏幕截图,以及展示崩溃前15秒发生了什么事情的视频。当游戏崩溃时,其核心转储文件会被自动上传至游戏开发者的服务器,甚至在游戏发行后也会做这件事情。这些设施彻底改革了崩溃分析及修复的工作。

**碰撞和物理**：碰撞检测对于每个游戏来说都十分的重要，没有碰撞检测的话，物体会互相穿透，并且无法在虚拟世界中进行合理的互动。一些游戏包含真实或者半真实的动力学模拟。这在游戏业界被称为物理系统，但是比较正规的术语是刚体动力学模拟。因为游戏中通常只考虑刚体的运动，以及产生运动的力和力矩。研究运动的物理分支是运动学，而研究力和力矩的是动力学。

碰撞和物理系统一般是紧密联系的，因为当碰撞发生时，碰撞几乎总是物理积分及约束满足逻辑来解决的。

**动画**：含有机或半有机角色的游戏，就需要动画系统。游戏会用到五种基本动画：

1. 精灵/纹理动画
2. 刚体层次结构动画
3. 骨骼动画
4. 顶点动画
5. 变形目标动画

骨骼动画让动画师使用相对简单的骨头系统,去设定精细三维角色网格的姿势。当骨头移动时,三维网格的顶点就跟随移动。虽然有些引擎支持变形目标及顶点动画,但在现今的游戏中,骨骼动画仍然是最流行的动画方式。

骨骼网格渲染组件是连接渲染器和动画系统的桥梁。虽然这些组件能非常紧密地合作,但它们的接口还是有明确定义的。动画系统生成骨骼中所有骨头的姿势,这些姿势以矩阵调色板(matrix palette)形式传至渲染引擎。之后,渲染器利用矩阵表转换顶点,每个顶点用一个或多个矩阵生成最终混合顶点位置。此过程称为蒙皮(skinning)

当使用布娃娃(ragdoll)时，动画和物理系统便产生紧密耦合。布娃娃是无力的(经常是死了的)角色,其运动完全由物理系统模拟。物理系统把布娃娃当作受约束的刚体系统,用模拟来决定身体每部分的位置及方向。动画系统计算渲染引擎所需的矩阵表,用来在屏幕上绘制角色。

**人体学接口设备**：游戏的玩家输入一般都来源于多个人体学接口设备(HID)，比如：  
1. 键盘和鼠标

1. 游戏手柄
2. 其他的游戏控制器，例如方向盘、鱼竿、跳舞毯等

该组件有时被称作玩家输入/输出(player I/O)组件,因为除了输入功能,一些人体学接口设备也提供输出功能,如游戏手柄的力反馈/震动

在架构HID引擎时,通常让个别硬件平台游戏控制器的低阶细节与高阶游戏操作脱钩。HID引擎从硬件取得原始数据，为控制器的每个摇杆(stick)设置环绕中心点的死区(deadzone)2，去除按钮抖动(debounce),检测按下和释放按钮事件,演绎加速计(accelerometer)的输人并使该输入平滑(例如,来自PlayStation Dualshock手柄),以及其他处理等。HID引擎通常允许玩家调整输入配置,即自定义硬件控制到逻辑游戏功能的映射。HID引擎也可能包含一个系统,负责检测弦(chord,即数个按钮一起按下)、序列(sequence,即按钮在时限内顺序按下)、手势(gesture，即按钮、摇杆、加速计等输入的序列)。

**音频**：游戏引擎中的音频和图形同样重要，然而相对于渲染、物理、动画、AI以及游戏性而言，音频通常容易被忽略。然而，没有出色的音频引擎，就没有完整的优秀游戏。

音频引擎的功能差异很大。Quake和虚幻的音频引擎只提供了非常基本的功能,一些游戏团队会为这些引擎加入自定义功能,或用内部方案替换。虚幻引擎4提供了相当稳健的三维音频渲染引擎，它的功能集比较有限,许多游戏团队很可能希望对它做出修改,为他们的游戏加入一些专门的高级功能。微软为DirectX平台(PC、Xbox 360和 XboxOne)提供了一个优秀的音频工具包,名为XACT,以及在运行时提供了功能丰富的XAudio2及X3DAudio API。艺电也开发了内部的音频引擎SoundR!OT。美国索尼计算机娱乐(SCEA)向其第一方游戏工作室,如顽皮狗,提供了一个强大的三维音频引擎Scream,这个引擎已应用在多个PS3作品上,如顽皮狗的《神秘海域:德雷克的诡计》和《最后生还者》等。然而,即使游戏团队使用既有的音频引擎,开发每个游戏时仍然需要大量的定制软件开发、整合工作及需注意的细节,才可以制作出有高质量音频的最终产品。

**在线多人/网络游戏**：许多游戏可供多位游戏游玩于同一个虚拟世界里。多人游戏最少有如下四种基本形式：

1. 单屏多人(single-screen multiplayer):两个或以上的 HID(游戏手柄、键盘、鼠标等)接到一台街机、PC、游戏主机上。多位玩家角色同聚于一个虚拟世界,一台摄像机将所有角色置于画面中。例子有《任天堂明星大乱斗》(Super Smash Brothers)、《乐高星球大战》(LegoStar Wars)、《圣铠传说》(Gauntlet)。
2. 切割屏多人(split-screen multiplayer):多个角色同聚于一个虚拟世界,多个HID连接到一台游戏机器,但每个角色有自己的摄像机。画面分割成多个区域,使每位玩家可以看到自己的角色。
3. 网络多人(networked multiplayer):多台计算机或游戏主机用网络连接在一起,每台机器接待一位玩家。
4. 大型多人在线游戏(massively multiplayer online game,MMOG):数百至数千位玩家在一个巨大、持久(persistent)、在线游戏世界里玩。这些虚拟世界由强大的服务器组运行。

