3D 游戏引擎架构综述

3D 游戏引擎架构综述

摘要

电子游戏被称为第九艺术，是集合了绘画音乐喜剧等等其它艺术形式的一种超媒体艺术形态，而游戏引擎于游戏就好比笔墨纸砚之于书法。本文针对 3D 游戏引擎的各方各面尝试做一个简单的介绍，试图予一关于游戏引擎的直观总体印象，并以 OGRE 为例子，剖析其中一个子模块专题，试图展开一个真实游戏引擎的神秘面纱，阐述内部源码设计实现。

关键词：游戏引擎；游戏；3D；OGRE

目录

第一章 前言

游戏引擎是方便游戏设计者能够快速规划和编程实现而不需要从底层一层层堆叠出一款游戏的一种工具[wikipedia]。通常来说，一个游戏引擎就是一个软件集成开发环境，提供了包括适用于 2D 或者 3D 的渲染引擎、物理引擎和碰撞检测、声音、脚本、动画、人工智能、网络、流媒体、内存管理、线程、本地化支持、场景图等核心功能，开发人员使用它们来为控制台，移动设备或者个人电脑创建游戏。

游戏引擎的种类有很多，譬如大多数游戏引擎的设计者会仔细设计和微调游戏引擎以让它们能够开发出某个特定类型的游戏（如 FPS，RPG）并且在某个特定平台（如 PC 端，移动端）上运行。另一方面，根据游戏引擎所能开发游戏的维度的不同，又能将游戏引擎分为 3D 游戏引擎和 2D 游戏引擎。相对于 2D 游戏引擎来说，3D 游戏引擎的世界坐标系多了一个维度，因此复杂度大大增加，开发难度也就不可相语。有趣的是，往往 3D 游戏引擎也能用于开发 2D 游戏，比较经典的一种开发方式是将任务做成垂直 xz 平面的二维纸片，同时将摄像机垂直对准 yz 平面，并根据 xy 轴坐标来表示它们之间的遮挡关系。在上面的基础上更进一步，倘若将垂直于 xz 平面的二维纸片改为向负 z 轴倾斜 45 度角，则能实现 2.5D 的效果。因为 2D 游戏引擎较为简单，本文将主要针对 3D 游戏引擎的设计原理和主要功能的设计方法做一个简要的综述，同时，本文以一个相对较于简单的开源 3D 游戏引擎——OGRE 作为例子，详细剖析其场景管理模块里面的渲染队列相关专题，旨在揭开游戏引擎的神秘面纱，提供一个大体上对游戏引擎内部设计思想和相关实现技术的直观印象。

本文的内容组织结构如下：

第一章阐述游戏引擎的概念，主要核心组件和简单分类，并引出和概述本文的论述内容和整体架构。

第二章先从一个宏观的角度具体介绍了游戏的概念和 3D 游戏引擎的需求架构，基本模块等，并阐述了游戏、游戏引擎和底层接口的关系，给予读者一个大图景。紧接着，针对一般 3D 游戏引擎均有的几个专题模块，本文以 OGRE 为例子进行详细的介绍和分析。

第三章从理论的介绍转入实例代码的剖析，本文以 OGRE 的场景管理模块里面的渲染队列相关的小模块代码为例子，剖析其代码，包括模块的整体分析（静态和动态

分析），各个类的描述（类功能的主要说明，类主要成员函数功能实现等），同时介绍采用的设计模式，企图将代码作者的设计意图和设计思想清晰地展示出来。

第二章 3D 游戏引擎设计综述

1.1. 游戏

游戏是一种基于物质需求已满足的条件下，旨在于一种特定的时间空间和特定的规则下追求精神需求满足的社会行为方法。

1.1.1. 游戏分类

游戏有很多种分类，如在第一章中所述，从适用平台这个角度来说，可以分为 PC 游戏、移动端游戏、平板游戏、体感游戏和游戏场游戏机等等，其中，PC 游戏主要使用键盘鼠标等控制，移动端则为触屏和陀螺仪等承当输入设备，对于体感游戏，较为出色的有微软的 Xbox360 等，主要采用体感设备监控玩家的骨骼动作，实现更好的沉浸式体验，而传统的游戏机，更多采用的是遥感和按键。

另一方面，从游戏风格上来说，又可以分为第一人称射击类游戏 FPS，角色扮演游戏 RPG，赛车游戏，即时战略游戏，大型多玩家在线游戏，解谜游戏等。像著名的 CS 即为 FPS 游戏，另外一些大型游戏如 GTA5、上古卷轴等为 RPG 游戏，即时战略游戏有 War3，星际争霸等等。

1.1.2. 游戏设计开发

对于一款完整的游戏来说，有几个要素是必须考虑的：主题、游戏受众、角色故事、玩法、场景风格、游戏平台等等，而游戏的成功则取决于这些元素各自的完整度设计成功度以及它们之间的完美配合。

在设计人员通过描述游戏内容及其卖点等完成游戏设计之后，游戏开发主要表现为开发人员利用现有的资源完成既定目标。一般来说，开发一款游戏和开发一款软件

的区别不大，主要流程为立项、策划游戏大纲、正式制作游戏、配音和配乐、检测和调试、广告和市场开发、生产和发售、售后服务。

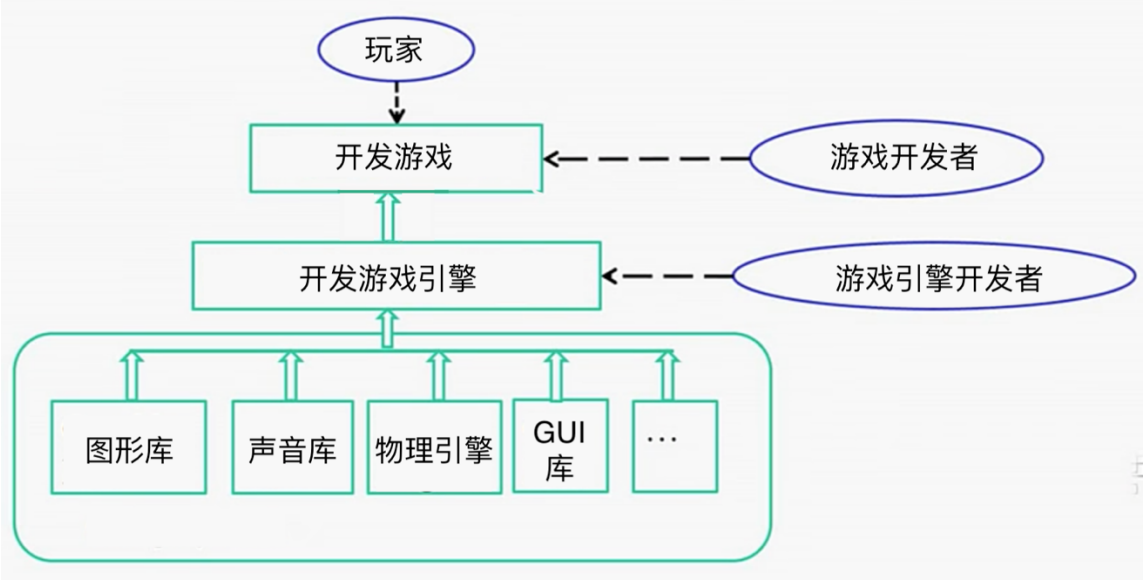
1.1.3. 游戏开发团队基本机构

主要包括工程师，艺术设计者，游戏设计者，生产商和出版厂商，其他工作人员可能包括工作室执行经理，市场部门，管理人员等等

1.2. 3D 游戏引擎概述

3D 游戏引擎的概念正如我们在前言所述，它通常不需要做较多修改就可以作为很多不同游戏的基础设施，并且可以根据游戏的不同做一定的扩展。然而，并没有一个游戏引擎能够适应所有的游戏，退一步讲，大多数的游戏引擎都只能针对某一或几特定类型的游戏。也就是说，游戏引擎也是有各种各样的分类的，即便如此，仍不妨碍我们从一个宏观的角度上来剖析一个游戏引擎基本应该提供的功能及其架构设计。

1.2.1. 玩家、游戏、游戏引擎和硬件的关系



正如图中所示，游戏对应游戏开发者，而游戏引擎对应的是游戏开发者，游戏引擎就相当于是把底层资源整合并且提供高层抽象接口的操作系统一样，供游戏开发者使用。而最顶层的玩家是游戏的用户。

1.2.2. 游戏引擎的需求分析

以下逐项分析一个游戏引擎的需求和应考虑的功能，尝试予一个游戏引擎的宏观图景。

1.2.2.1. 目标硬件、设备驱动程序、操作系统

- 游戏引擎需要考虑支持硬件的环境，如 PC、平板、手机或者游戏专用设备等
- 硬件资源都需要驱动程序，一般地，其接口设计是通用或标准的
- 操作系统 OS，最重要的系统软件——好的引擎是能跨平台的，比如 Unity 3D
  - Windows
  - Linux
  - iOS
  - Android
  - ……

