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

**1.1-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。