多人游戏和单人游戏有许多相似的地方。然而,支持多人游戏,会极大地影响某几个游戏引擎组件的设计。游戏世界对象模型、人体学接口设备系统、玩家控制系统、动画系统等都会受到影响。把一个现有的单人引擎改装为多人引擎,并非不可能,但会是一个让人望而生畏的任务。然而,有些游戏团队仍可完成这个任务。但是如果可以,最好还是在项目之初就设计多人游戏的功能。

有趣的是,如果进行反向思维——改装多人游戏为单人游戏,问题就再简单不过了。事实上,许多游戏引擎把单人游戏模式当作多人游戏的特例,换言之,单人游戏模式是一个玩家参与的多人游戏。一个知名例子就是雷神之锤引擎的客户端于服务器之上(client-on-top-of-server)模式。运行单人游戏模式时,该可执行文件在单台PC上执行,这台PC同时作为客户端和服务器。

**游戏性基础系统**：游戏性(gameplay)这一术语是指:游戏内进行的活动、支配游戏虚拟世界的规则(rule),玩家角色的能力(也称为玩家机制, player mechanics)、其他角色和对象的能力、玩家的长短期目标(goal and objective)。游戏性通常用两种编程语言实现,除了使用引擎其余部分采用的原生语言,也可用高阶脚本语言,又或者两者皆用。为了连接低阶的引擎子系统和游戏性代码,多数游戏引擎会引人一个软件层,因无标准术语,笔者称之为游戏性基础层(gameplay foundationlayer)。该软件层提供一组功能,以方便实现以上所述的游戏专有逻辑。

1. 游戏世界和游戏对象模型

游戏性基础层引入游戏世界的概念,游戏世界包含动态元素及静态元素。游戏世界的内容通常用面向对象方式构建(多数使用面向对象语言,但也有例外)。在本书中,组成游戏的对象类型集合,称为游戏对象模型(game object model)。游戏对象模型为虚拟游戏世界里的各种对象集合提供实时模拟

典型的游戏对象包括：

1. 静态背景几何物体,如建筑、道路、地形(常为特例)等。
2. 动态刚体,如石头、饮料罐、椅子等。
3. 玩家角色(player character, PC)。
4. 非玩家角色(non-player character, NPC)。武器。
5. 抛射物(projectile)。
6. 载具(vehicle)。
7. 光源(可在运行时用于动态场景,也可离线用于静态场景)。
8. 摄像机。

游戏对象模型与软件对象模型(software object model)紧密结合,并且渗透于整个引擎中。软件对象模型是指,用于实现面向对象软件的一组语言特征、原则(policy)、惯例(convention)。在游戏引擎的语境中,软件对象模型要回答以下问题。

1. 游戏引擎是否使用面向对象方式设计?
2. 使用什么编程语言? C、C++、Java还是OCaml?
3. 怎样组织静态类层阶?一个巨大的层阶,或很多低耦合组件?
4. 使用模板(template)及基于原则设计(policy-based design)，或传统的多态(polymor-phism)?
5. 如何参考对象?简明的旧式指针﹑智能指针(smart pointer)还是句柄(handle)?
6. 如何独一无二地标识对象?只凭内存地址、用名字,或用全局统一标识符(global uniqueidentifier,GUID)?
7. 如何管理对象的生命周期?
8. 如何随时间模拟对象的状态
9. 事件系统

游戏对象总要和其他对象通信。有多种方法可完成通信,例如,对象要发消息,可简单调用接收对象的成员函数。事件驱动架构(event-driven architecture),常用于典型图形用户界面,也常用于对象间通信。在事件驱动系统里,发送者建立一个称为事件(event)或消息(message)的小型数据结构,其中包含要发送的消息类型及参数数据。事件传递给接收对象时,调用接收对象的事件处理函数(event handler function)。事件也可存储于队列上,以推迟在未来处理。

1. 脚本系统

很多游戏引擎使用脚本语言,使游戏独有游戏性的规则和内容,能更容易、更快地进行开发。没有脚本语言,每次改动引擎中的游戏逻辑或数据结构,都必须重新编译链接方可执行程序。若将脚本语言集成至引擎,要更改游戏逻辑或数据，只需修改脚本代码并重新载入即可。有些引擎允许在游戏运行中重载脚本,其他引擎则需要终止后才能重编脚本。但两种形式所需的作业时间,总比重编、重链程序快得多。

1. 人工智能基础

一般而言,人工智能(artificial intelligence,AI)一直是为个别游戏专门开发的软件,一般并不隶属于游戏引擎。但近期游戏开发商找到一些差不多每个AI 系统都共有的模式,使这些基础部分逐渐进入游戏引擎的范畴。

有一家名为Kynogon的公司开发了一个名为Kynapse的中间件SDK,它提供的基础层技术可用来开发商业水平的游戏AI。此技术被Autodesk收购后,被Kynapse原班人马重新设计的AI中间件Gameware Navigation 所取代。此SDK提供了基础的AI构件,例如导航网格(navigation mesh)生成、寻路、静态/动态物体回避、游戏空间中的弱点识别(如打开的窗户可能会被伏击),以及AI和动画之间定义明确的接口。Autodesk也提供了一个视觉化编程系统及运行时引擎Gameware Cognition,可与Gameware Navigation结合使用,令制作规模宏大的游戏AI系统变得容易多了。