1.2.2.2. 第三方软件开发包和中间件

一般来说，很少引擎是完全独立开发的，许多通用的基础软件包或中间件能够提高开发效率

- STL, Boost
- OpenGL, DirectX
- PhysX, ODE
- Havok Animation
- OpenAI
- OpenAL 等等

1.2.2.3. 支撑引擎运行的平台独立层

隔离了引擎与硬件、操作系统、第三方软件包，是引擎可以跨平台运行的基础

一般平台独立层可能包括 IO，网络传输层，高分辨率时钟，线程库，图形 API 包装，物理引擎包装等模块

1.2.2.4. 引擎的核心子系统

核心子系统层是引擎开发的必要功能，主要包括日志系统、调试及控制台、内存分配、数学库、性能分析、本地化服务、引擎配置、反射/序列化、单元测试等等模块

1.2.2.5. 引擎的资源管理

游戏引擎支持资源管理的功能某种程度决定了引擎的成败，Unreal Engine4 支持丰富的游戏资源，为游戏开发者提供了方便

一般来说，游戏的资源主要包括模型、纹理、字体、地形、动画、声音等等，其资源管理包括离线资源管理和运行时资源管理

1.2.2.6. 图形渲染、角色动画、声音

引擎的渲染是游戏画面处理的关键，通常会运用 GPU Shaer 进行并行处理。另一方面，图形渲染还需要场景剔除、遮挡判断等优化技术。关于渲染，游戏的低阶渲染器主要包括高阶 shader 管理，阴影光照，文本渲染，纹理及表面管理。

对于游戏中的动画，主要是为了描述游戏人物、动物的部分，有多种角色动画的表示方法。骨骼动画管理主要包括骨骼网格渲染，骨骼节点，动画插值及播放，反向动力学等内容。

声音则可以增强游戏的表现效果，有一些引擎使用第三方的音频库，有些引擎自带音频功能。引擎一般会提供音频播放接口，也可能会有 3d 音效等功能

1.2.2.7. 其他功能需求

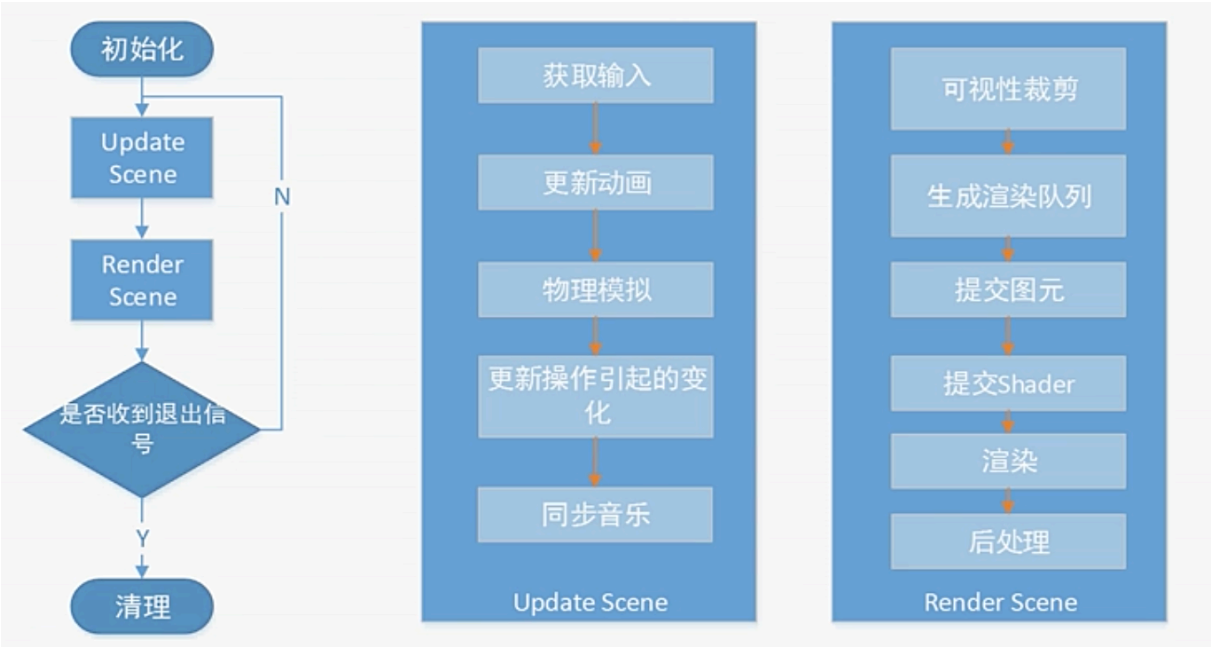
- 物理和碰撞检测
- 场景管理
- 人工智能
- 图形用户界面 GUI
- 脚本处理
- ……

1.2.3. 游戏引擎架构的设计原则

- 包含基本的游戏功能
- 要关注的核心功能的分离与各模块的结合，即高内聚、低耦合的原则
- 简洁，易于扩展
- 运用渐进式架构，不断地修改架构，分离出新的组件
- 性能好，运行稳定
- 可构建性强
- 模块化
- 可扩展性好

1.2.4. 游戏引擎设计——游戏引擎主循环

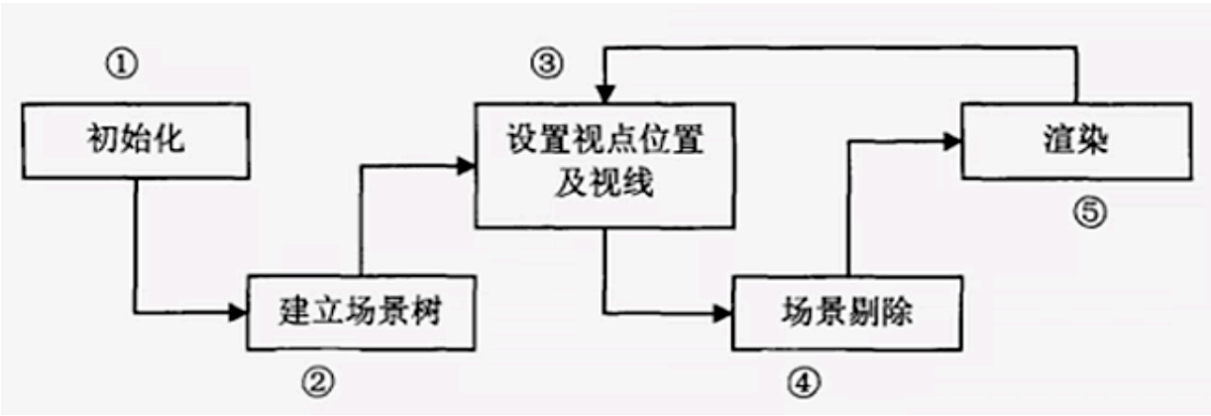
一般来说，游戏的主体都是一个循环，称为主循环。游戏即是不断地根据输入去更新画面给予玩家反馈刺激。其基本的架构图如下