**个别游戏专用子系统**：在游戏性基础软件层和其他低阶引擎组件之上，需要游戏程序员和设计师合作实现游戏本身的特性。游戏性系统通常变化很大,并且针对特定游戏来开发。这些系统包括但不局限于玩家机制、多种游戏内摄像机、控制非玩家角色的AI、武器系统、载具等。如果可以清楚地分开引擎和游戏,这条分界线会位于特定游戏专用子系统和游戏性基础软件层之间。实际上,这条分界线永远不会是完美的。一些游戏的特定知识,总会向下渗透到游戏基础软件层中,更有甚者,会延伸至引擎核心。

**1.7-工具及资产管道**

游戏引擎需要读取大量数据，数据形式包括游戏资产、配置文件、脚本等。

**数字内容创作工具**：游戏本质上是多媒体应用。游戏引擎的输入数据形式广泛,例如三维网格数据、纹理位图、动画数据、音频文件等。所有源数据皆由美术人员使用数字内容创作(digital content creation,DCC)应用软件制作。

DCC应用软件通常是为制作某一类数据而开发的,但也有些工具能制作多种数据。例如,Autodesk公司的Maya和3ds Max常用来制作三维网格及动画数据,Adobe公司的Photoshop和其家族成员用于创作及修改点阵图(纹理), Sound Forge是制作音频片段的流行工具。然而,有些游戏数据并不能用现成的DCC应用软件去制作。例如,多数游戏引擎提供专门的编辑器,用来设计游戏世界。也有一些引擎使用现成工具去编辑游戏世界。笔者见过有些游戏团队,使用3ds Max或Maya 作为世界编辑工具,有些团队甚至会开发插件去辅助游戏开发者在这些软件上工作。若去问一些较有经验的游戏开发者,他们大多数可能会记得,曾使用简单的位图编辑器去制作地形高度图(height field),或直接把世界布局手动写到文本文件里。工具不需要精美,只要能完成工作即可。话虽如此,但游戏团队要想及时开发高完成度的产品,工具必须相对易用,并且绝对可靠。

**资产调节管道**：DCC应用软件所使用的数据格式,鲜有适合直接用于游戏中的,原因有二。

1. DCC软件在内存中的数据模型,通常比游戏所需的复杂得多，游戏引擎通常只需这些信息的一小部分就能在游戏中渲染模型。
2. 在游戏中读取DCC软件格式的文件,其速度通常过慢。而有些格式更是不公开的专有格式。

因此,用DCC 软件制作的数据,通常要导出为容易读取的标准格式或自定义格式,以便在游戏中使用。

从 DCC到游戏引擎的管道,有时候被称为资产调节管道(asset conditioning pipeline)。每个引擎都有某种形式的资产调节管道。

**三维模型/网格数据**：游戏中所见到的几何体通常是由三角网络组成的。在一些早期的游戏中也会使用到一种称之为笔刷(brush)的几何体

1. 三维模型(网格)：网格是由三角形和顶点(vertex)组成的复合形状,也可以使用四边形和高次细分曲面(higher order subdivision surface)建立可渲染的几何体。但现在的图形硬件,几乎都是专门为渲染光栅化三角形而设计的,渲染前必须把所有图形转换为三角形。每个网格通常使用一种或多种材质(material)，以定义其视觉上的表面特性,如颜色、反射度(reflectivity) 、凹凸程度(bumpiness)、漫反射纹理(diffuse texture)等。在本书中,以“网格”一词代表可渲染的图形,并以“模型”一词代表一个组合对象,它可能包含多个网格、动画数据和为游戏而设置的其他元数据(metadata)。网格通常在三维建模软件里制作，如 3ds Max 、 Maya 、 SoftImage。Pixologic公司的ZBrush是一个强大且流行的工具,可用直观方式制作超高分辨率的模型,并向下转为有法线贴图(nor-mal map)的低分辨率模型,以模拟高频率的细节。我们必须编写导出器(exporter)才能从DCC工具获取数据并存储为引擎可读的格式。DCC软件提供许多标准或半标准的导出格式,但通常都不完全适合游戏使用(COLLADA可能是例外)。因此游戏团队经常要建立自定义格式及专门的导出器。
2. 笔刷几何图形笔刷几何图形(brush geometry)由凸包(convex hull)集合定义,每个凸包则由多个平面定义。笔刷通常直接在游戏世界编辑器中创建及修改凸包。这种制作可渲染几何图形的方法比较“土”,但仍然应用在一些引擎中。优点是制作方法简单迅速，便于游戏设计师用来建立粗糙关卡，制作原型，既可以用作碰撞体又可用作可渲染几何图形；缺点是分辨率低，难以制作复杂图形，不能支持有关节的物体或运动的角色。

**骨骼动画数据**：骨骼网格(skeletal mesh)是一种特殊网格,为制作关节动画而绑定到骨骼层次结构(skele-tal hierarchy)之上。骨骼网格在看不见的骨骼上形成皮肤,因此,骨骼网格有时候又被称为皮肤(skin)。骨骼网格的每个顶点包含一组关节索引(joint index)，表明顶点绑定到骨骼上的哪些关节。每个顶点也包含一组关节权重(joint weight),决定每个关节对该顶点的影响程度。游戏引擎需要三种数据去渲染骨骼网络：

1. 网格本身
2. 骨骼层次架构，包含关节名字、父子关系、当网格绑定到骨骼时的姿势
3. 一个至多个动画片段，指定关节如何随时间而动

网格和骨骼通常由DCC软件导出成单个数据文件。可是,如果多个网格绑定到同一个骨骼,那么骨骼最好导出成独立的文件。而动画通常是分别导出的,特定时刻可只将需要的动画载入内存。然而,有些引擎支持将动画库(animation bank)导出至单个文件,有些引擎则把网格、骨骼、动画全部放到一个庞大的文件里。

未优化的骨骼动画以每秒30帧的频率,对骨骼中每个关节(像真的人型角色可能有500个或以上)采样(sample)，记录成一串4x3矩阵。因此,动画数据生来就是内存密集的,通常会用高度压缩的格式存储。各引擎使用的压缩机制各有不同,有些是专有的。为游戏准备的动画数据,并无单一标准格式。

**音频数据**：音频片段(audio clip)通常由Sound Forge或其他音频制作工具导出,有不同的格式和采样率(sampling rate)。音频文件可为单声道(mono)、立体声(stereo)、5.1、7.1或其他多声道配置(multichannel configuration)。Wave文件(.wav)最普遍,但其他格式如 PlayStation 的自适应差分脉冲编码(ADPCM)文件(.vag及.xvag)也是很常见的。音频文件通常组织成音频库(audio bank)，以方便管理、容易载入及串流。

**粒子系统数据**：当今的游戏采用复杂的粒子效果(particle effect)。粒子效果由专门制作视觉特效的设计师制作。一些第三方工具,如Houdini,可制作电影级别的效果。可是,大部分游戏引擎不能渲染Houdini制作的所有效果。因此,多数游戏引擎有自制的粒子效果编辑工具，只提供引擎支持的效果。定制的编辑器,也可以让设计师看到与游戏一模一样的效果。

**世界编辑器**：游戏引擎的所有内容都集合在游戏世界中。以笔者所知,并没有商用游戏世界编辑器(worldeditor)(即和Max或Maya软件等同的游戏世界版本)。然而,不少商用游戏引擎提供优良的世界编辑器。

1. 不同版本的Radiant 游戏编辑器1,应用在基于Quake技术的游戏引擎上。
2. 《半条命2》的Source引擎提供名为Hammer的世界编辑器。
3. UnrealEd是虚幻引擎的世界编辑器。这款强大的工具同时也作为资产管理工具,管理引擎支持的所有资产类型。

优秀的游戏世界编辑器虽难以编写,但它却是优秀游戏引擎的极重要部分

**资源数据库**：游戏引擎需要处理许多种资产类型,例如可渲染几何体、材质、纹理、动画数据,以及音频等。这些资产的部分定义来自美术人员，例如他们使用 Maya、Photoshop或Sound Forge等工具所制作的原始数据。然而,每种资产也带有大量的元数据(metadata)。例如,当动画师在Maya中制作了一个动画片段,其元数据为资产调节管道及最后的游戏引擎提供以下信息:

1. 用于在运行时识别该动画片段的唯一标识符。
2. 原始Maya (.ma或.mb)文件名及路径。
3. 帧范围(frame range)——动画开始及结束的帧。
4. 动画是否用于循环播放。
5. 动画师所选择的压缩技术及程度。(有些资产能高度压缩且不会在视觉品质上有影响,而另一些可能需要较少压缩或不压缩以保持在游戏中能正常显示。)

每个游戏引擎都需要某种数据库去处理游戏资产所带的元数据。可以使用真正的关系数据库如MySQL或Oracle来实现,也可以使用一组文本文件去实现,并以Subversion 、 Perforce或Git等版本控制系统管理。在本书中我们称这些元数据为资源数据库(resource database)。

**一些构建工具的方法**：可用不同方式去构建游戏引擎工具套装。一些工具可能是独立的软件,一些工具可能构建在运行时引擎使用的低阶软件层之上,还有一些工具可能嵌人游戏本身。有些类型的工具越来越倾向采用基于网页的用户界面。

1.基于网页的用户界面：对某些类型的游戏开发工具而言,基于网页的用户界面(web-based user interface)很快变成常态。在顽皮狗,我们使用大量基于网页的UI。顽皮狗的本地化工具是本地化数据库的前端入口。而Tasker也是一个网页界面,供所有顽皮狗员工创建、管理、安排时间、交流、协作游戏制作中的开发任务。另一个Connector 网页界面作为多种调试信息流的窗口,这些信息流是游戏引擎在运行时产生的。游戏会把其调试文本按多个具名的频道分拆,每个频道关联到某个引擎系统(动画、渲染、AI、声音等)。这些数据流由轻量的Redis数据采集。最后,用户通过基于浏览器的 Connector 界面去方便地检视、过滤这些信息。

相对于传统的独立GUI应用,基于网页的UI有多种优势。首先,相比以Java、C#或C++等语言开发的独立应用,开发及维护网页应用通常更简易、更快捷。网页应用不需特别安装,用户只需要一个兼容的网页浏览器即可。更新网页界面时也可直接推送给用户,无须安装步骤,用户只需刷新网页或重启浏览器便可获得更新。网页界面还迫使我们把工具设计成主从式架构(client-server architecture)。这种设计使我们可以把工具发布给更广的用户群。例如,顽皮狗的本地化工具直接供全球的外包合作方使用,他们通过这个工具向我们提供语言翻译服务。当然,独立工具仍有它们的价值,特别是一些涉及三维可视化的特殊GUI。若你的工具只需要让用户编辑表格或表列数据,基于网页的工具也许是最佳选择。