1.3. 场景管理

场景管理主要是通过对三维场景中的各种数据（如几何数据、地形数据、渲染属性数据等）进行有效地组织，提高场景的绘制效率。

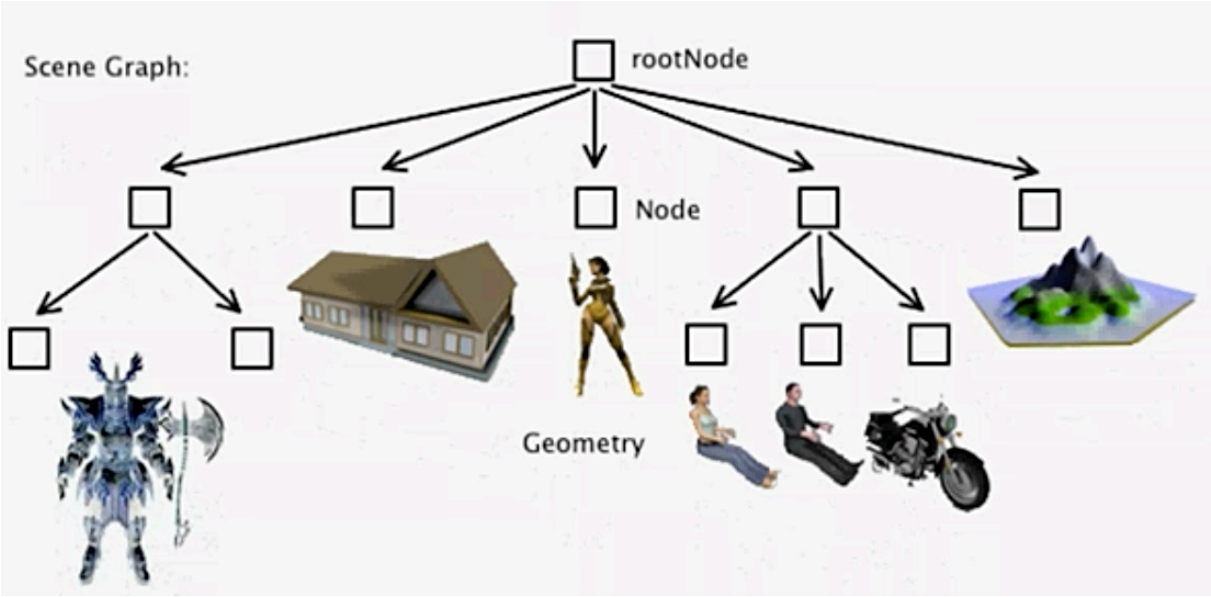
一般来说，其工作流程如下图所示：



1.3.1. 场景图

为了有效地组织和管理场景中的数据，游戏引擎通常采用场景图的方式。场景图时一种将场景中的各种数据以树或图的形式组织在一起的场景数据管理方式，它可以让提高场景管理的性能，增强有效性便携性和可扩展性。所以，对于一个引擎，场景管理就是对场景图的管理。

以下给出一个场景管理图：



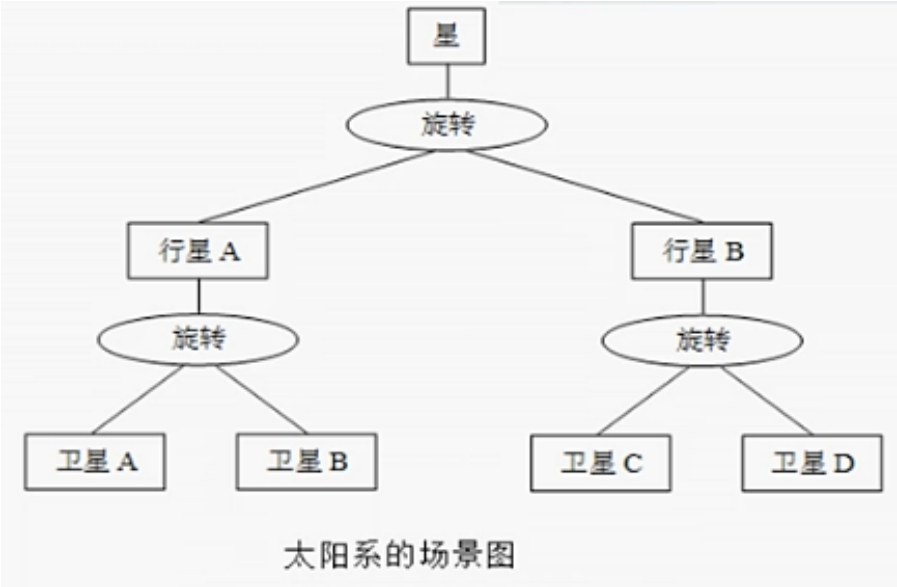
如上图所示，根结点即为该场景，其下的子树分别是此场景中的不同子场景，可能是一栋房子，一个机器人，或者一个假山。同样的递归下去，一个机器人场景下面又可能有手，身体，脚，头等子节点。

1.3.1.1. 场景图的节点

虽然不同引擎对于场景图的组织方式各不相同，但大体上来讲，场景图的节点主要分为：

- 几何节点：几何节点是各类物体的几何表示，它的成员函数主要完成绘制功能
- 变换节点：变换节点包含了场景中模型的图形变换操作，将模型的平移、旋转、缩放等合成为一个变换矩阵保存在节点中

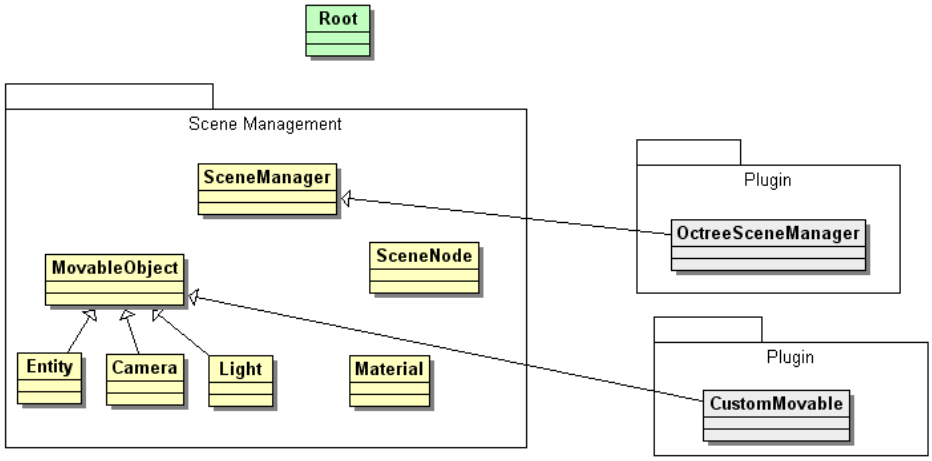
以下给出太阳系场景图可能的组织方式以提供一个直观的理解：



太阳系的场景图

1.3.1.2. 实例——OGRE 的场景管理

以 OGRE 为例，Scene 模块是核心模块 core 下面诸多模块之一，主要负责 OGRE 中场景以及场景内物体的组织和管理，其下面有 SceneManager 等主要类用于管理场景。该模块的类图如下所示：



可以看到，

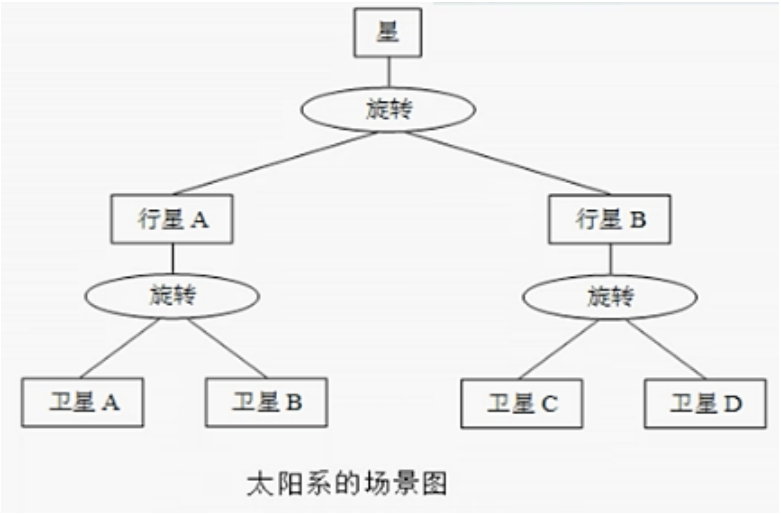
- SceneManager 管理模块中所有类，其下边为 SceneNode
- 由 SceneNode 构建树结构，下有 MovableObject
- MovableObject 为可移动的物体
  - Entity 构建树 （如人）
  - SubEntity (如头手脚）

上面节点一般均继承于 Node 父类，该虚类拥有增删改查子节点等一般树节点所拥有的通用接口，方便 OGRE 对于场景的管理。

1.3.2. 三维游戏场景的组织和管理

这个小节将叙述 3D 游戏场景的组织和管理以及其通常使用的相关技术，包括上面提及到的场景图，以及包围体技术，场景剖分方法等。

1.3.2.1. 基于场景图的场景管理



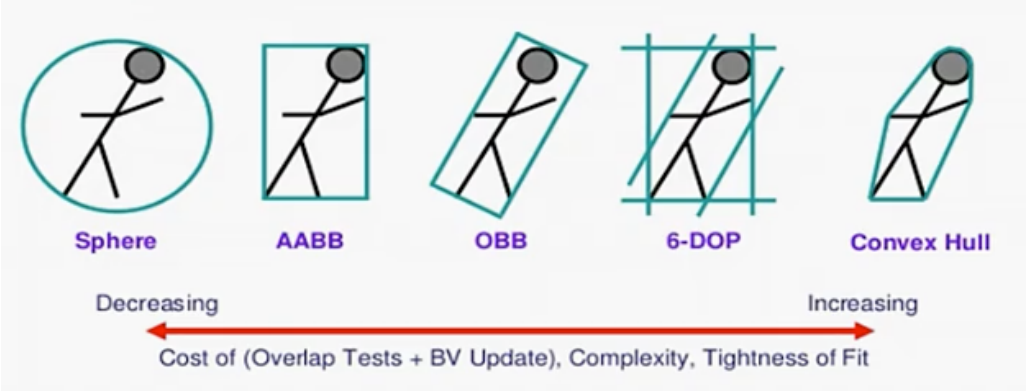
太阳系的场景图

如图为场景图的一个实例，基于场景图的场景绘制分为两步进行

1. 根据游戏的需要更新场景图必要的部分
2. 场景图的剔除和绘制过程

1.3.2.2. 三维场景组织——包围体技术

对于场景中的每个物体，为了加速判断场景物体之间的空间关系从而加快三维场景的渲染，我们通常采用一种称为包围体技术的方法对物体进行包围处理。顾名思义，即是采用简单的几何体对场景中的物体进行包围，常用的包围体技术有以下几种：