1. **专业工具**

**2.1版本控制**

版本控制系统(version control system)允许多位开发者在同一组文件上工作。由于版本控制系统记录了每个文件的历史,所以它可以追踪文件中的每个改动,需要时还可以把改动还原。版本控制系统允许多位用户同时修改文件,甚至修改同一个文件,并避免互相破坏成果。因为版本控制系统主要供程序员管理源代码使用,所以有时候又称其为源代码控制(source control)。版本控制系统也可以用来管理其他类型的文件,一般以文本为佳,下文将探讨其中的原因。许多游戏工作室使用单一版本控制系统,同时管理文本类型的源代码,以及以二进制文件为主的游戏资产,如纹理、三维网格、动画、音频文件等。

**为何使用版本控制**：多位工程师组成团队合作开发软件时,版本控制至关重要。版本控制系统有以下功能。

1. 提供中央版本库(repository)，工程师们可以共享其中的代码。
2. 保留每个源文件的所有更改记录。
3. 提供为某些版本加上标签的机制,供以后提取已加标签的版本。
4. 允许代码从主生产线上建立分支(branch)。这一功能经常用来制作示范程序,或是为较旧的软件版本制作补丁(patch)。

源代码控制系统甚至在单人开发的项目里也有所应用。单人开发的项目虽然用不上多人开发的功能,但是其版本控制功能,如维护历史修改记录、为版本添加标签、建立示范程序/补丁的分支、追踪缺陷等,仍然是非常有用的。

**常见的版本控制系统**：常见的版本控制系统如下：

1. SCCS和RCS:源代码控制系统(Source Code Control System, SCCS)和版本控制系统(Re-vision Control System，RCS)是两个古老的版本控制系统。两者皆使用命令行界面,主要用于UNIX 上。
2. CVS:并发版本管理系统(Concurrent Version System,CVS)是高强度、专业级、基于命令行接口的版本控制系统,原本建立在RCS 之上(但CVS 现在已成为独立工具)。CVS在UNIX上应用较多,但在其他开发平台如微软Windows也能使用。CVS是开源的,并按GPL授权。CVSNT (也称为 WinCVS)是一个原生的 Windows实现,基于CVS 并和CVS兼容。
3. Subversion: Subversion (简称SVN)是一个开源版本控制系统,其目标是取代并改进CVS。因为Subversion开源且免费,是个人项目、学生项目和小工作室的首选。
4. Git: Git是开源版本控制系统,用于许多受人敬佩的项目,包括Linux内核。在Git 开发模型里,程序员把文件的变更提交到一个分支上。之后，该程序员可以轻易地把其修改合并到任何一个分支上,因为Git“知道”如何回溯文件的区别(diff)，并把区别重新应用在新的基修订版(base revision)上,这个过程Git 将之称为衍合(rebasing)。此开发模型使Git在处理多个代码分支时非常高效和快捷。有关Git的更多信息可参考官网。
5. Perforce:Perforce是专业级的源代码控制系统,同时支持基于文本和GUI的接口。Perforce成名之处在于其变更列表(changelist)的概念。变更列表,指被视为同一个逻辑单元而进行修改的源文件集合。变更列表会以原子方式(atomically)签入(check-in)版本库,即,要么整个变更列表成功提交,要么没有东西提交进去。许多游戏公司使用Perforce,包括顽皮狗和艺电。
6. NzN Alienbrain: Alienbrain是针对游戏产业而特别设计的强大版本控制系统,具有丰富的功能。其最著名的特点是支持包含文本及二进制游戏资产的海量数据库,并配合可定制的用户界面,以针对特定的人员,如美术设计师、制作人及程序员等。
7. ClearCase: ClearCase是专业级的源代码控制系统,是为超大规模的软件项目而开发的。ClearCase功能强大,并且提供独特的用户接口，以扩展Windows资源管理器的功能。笔者未曾见过游戏业界内使用ClearCase，可能是因为其价格较为昂贵。
8. 微软的 Visual SourceSafe: SourceSafe是轻量级的源代码控制软件包,已成功地应用于一些游戏项目中。

**2.2-IDE**

编译式语言，如C++。需要使用编译器和链接器，将源代码转换成可执行程序。

**源文件、头文件及翻译单元**：用C+＋编写的程序由源文件(source file)所组成。常见的C++源文件的扩展名为.c、.cc、.cxx和.cpp,这些文件中包含程序的大量源代码。因为编译器每次只翻译一个C++源文件至机器码,所以在技术上,源文件被称为翻译单元(translation unit)。

有一种特殊的源文件被称为头文件(header file)。头文件通常用于在多个翻译单元之间分享信息,例如类型声明及函数原型。C++编译器并不“了解”头文件,实际情况是,C+＋预处理器(preprocessor)预先把每个#include语句替换为相对应的头文件内容,然后再把翻译单元送交编译器。这是头文件和源文件之间一个细微但非常重要的区别。从程序员角度来看,头文件是独立的文件,但多亏有预处理器把头文件展开,编译器接收到的才都是翻译单元。