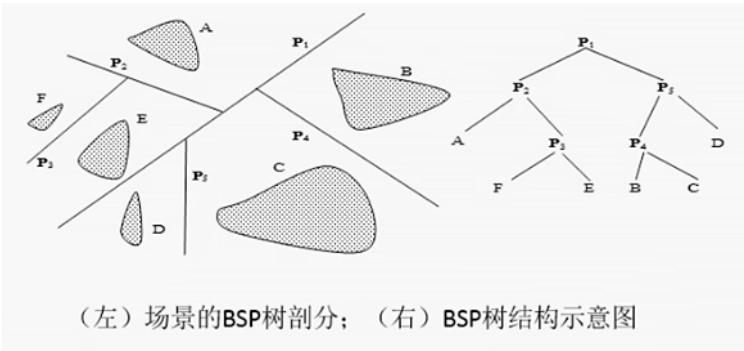


不难看出，从左至右包围技术越加复杂，效果越为精细，采用的算法自然也是复杂度递增，代价逐渐增加。可能使用到凸包算法，分治式增量式礼包式等方法，涉及到基本的计算机几何图形学算法，此处便不深究。

1.3.2.3. 三维场景剖分技术

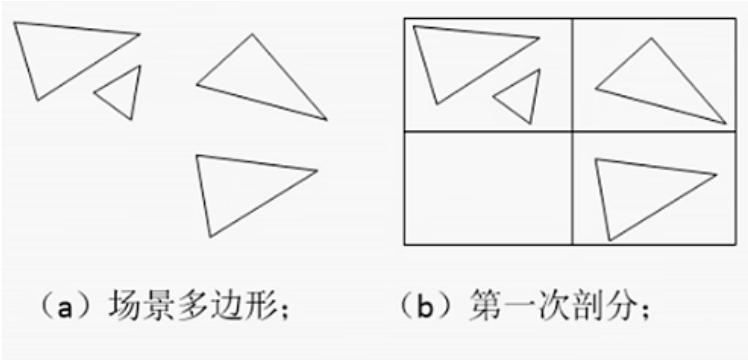
场景剖分技术主要讨论的是如何将场景剖分加以组织管理，常用的有二叉树（即BSP 树），四叉树和八叉树等，示意图如下：

- 二叉树



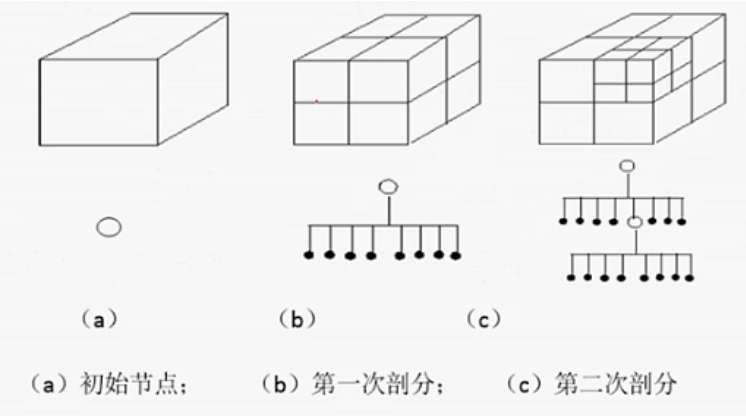
（左）场景的BSP树剖分；（右）BSP树结构示意图

- 四叉树



（a）场景多边形；　　（b）第一次剖分；

- 八叉树



（a）初始节点；　（b）第一次剖分；　（c）第二次剖分

以下对这三种技术做一个对比：

技术名称	使用场景	构建复杂度	实用性
二叉树	尺寸不大的室内建筑场景	复杂	大部分三维游戏引擎
四叉树	室外基于高度场的地形	一般	仅用于地形绘制
八叉树	大规模室内外空间场景	一般	复杂游戏引擎

这里仅仅对组织方式做一个简单的介绍，以下会详细介绍其中最为常用的一种，即 BSP 二叉树。

1.3.3. BSP 树

BSP 树的全称是 Binary Space Partitioning，即二叉空间分割，一种用于场景剖分的技术。

1.4. 实时渲染技术

实时渲染就是指场景的一个视图在显示窗口中的实现，这个场景的视图可能是由镜头（视点）的改变或玩家的操作事件出发的。场景管理的主要目的就是实时渲染，即快速生成真实效果的场景视图。在游戏引擎中，这通常是最接近图形处理软件包的一层（directorX，OpenGL 等）。

实时渲染是对场景的所有网格面进行像素化着色计算，产生渲染帧，对于一个游戏而言，这就要求：

1. 渲染帧生成的速度要快——剔除不可见场景
2. 渲染帧生成的效果要真实——真实感处理

通常，为了达到这些计算密集型的任务，游戏引擎会使用 GPU Shader 来进行编程。

总的来说，实时渲染的主要任务有：

- 当镜头改变 或 可移动对象变化
- 遍历场景树，渲染场景
  - 剔除不可见场景：用包围盒、LOD 等技术
  - 真实感绘制：光照、纹理、阴影、特效等混合
- 生成场景视图（FrameBuffer）的渲染结果

1.4.1. 镜头确定

一个镜头（相机）对应一个图形渲染的现实串口，同时，它也是场景中的一个可移动节点。在场景图中，镜头可以相对世界坐标定义，也可以相对一个局部坐标定义。而场景与镜头的关系则是为人熟知的第一人称和第三人称。

由上一节的讨论可以知道，在场景渲染中，每当镜头改变的时候，场景就需要被更新渲染，即需要监听镜头事件。另一方面，根据镜头与物体的距离，又可以确定物体之间的遮挡关系，即在渲染的时候，哪些物体是应当被剔除的。

1.4.2. 不可见场景的剔除

为了提高渲染效率，通常引擎内部会对不可见物体进行剔除操作，减少渲染物体个数，提高渲染效率。

一般来说，剔除物体的物体有以下几个分类：

- 根据镜头为止，剔除视见体外的物体
- 用包围盒判断视见体中物体的遮挡关系，剔除被遮挡的物体
- 剔除可见物体的背面
- 根据镜头与视见体中的可移动物体的位置，确定物体的现实细节（如网格，纹理等）

场景物体的剔除分为：

- 物空间（矢量空间）：剔除不可见的物体，或被遮挡的面

- 算法包括基于包围盒判断的裁剪算法，画家算法，射线追踪算法等

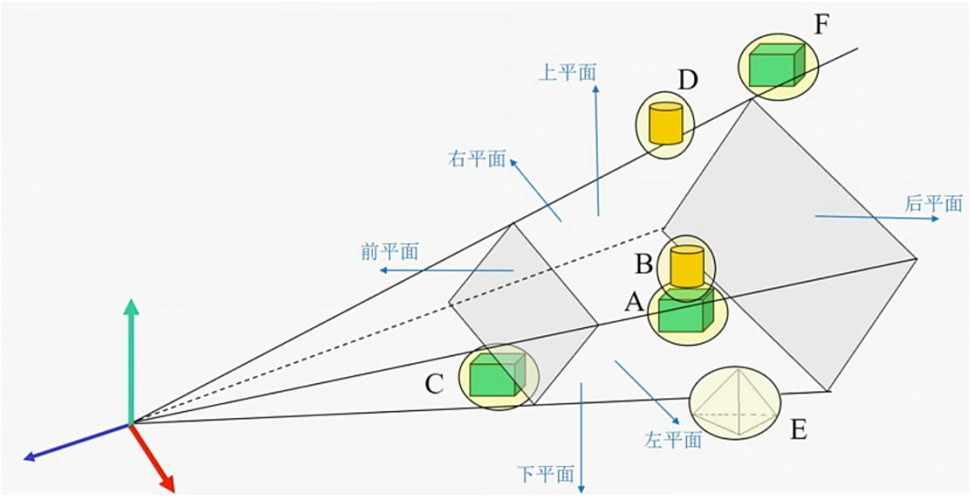
- 像空间（像素空间）：剔除被遮挡的像素点
  - 包括深度(z_Buffer)缓存算法等

另外细节层次(LOD)方法也是减少面渲染计算工作量的一种方法：

- 网格层次：减少网格面的个数
- 纹理层次：减少纹理图的大小

1.4.2.1. 视见体外的物体剔除

视见体外的物体是指在视见体包围体之外的物体，将其剔除主要采用的是裁剪算法。一般用视见体的六个面分别裁剪物体的包围体，在视见体外的物体即直接剔除。



如图所示，包围盒视见体的裁剪中，为剔除的有 A, B, C, D, F（F 在由平面的左侧），被剔除的有 E（E 在左视见体面的左侧）

1.4.2.2. 视见体内物体的遮挡剔除

主要有以下几种方法：

- 基于包围体判断物体的前后关系。例如大规模场景物体的剔除
- 用射线追踪法判断物体平面与视点距离的远近，剔除被遮挡物体的三角形面片。例如地形网格面的剔除
- 像空间的深度缓存判断三角形面片上的被渲染像素的前后关系

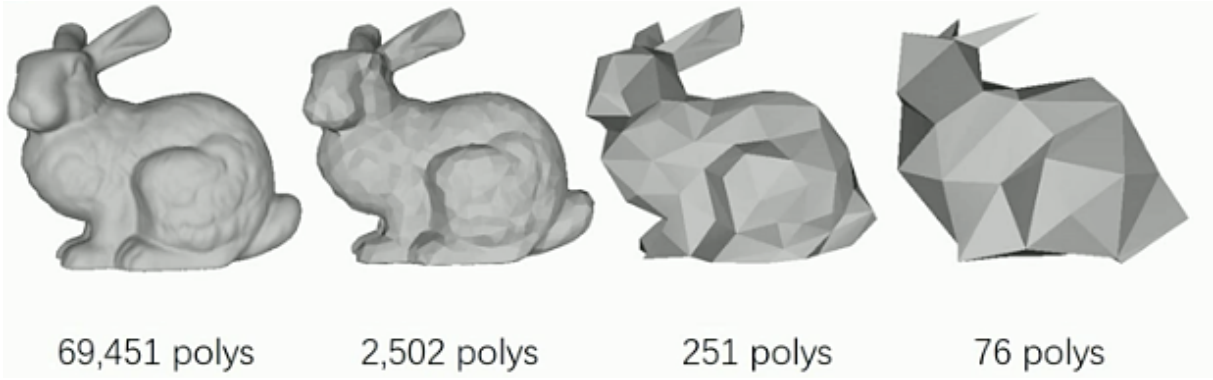
1.4.2.3. 背面剔除

物体上的每个三角形面片都有正面和背面，由面的法向量决定。若三角形面片的背面向着镜头，则剔除（减少面的渲染计算）。一般来说，基础图形库中都有背面剔除开关，

1.4.2.4. LOD

LOD 即 Level Of Detail 的意思，意指在渲染的时候，对于一些比较小比较远或者比较不重要的场景部分，使用一个更少细节的表示方法。一般 LOD 可以用于模型网格的细节分层，也可以用于纹理贴图的细节分层。

以下是 LOD 的一个经典的例子图片，即使用数量不同的三角形面片去渲染一只兔子：



可见，倘若该模型距离视点比较远，那么靠右的方案也是可以接受的，这就大大地降低了渲染成本。

1.4.3. 真实感场景绘制

真实感绘制主要是为了提高真实效果，其主要通过光照、纹理、阴影、雾化、视觉特效等内容的融合得到渲染结果

- 根据场景定义的灯光，用图形系统支持光照模型进行场景绘制计算
  - 局部光照模型：环境光、漫反射光、镜面放射光
  - 全局光照模型：光线追踪、辐射度
  - 光照绘制——由光源、三角形面片的定点法向量和材质属性等决定

- 光照绘制算法由 flat, gouraud, phone 等
- 灯光可移动，需绘制

- 纹理一般作为引擎的资源，与网格一起定义，并映射到网格面上
  - 纹理是真实感渲染的重要内容，由像素表示，以贴图形式和网格模型关联，是一种资源文件
  - 包括多重纹理、凹凸纹理、光泽纹理、投影纹理、环境纹理、球面纹理、立方纹理等
- 阴影可以是贴图或直接计算的形式实现
  - 阴影是从视点能看到的从光源看不到的东西，分为纹影区和半影区，主要由光源类型决定
- 雾化可以产生场景深度效果
- 视觉特效是由布告板或粒子系统等产生的光影效果
  - 指动态表现的光影效果

1.5. 游戏角色动画控制

游戏角色推动游戏进行的载体，包括有玩家与游戏进行交互的桥梁，实际上游戏角色带给我们游戏的各种视觉感光，通过游戏角色造型，设计，变化，运动，感受到游戏的乐趣。主要可以分为：主角、NPC（即 Non-Player Character）和敌人。

对于控制游戏角色采用的动画，目前或曾经的主流有：

- 赛璐璐动画（cel animation) 简单粗暴，难应用于 3D
- 刚性层阶式动画（rigid hierarchical animation)，各部分由刚体组成，不自然
- 每顶点动画（per-vertex animation）最佳但运算量大
- 变形目标动画（morph target animation）适合面部表情动画效果实现，不适合角色躯干动画
- 骨骼蒙皮动画（skinned animation）当前主流，绑定骨骼

以下主要介绍骨骼蒙皮动画

1.5.1. 骨骼蒙皮动画

骨骼蒙皮动画即使将皮肉绑定都骨骼上，通过骨骼动画来控制游戏角色的动作，其实现步骤如下：

1. 确定人体骨骼(kinet 等设备）
2. 关节从模型坐标到世界坐标以及其运动（DOF 等技术）
3. 蒙皮绑定到骨骼：蒙皮有自己的模型坐标轴，在绑定的时候将蒙皮从自己的模型坐标轴连接到关节的坐标轴上，做一个顶点与关节点的绑定

1.5.1.1. DOF 技术

DOF 是 Degree Of Free 的意思，即描述关节的自由度，在骨骼控制技术中主要有以下几种关节：

- 自由关节（Free Joint）是一种特殊的关节，可以围绕三个坐标轴旋转，并在三个坐标轴方向平移，所以它有六个自由度
  - 只有根关节（root joint）才可以是自由关节，其他所有关节都不可以设置为自由关节
  - 自由关节是唯一可以围绕坐标轴旋转和又可以平移的关节，其它的关节只能旋转，不可以平移
- 球关节（Ball Joint）可以围绕三个坐标轴旋转，具有三个自由度
- 普通关节（Universal Joint）只能围绕其中两个坐标轴旋转，具有两个自由度
- 铰链关节（Hinge Joint），它仅能围绕一个坐标轴旋转，只有一个自由度

1.5.1.2. 角色动画的自动合成方法

为了尽可能减少手工编写角色动作的工作量，人们设计出了种种可以从现有角色动画中自动生成新的角色动画的方法，这就是角色动画的自动合成方法。游戏引擎可以

采用线性混合，骨骼分部混合，加法混合，运动捕捉（MoC, Motion Capture）等方法。

A. 线性混合

关节插值后的姿势：将两个源姿态的每个关节的局部姿态线性插值(Linear interpolation, LERP)

整个骨骼的姿态：整个骨骼插值后的姿态是所有关节插值后姿态的集合。

B. 骨骼分部混合

Partial-skeleton Blending

即按照一个百分比将不同骨骼动画加权混合。

C. 加法混合

设源片段为 S，参考片段为 R，则加法片段 D=S-R，即 S=D+R

当有一个加法片段 D 之后，只需要把某百分比的 D 加进参考片段 R 中，则可以产生介乎 R 和 S 之间的动画。

D. 动作捕捉（Moc, Motion Capture)

makes use of live action

利用设备感应人体动画，采集人体运动到电脑，即运动序列。主要是使用一个真实的人或动物生成的动作。

1.6. 游戏引擎的交互控制、脚本语言和内存管理

1.6.1. 交互控制

游戏引擎对交互控制主要采取“事件“的方式，事件是指游戏状态或游戏环境状态的有趣改变，是游戏过程中发生或希望发生的用户玩家所关注的事情。比如玩家按下

手柄或键盘上的按钮、游戏中发生爆炸、敌方角色发现玩家等，这些事件控制游戏的进展和走向，而游戏引擎正是通过不断处理发生的事件，驱动着游戏的进行

1.6.1.1. 游戏引擎的时间处理

游戏一般有不同的游戏循环架构风格(windows 的视窗消息泵，回调驱动框架，基于事件的更新等等)，在不同的风格里面有不同的监听事件的方法，当循环中游戏监听到特定事件发生的时候，就会针对该事件执行特定的处理函数，这就是游戏引擎的事件处理