**程序库、可执行文件及动态链接库**：编译翻译单元后,输出的机器码会存储在对象文件(在Windows下采用.obj扩展名,在基于UNIX的系统里则是.o)中。对象文件中的机器码具有如下特征。

1. 可重定位的：未决定代码的内存地址
2. 未链接的：未解决的外部函数参考，以及翻译单元外定义的全局数据

对象文件可以集合成程序库(library)。程序库只是一个简单的存档(archive),像ZIP或tar文件一样,包含零到多个对象文件。程序库只是为方便而设置,允许把大量的对象文件集合成单个易用的文件。

链接器把对象文件和程序库链接成可执行文件(executable)。可执行文件包含完全解析的机器码,操作系统可载入及执行这些机器码。链接器的工作包括:

1. 计算全部机器码的最终相对地址，即当程序执行时机器码在内存中的分布
2. 确保正确地解析每个翻译单元（对象文件）中地所有外部函数参考和全局数据

可执行文件里的机器码仍然是浮动的，即文件中的所有指令和数据地址相对于一个任意的基址,而非绝对地址。直至程序载入内存,在执行之前,程序的最终绝对基址才会决定下来。

动态链接库(dynamic linked library, DLL)是一种特殊的库,其行为像正常的静态链接库和可执行文件的混合体。DLL的行为像库,因为它包含函数,供其他多个不同的可执行文件调用。然而, DLL的行为也像可执行文件,因为操作系统能独立地载入DLL,而且DLL可包含启动及终止代码,其执行方式和C++可执行文件的main ()函数相似。

使用了DLL的可执行文件含有未完全链接(partially linked)的机器代码。在最后的可执行文件中,已解析大多数函数及数据参考,但是存于DLL的函数和数据参考则维持未链接的状态。当运行可执行文件时,操作系统需要解析所有未链接的函数。在此过程中,操作系统会找出合适的DLL文件,若该DLL文件不在内存中则要载入,之后需要修正一些内存地址。载人动态链接库是操作系统非常重要的功能,因为这样就可以只更新个别 DLL,而不需要更新使用到这些DLL的可执行文件。

**生成配置**：C/C++的预处理器、编译器和链接器都提供了大量选项,用来控制代码生成的方式。这些选项通常由执行编译器的命令行设定。例如,使用微软编译器,生成一个翻译单元的典型命令行如下:

C :\> cl /c foo.cpp /Fo foo.obj /wall /od /zi

此命令行告诉编译器和链接器编译但不链接(/c)foo.cpp翻译单元,将结果输出到foo.obj对象文件(/Fo foo.obj),打开所有警告(/wall),关掉所有优化(/od)并产生调试信息(/zi)。

现代的编译器提供了大量选项,每次生成代码时都重新指定这些选项,这既不现实又易犯错,因此生成配置(build configuration)应运而生。生成配置是解决方案内个别项目的预处理器、编译器和链接器的选项集合。程序员可设置任意数量的生成配置,可任意命名,并在每个生成配置中设定不同的预处理器、编译器和链接器选项。默认把同一组选项应用到项目中的每个翻译单元,但也可以在个别翻译单元上做特殊设置,以替代项目的全局设置。(笔者建议,如非必要,避免使用此设置方式,因为很难分辨哪些.cpp有自定义设置,哪些没有。)

多数项目都至少有两个生成配置,通常名为“调试(Debug)”和“发布(Release)”。发布生成做最终软件出版之用,而调试生成则做开发之用。调试生成比发布生成运行得慢,但调试生成向程序员提供了宝贵的开发及调试信息。

**常用生成选项**：列举一些游戏引擎项目生成设置中最常见的选项

1. 预处理器设：C++预处理器处理#include文件的展开,以及处理#define宏(macro)的定义和替换。所有现代的C++预处理器皆有一个极强大的功能,就是可以通过命令行定义预处理宏(因而也能通过生成配置定义)。用这种方式定义宏,和在代码中编写#define 指令等效。多数编译器提供此功能的命令行选项为-D或/D,此选项可出现多次。

此功能让生成选项和代码沟通，而不需要修改代码本身。举一个常见的例子，在调试生成中必然会定义\_DEBUG 符号,而在发布生成中会定义NDEBUG符号进行替代。源代码可以检查这些符号,去“了解”目前是生成调试模式还是发布模式。这称为条件编译(conditional compilation)。

编译器也可以基于其编译环境和目标平台的信息，自由地将“魔法”预处理宏加入代码中。例如,当编译一个C++文件时,大多数编译器会定义\_\_cplusplus宏,从而能编写代码自动地适应C或C++编译。

又例如,每个编辑器都会通过一个“魔法”宏,让代码识别编译器。当用微软的编译器编译代码时,编译器会定义\_MSC\_vER宏;当使用GNU编译器(gcc)时,则会定义\_GNUC\_宏,其他编译器也如是。与此相似,执行代码的目标平台也是用宏来定义的。例如,生成32位Windows机器的执行代码时,就会定义\_WIN32符号。可以利用这些关键功能去编写跨平台代码,因为这些宏使代码“了解”目前被哪个编译器编译,并需要编译至哪个目标平台。

1. 编译器设置：控制编译器产生的对象文件是否包含调试信息(debugging information)是最常见的编译选项之一。调试器使用此信息去逐步执行代码、显示变量的值等。调试信息会增大磁盘上的可执行文件大小,也会方便黑客做反向工程。因此,最终发布的可执行文件必会去除这些调试信息。然而,在开发期间,调试信息是无价之宝,应该经常包含在生成的代码中。

另外,也可以控制编译器是否展开内联函数(inline function)。如关掉内联函数展开,每个内联函数在内存中只有一份,有唯一的内存地址。这样设置,使用调试器追踪代码时就容易得多,但其明显的代价是放弃了正常内联函数执行速度的提升。

内联函数展开是被称为优化(optimization)的泛代码转换例子之一。可以使用编译器选项控制编译器去尝试优化代码的进取性(aggressiveness),以及使用哪些优化方法。优化可能会打乱代码里的语句次序,完全去除一些变量,把变量移到不同地方,或在函数里将CPU寄存器(register)作为新用途重复使用。经优化的代码常会迷惑大多数调试器,令调试器以不同方式对用户“说谎”，并难以观察真实的执行情况。因此,在调试生成中,通常会关上优化选项。这样一来,每个变量、每行代码都会和原来编写的保持一致。但是,未经优化的代码,执行时较完全优化的代码会慢许多。

1. 链接器设置链接器也提供了多个选项,例如,控制输出文件的类型(如可执行文件或DLL)指定将哪些外部库链接至可执行文件,以及指定搜索哪些程序库的路径。惯例之一,调试时,可执行文件链接调试用的库,发布版本则链接优化的库。

链接器选项也可控制堆栈大小、程序载入内存时的首选基址、代码在哪些平台上执行(以做平台相关的优化),以及许多其他细节选项,不在此展开叙述。

**典型生成配置**：通常,游戏项目不止有两种配置。以下是游戏开发中的一些常见配置。

1. 调试(Debug):调试生成版本是非常慢的程序版本。此版本关闭了各种优化,禁用了所有函数内联,并且包含完整的调试信息。此生成版本用来测试新代码,以及调试在开发过程中出现的几乎所有最不平凡 (nontrivial)的问题。
2. 发布(Release):发布生成版本是较快的程序版本,但仍然保留调试信息并开启断言(assertion)。游戏能表现接近最终产品的运行速度,并留有机会去调试问题。
3. 制作(Production):制作生成配置是为生成最终发行给消费者的游戏版本而设置的。此配置有时也被称作“最终(Final)”或“光盘(Disk)”配置。制作生成配置与发布生成的差别在于前者去除了所有调试信息,通常关闭了所有断言,并完全启动优化。调试制作生成版本非常棘手,但制作生成版本是最快及最精干的生成类型。
4. 工具(Tools):有些游戏工作室的工具和游戏本身会共用代码库。在此方案中，加入“工具”生成配置很合理,用于为工具条件编译共用代码。工具生成配置一般会定义一个预处理宏(如TOOLS\_BUILD)，以告之代码当前是在生成工具用的版本。例如，某个工具可能需要一些C++类提供编辑用函数，而这些函数在游戏中并不需要,那么就可以用#ifdef TOOLS \_BUILD... #endif指令包围这些函数。由于工具通常也要分调试和发布版本,所以开发者会建立两个工具生成,如命名为“ToolsDebug”及“ToolsRelease”。

混合生成版本：混合生成版本(hybrid build)是指在其配置中,大部分翻译单元是发布模式，只有一小部分翻译单元为调试模式。使用这种配置,容易调试当前要监视的代码,而其余的代码能继续以全速运行。

基于文本的生成工具，如 make，能很容易地设置混合生成。用户能以翻译单元为单位把某些翻译单元设置为调试模式。大致做法是:定义一个make变量,如 $HYBRID\_SOURCES,列举所有要设置为调试模式的翻译单元(.cpp文件);设置生成规则,编译所有翻译单元的调试及发布两个版本,并将每个对象文件(.obj/.o)按其版本分别输出到两个文件夹;设定最终的链接规则,链接$HYBRID\_SOURCES列举的对象文件调试版本,以及其他对象文件的发布版本。若设置正确,make的依赖规则能处理余下的工作。

生成配置和可测试性：项目支持的生成配置越多,也就越难测试。虽然配置之间可能相差无几,但一些bug仍有可能只出现在某个配置中，而不出现在其他配置中。因此,每个配置都必须彻底测试。多数游戏工作室并不正式测试调试生成版本，因为调试配置主要用于开发新功能时,以及在其他配置遇到问题时做调试之用。然而,若测试人员花大部分时间测试发布配置,那么并不能在制作母片(gold master)2的前夜,直接制作出游戏的制作生成版本,并期望它的 bug状况与发布生成版本一模一样。实际上,在alpha到beta测试阶段,测试团队应同样彻底地测试发布及制作生成版本,保证制作生成版本不会暗藏任何烦扰人的意外情况。保持最少数量的生成配置,最有利于测试。事实上,有些工作室为此而不加入制作生成配置,彻底地测试发布生成后,就直接发行发布生成(但这去除了调试信息)。

**调试代码**：

1. 断点：断点(breakpoint)是代码调试的基本所需。每个断点都可以使得调试器在程序中的某行停下来,以便观察程序当时的运行状况。
2. 单步执行代码：当断点被命中时,可单步执行代码。
3. 调用堆栈：命中断点时(或手动暂停程序),双击调用堆栈窗口中的条目,就可在调用堆栈里上下移动。此操作非常有用,能检查从main()开始调用至目前代码行的一连串函数调用。例如,有时候在深层的嵌套函数调用中,此方法能往上追查父代函数,从而找出bug的源头。
4. 监视窗口：当单步执行代码并在调用堆栈里上下移动时，程序员需要检查程序中变量的值。监视窗(watch window)就是为此而设置的
5. 数据断点：常规的断点触发条件是CPU的程序计数器命中某个机器指令或代码行。然而,现在的调试器提供另一个极有用的功能,就是能够设立另一种断点,其触发条件是数据写入(即改变)某指定地址,所以这种断点称为数据断点(data breakpoint)。由于它是通过CPU的一个特殊功能而实现的,该功能可以在指定地址被写入时引发一个中断(interrupt)，所以这种断点又称为硬件断点(hardware breakpoint)。