另外，多数游戏引擎都有一个事件处理机制。

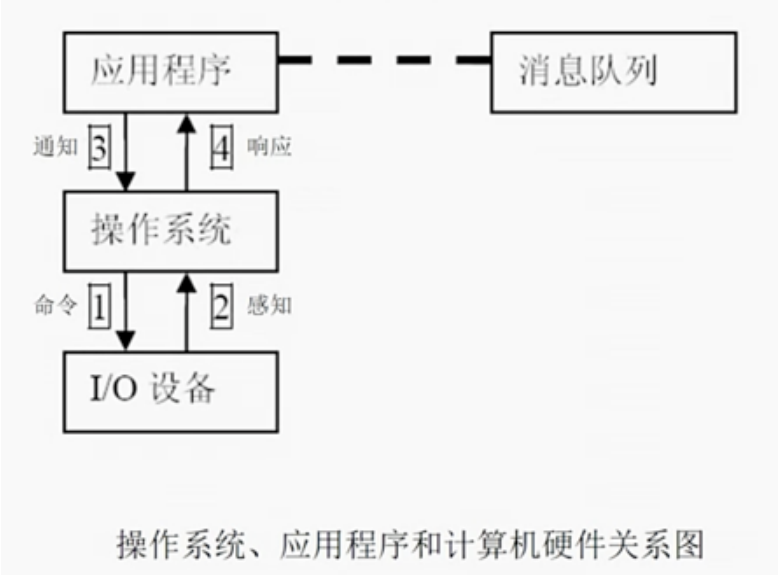
引擎系统有一个事件队列，保存并登记已定义的各类事件，当某一事件发生的时候，就调用处理程序执行该事件。这有个好处就是可以定义不同事件之间的优先级和处理顺序。

游戏引擎的事件处理机制通常和图形用户界面里的事件或消息处理机制类似，比如微软的 windows 视窗消息处理机制、Java AWT 的事件处理机制、c#的 delegate 和 event 关键字处理机制。

1.6.1.2. 基于事件的交互式控制方法

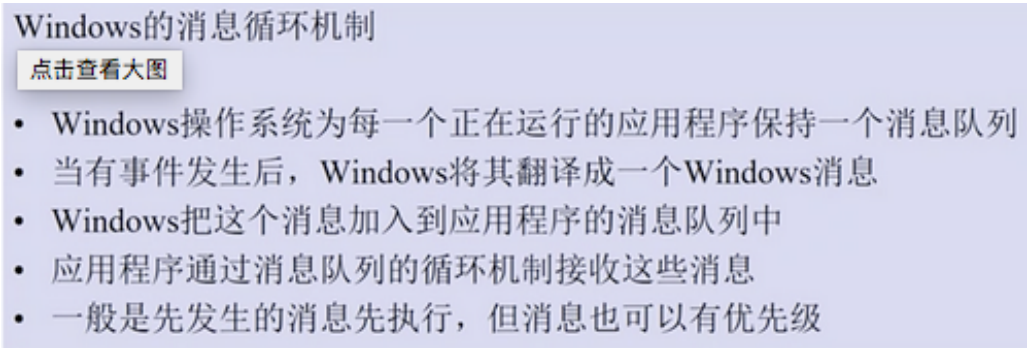
一般来说，游戏事件主要分为外部事件（外部状态的改变比如键盘按下，鼠标移动点击等）和内部事件（内部状态的改变比如发生爆炸，玩家被敌人发现等）

A. 外部事件

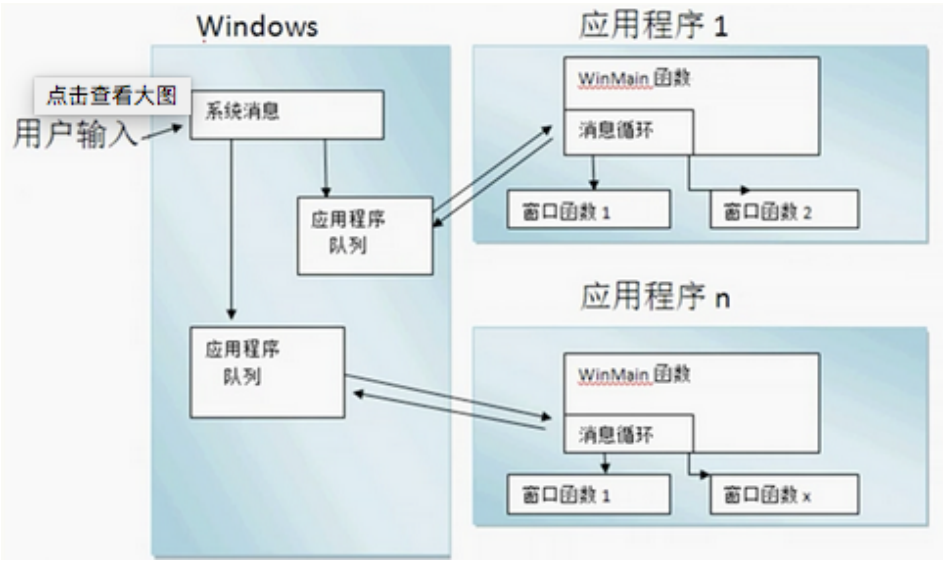


操作系统、应用程序和计算机硬件关系图

以下为 windows 的例子



不同消息丢到不同的 APP



主要代码如下：

```
1. MSG msg
2. while(getMessage(&msg, NULL, 0, 0)){
3. TranslateMessage(&msg);
4. DispatchMessage(&msg);
5. }
6. return msg.wParam;
7.
8. //用 switch 根据每个消息 id 处理
9.
10.  switch(uMsgId)
11.  {
12.  case WM_TIME:
13.  foo();
14.  return 0;
15.  ...
16.  default:
17.  return DefWindowProc(hwnd, uMsgId, wParam, lParam);}
```

B. 内部事件

内部事件主要考虑如何处理响应，比如发生爆炸如何响应，方法就是把事件封装成对象，包含事件的类型，参数，处理器，是否排队等要素。

1.6.2. 游戏脚本语言

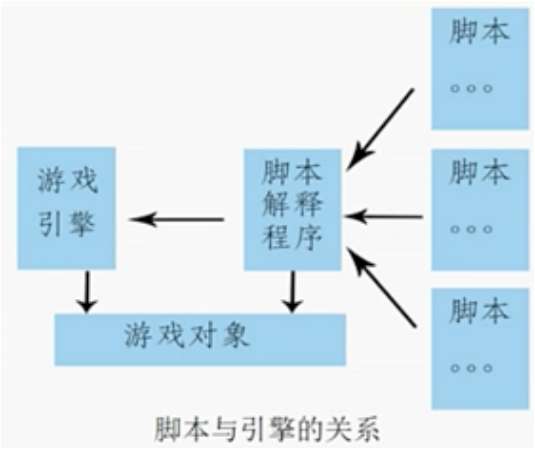
在游戏引擎的语境中，脚本语言是高级的、相对容易使用的编程语言，主要目的是让游戏变成人员方便地控制和定制应用软件程序的行为和逻辑，为游戏编程人员提供引擎 API 的功能。因此，脚本语言可以让程序员开发新游戏，或者是定制（"mod"）一个已有的游戏。

脚本往往具有直译性、轻便、支持快速迭代、方便易用等特性。

脚本拥有以下作用：

- 脚本语言可以为引擎的使用建立一个高端的接口，即可以用控制语言的方式调用引擎中的函数
- 脚本语言既可以用逻辑关系使用引擎，也可以简化游戏的程序设计，不需要像宿主语言一样复杂
- 脚本语言的解释执行更有效率，可以跨平台

脚本与引擎的关系图阐述：



1.6.3. 内存优化技术

内存管理是引擎底层和核心内容之一，决定了系统的运行效率。

- 通常来讲，游戏引擎的内存管理是指:
  - 各种结构的对象数据的内存分配与释放(队列、map、树、对象工厂、容器等)
  - 数据访问(查找、get、set 等)
- 引擎中需要管理内存的部分(CPU 访问的内存，主内存) :
  - 资源管理(模型几何信息、纹理信息、骨架信息、特效信息等)
  - 场景图(天空、地形、对象的变换信息等)
  - 配置信息、状态信息、游戏规则信息、事件信息、声音。

显卡内存管理，有 1GB 以上的内存空间，从模型顶点变换到像素颜色计算都可以在 GPU 和显卡内存上完成

1.6.3.1. 管理 GPU 内存以加速——以 OGRE 为例子

以下简单介绍 OGRE 的 GPU 内存管理：

- 常规的内存管理方法只是针对主内存的管理，即 CPU 访问的内存
- 此外还有其他硬件中的内存，显卡，声卡等
- 显卡的内存很大，包括 vertex buffers,index buffers,texture memory or framebuffer memory 等
- 引擎系统在渲染时要用 Shader 程序，同时要访问 GPU 内存