以下介绍数据断点的典型用法。例如,在追查一个bug 时，发现某个对象的成员变量m\_angle 的值为零(0.of)，而此变量的值应该永不为零。程序员可能不知道哪个函数可能把零写入了此变量,但是程序员知道该变量的地址。要找出“肇事者”,可在变量的地址中设立数据断点,之后让程序继续执行。当该变量的值被改动时，调试器就会自动停下来。那时便可以检查调用堆栈,肇事的函数就能被捉个正着。

1. 条件断点：在“断点”窗口中,任何种类的断点(数据断点或常规的代码行断点)都可以设置条件(condition)和命中次数(hit count)。

条件断点(conditional breakpoint)使调试器在每次断点被命中时,都对输入的C/C++表达式进行评估。若表达式为真(true)，调试器会暂停程序,让用户检查程序运行情况。若表达式为假(false),调试器会忽略断点,继续运行程序。这个功能可以用来建立一种断点,当一个函数被某个实例调用时才触发。例如,假设一个游戏关卡中有20辆坦克,程序员希望第3辆坦克被调用时暂停程序。如果第3辆坦克的内存地址为0x12345678，便可以设置断点条件为“(uintptr\_t)this == 0x12345678”，以限制断点只命中第3辆坦克实体。

另外，设置命中次数之后,调试器每次命中时就会令该计数器减1,直至计数器变为零才触发暂停程序。这种设置对位于循环中的断点很有用。例如,要检查循环中第376个迭代的情况(例如数组中的第376个元素),不可能人工慢慢地去按F5键375次!但你可以设置命中次数,让 Visual Studio自动完成任务。

有一点要小心,如果设定了条件断点,那么每次命中时调试器都要对表达式取值,因而会减慢调试器和游戏的运行速度。

**调试已优化的生成**：之前提及,使用发布版本调试问题是非常棘手的,这主要缘于编译器优化代码的方式。在理想情况下,程序员都宁愿选用调试生成进行调试。然而,这经常不可行。有时候,一个bug重现率很低，当bug出现时就要好好把握调试机会，甚至有时候某个bug只出现在别人机器的发布生成中。另外,一些bug只出现在发布生成中,在调试生成中又会神奇地消失。这些只出现在发布生成的可怕bug,有时候是由未初始化变量造成的。因为在调试模式中变量和动态分配的内存通常会被设为零，而在发布模式中这些内存没有初始化,内容为随机值。bug只出现在发布生成版本的其他常见原因包括,在发布生成中意外地略去一些代码(例如,一些重要代码被错误地放进断言语句),数据结构的大小或数据成员打包方式在调试和发布模式间有差异,内联或编译器引进的优化触发bug, (罕见情况下)编译器的优化器本身的bug,也会导致在发布生成中产生错误的代码。

显然,每位程序员都必须有调试发布生成的能力,即使这看上去并不是一件轻松的事情。减轻调试优化代码之痛,最佳办法是多练习,并且在有机会时扩展这方面的技能。以下是一些窍门。

1. 学习在调试器中阅读及单步执行反汇编(disassembly):在发布生成中,调试器经常不能正确地显示目前正在执行的代码行。“感谢”指令乱序,在源代码检示模式中,经常会看见程序计数器不规律地在函数内游走。然而,若使用反汇编检示模式,则一切都变得正常(即能逐条执行汇编语言指令)。每位C/C+＋程序员都应该稍稍了解目标平台的架构和汇编语言。如能这样,即使调试器受迷惑了,程序员也不会被迷惑。
2. 运用寄存器去推理变量的值或地址:有时候,调试器不能在发布生成中正确显示变量的值或对象的内容。但是,如果程序计数器距离变量的初次使用不远,那么有很大机会该变量的值或地址仍然存于其中一个CPU寄存器里。若可以向前追踪反汇编,找到变量第一次载入寄存器的位置,就可以不断检查该寄存器以得知变量的值或地址。可使用寄存器窗口,或在监视窗口键入寄存器名字,检查寄存器内容。
3. 使用地址去检查变量及对象内容:知道变量或数据结构的地址,就可以在监视窗口中转换为适当的类别去检查其内容。例如,一个Foo类的实体位于0x1378AOCo,就可在监视窗口键入“(Foo\*)0x1378A0co”,那么调试器就会诠释该地址为指向一个Foo对象的指针。
4. 利用静态和全局变量:就算经过优化的生成版本,调试器通常也能够检查静态和全局变量。若不能推算出一个变量或对象的地址,则可以看看可能直接或间接地存有该地址的静态和全局变量。例如,若要找一个物理系统内部变量的地址,可能会发现它是存于PhysicsWorld全局变量中的一个成员变量。
5. 修改代码:若想相对简单地重现一个只出现在发布生成版本中的 bug,可考虑修改源代码以协助调试问题。增加打印语句去显示情况,引入全局变量使在调试器里检查问题变量或对象更容易,加入代码以检测问题状态,加入代码去孤立一个类的某个实例。