- CPU 内存与 GPU 内存要交换数据，渲染时交换的次数越少，渲染效率越高
- 一种方法“shadow" copy, 即 CPU 留一份 GPU 的影子,同时做修改
- HardwareBuffer 是抽象接口类，派生 vertex buffers,index buffers,counterbuffers,Uniform buffers,pixel buffers 等非主存的各种缓存子类
- HardwareBufferManager 类中的多个 createVertexBuffer()方法创建对应的缓存，其方法的返回值为缓存的共享指针类型(SharedPtr)，即不必担心内存泄漏
- 如果 GPU 数据要传回给 CPU,则设置 useShadowBuffer 参数为 True,即 CPU 留一份 GPU 的影子缓存，两边同时做修改

第三章 OGRE 场景管理渲染队列子模块分析

1.1. 引擎名称版本和分析模块

Ogre-1.10.11 版本（Object-Oriented Graphics Rendering Engine）

模块分析：Scene 模块中的 RenderQueue 相关模块

1.2. 模块整体分析

1.2.1. 模块概述

1.2.1.1. 涉及主要类

- SceneManager
- RenderQueue
- RenderQueueListener
- RenderQueueGroup
- RenderPriorityGroup
- QueuedRenderableCollection
- QueueRenderableVisitor
- RenderQueueInvocation
- RenderQueueInvocationSequence
- Pass
- Renderable
- RenderablePass

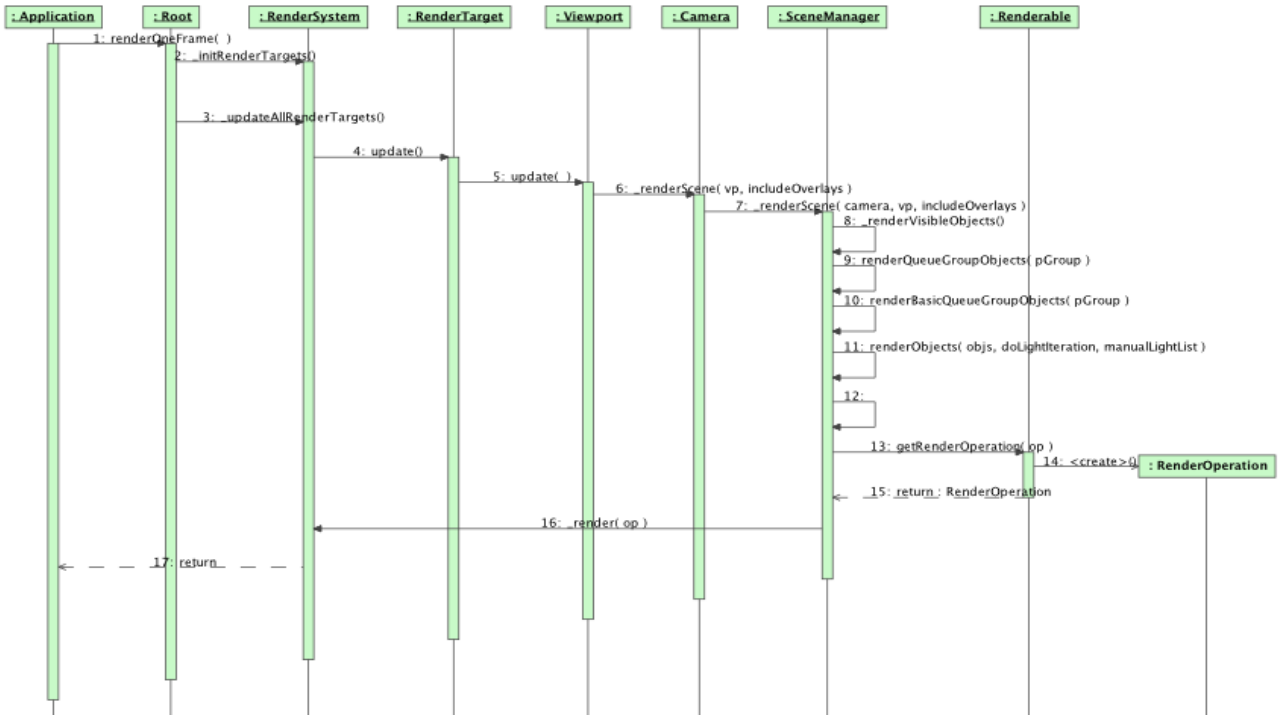
1.2.1.2. 概念综述

该模块主要涉及 ogre 中 sceneManager 是如何管理、组织和处理可渲染物体，同时采用了诸多的设计模式比如访问者模式、监听者模式等，通过不同层级的分组、排队、排序、队列组织形态等不同的方式对可渲染物体进行管理，其具体方式如下：

- 在 Ogre 中，采用一个渲染队列 RenderQueue 来对待渲染 objects 进行排队
- 因为先后渲染顺序会影响到 objects 最终在屏幕上的效果（比如遮挡关系），Ogre 先将待渲染物体分成不同的组，即 RenderQueueGroup。相近概念的物体属于同一个 group，比如背景是 groupA，世界片段是 groupB，物体是 groupC，那么明显渲染顺序为 groupA 先于 groupB 先于 groupC。通过将物体收纳到不同的 group，定义了其大体上的渲染顺序。
- 对于同个 group 里面的 objects，其渲染也要有一定的顺序，ogre 用 priority 的概念来定义：即，用一个值的大小来规定渲染顺序，这是由 RenderPriorityGroup 来决定的。
- 最后，Ogre 用 QueuedRenderableCollection 来作为存放基本物体（即 Renderable, Pass, RenderablePass）的最终场所，以 vector 的形式储存
- Ogre 在对物体进行渲染调用的时候，采用的是访问者模式，即会有一系列 QueueRenderableVisitor 类，每个类都会有访问三种基本物体的 visit 函数（如上图所示），基本物体则会有 acceptVisitor 的方法，当某个物体 accept 了一个 visitor 之后，会自动调用该 visitor 去访问这个物体。
- 以上所述类基本均被包含与被包含的关系，以及上层类对下层类成员函数的调用比如排序。

1.2.2. 模块动态时序调用分析

1.2.2.1. Ogre 总体时序图

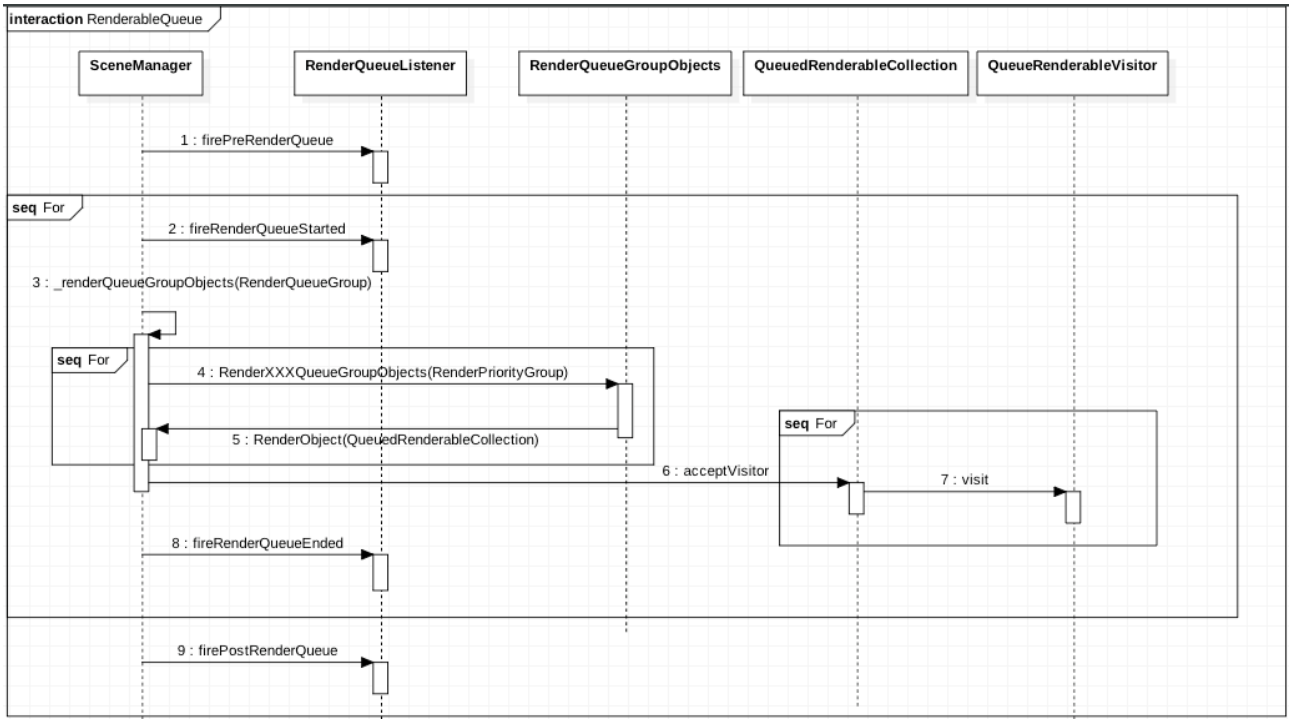


1.2.2.2. 从 SceneManager 的 renderVisibleObject 讲起

在 SceneManager 里面，根据渲染队列 RenderQueue 是否为自定义分为两个函数，分别为

- `void renderVisibleObjectsDefaultSequence(void);`
- `void renderVisibleObjectsCustomSequence(RenderQueueInvocationSequence* s);`

下面以第一个函数为例进入调用历程。



C. renderVisibleObjectsDefaultSequence

a) 说明

在 Ogre 中，SceneManager 渲染的时候采用监听者模式，不同的 RenderQueueListener 可以在 SceneManager 中注册，每当 SceneManager 开始渲染的时候便会自动通知 Listener

在上图中，整个 renderQueue 开始渲染之前 SceneManager 会先调用 preRenderQueue 通知各个 Listener，在结束的时候同样会做 postRenderQueue。 而 2,3,4 三个调用其实是在一个 for 循环里面，SceneManager 通过不断调用 _renderQueueGroupObjects 对 renderQueue 里面的每个 QueueGroup 进行渲染

D. _renderQueueGroupObjects

a) 说明

此方法使用 if&else 根据 RenderQueueGroup 里面各项参数对其进行渲染调用，具体选项有是否 doShadows，用用的 mShadowTechnique 等等，根据选项组合的不同调用不同的 renderObject 函数，其中总共有以下几个函数

- renderAdditiveStencilShadowedQueueGroupObjects(pGroup, om)
- renderModulativeStencilShadowedQueueGroupObjects(pGroup, om)
- renderTextureShadowCasterQueueGroupObjects(pGroup, om)
- renderAdditiveTextureShadowedQueueGroupObjects(pGroup, om)
- renderModulativeTextureShadowedQueueGroupObjects(pGroup, om)
- renderBasicQueueGroupObjects(pGroup, om)

以下以最基本的 renderBasicQueueGroupObjects 为例

E. renderBasicQueueGroupObjects

此方法对 RenderQueueGroup 里面不同 priority 组别的不同类型进行渲染，以 while 对其迭代，简单代码如下

```
RenderQueueGroup::PriorityMapIterator groupIt = pGroup->getIterator();
    while (groupIt.hasMoreElements())
    {
        RenderPriorityGroup* pPriorityGrp = groupIt.getNext();
        // 先排序队列
        pPriorityGrp->sort(mCameraInProgress);
        // 分别对队列里面的solids，unsorted transparents和descending的transparents进行渲染
       //其实就是三种类型的QueuedRenderableCollection
        renderObjects(pPriorityGrp->getSolidsBasic(), om, true, true);
        renderObjects(pPriorityGrp->getTransparentsUnsorted(), om, true, true);
        renderObjects(pPriorityGrp->getTransparents(),
           QueuedRenderableCollection::OM_SORT_DESCENDING, true, true);
}// 对于每个 priority 执行相同的操作
```

F. renderObjects

此方法调用 Visitor 对 renderable collection 里面的基本元素进行访问，ogre 采用访问者模式，对 renderable 最终的调用是通过相对应的 visitor 去访问参数中的 object，代码如下

```
void SceneManager::renderObjects(const QueuedRenderableCollection& objs,
                                 QueuedRenderableCollection::OrganisationMode om,
                                 bool lightScissoringClipping,
                                 bool doLightIteration,
                                 const LightList* manualLightList)
{
  //首先是对visitor各个参数的赋值
   mActiveQueuedRenderableVisitor->autoLights = doLightIteration;
   mActiveQueuedRenderableVisitor->manualLightList = manualLightList;
   mActiveQueuedRenderableVisitor->transparentShadowCastersMode = false;
   mActiveQueuedRenderableVisitor->scissoring = lightScissoringClipping;
   // 然后调用visitor进行访问
   objs.acceptVisitor(mActiveQueuedRenderableVisitor, om);
}
```

G. acceptVisitor 系列类

在 QueuedRenderableCollection 中 acceptVisitor 通过调用 visitor 的 visit 函数，对 collection 中的 renderable 和 pass 进行最终的访问，根据访问的不同分为以下三个函数：

- `void acceptVisitorGrouped(QueuedRenderableVisitor* visitor) const;`
- `void acceptVisitorDescending(QueuedRenderableVisitor* visitor) const;`
- `void acceptVisitorAscending(QueuedRenderableVisitor* visitor) const;`

对于每个 acceptVisitor 函数，QueuedRenderableCollection 按照访问者模式的标准，通过迭代器循环不断调用 visitor 的 visit 函数对基本元素进行访问渲染。

1.3. 各类解析

1.3.1. Pass & Renderable

A. Pass

Pass 类是一次 render 调用，具体可以分为以下两种：

- Fixed-Function Pass
  - 为默认的方式，即不使用 shader 编程
  - 快速方便有效
- Programmable Pass
  - 可以选用 vertex shader 或者 fragment shader program
  - 提供更多 flexibility，可利用 material 和 light 创造各种特效

B. Renderable

Renderable 是一个虚类，被各种可渲染的物体所继承实现

C. RenderablePass

RenderablePass 其实就是一个有 Pass 成员和 Renderable 成员以及相应构造函数的结构体，主要用于阐明 Pass 和 Renderable 的对应关系

1.3.2. RenderQueue

- SenceManager 中有一个记录待渲染物体的的队列
- 用于管理场景物体的渲染队列，有 addRenderable 等接口用于删减
- 另有变量 typedef map< uint8, RenderQueueGroup* >::type RenderQueueGroupMap 变量用于获取待渲染对象的优先级，map 的 key 的 unit8 是一个枚举类型，命名为 RenderQueueGroupID，即在上文所说的某个 RenderQueueGroup 的 ID。该类同时也提供接口供设置渲染优先级。
- 该类同时拥有 RenderQueueGroup 的迭代器用于对下个层次 RenderQueueGroup 的迭代。
- 类公有接口如下：

```
//清除队列内容

void clear(bool destroyPassMaps = false);
```

```
        //根据RenderQueueGroupID的到相对应的RenderQueueGroup
          RenderQueueGroup* getQueueGroup(uint8 qid);
        //根据groupID和Priority添加一个renderable
          void addRenderable(Renderable* pRend, uint8 groupID, ushort priority);
        //重载函数，只添加并规定groupId
          void addRenderable(Renderable* pRend, uint8 groupId);
        //重载函数，只添加
          void addRenderable(Renderable* pRend);
          //得到默认的groupId，即在改queue中如果没规定renderable的groupId就用这个groupId
          uint8 getDefaultQueueGroup(void) const;
        //设置默认优先级
          void setDefaultRenderablePriority(ushort priority);
        //得到默认优先级
          ushort getDefaultRenderablePriority(void) const;
        //设置默认groupId
          void setDefaultQueueGroup(uint8 grp);

          //得到迭代器，用于遍历该队列中的元素
          QueueGroupIterator _getQueueGroupIterator(void);
          ConstQueueGroupIterator _getQueueGroupIterator(void) const;

          //系列setter和getter规定
        //是否队列会将pass按照lighttype or NoShadow or shadowCasterCannotBeReceivers分
组
          void setSplitPassesByLightingType(bool split);
          bool getSplitPassesByLightingType(void) const;
          void setSplitNoShadowPasses(bool split);
          bool getSplitNoShadowPasses(void) const;
          void setShadowCastersCannotBeReceivers(bool ind);
          bool getShadowCastersCannotBeReceivers(void) const;
```

```
//监听器的getter和setter函数
  void setRenderableListener(RenderableListener* listener);
  RenderableListener* getRenderableListener(void) const;
  //和另外一个队列合并
  void merge( const RenderQueue* rhs );
  //真正开始处理的调用函数
  void processVisibleObject(MovableObject* mo,
     Camera* cam,
     bool onlyShadowCasters,
     VisibleObjectsBoundsInfo* visibleBounds);
```

1.3.3. RenderQueueGroup

- 这是一个在 RenderQueue 之下的分组的层次，粗略地决定了物体渲染的大概顺序
- 该类与 RenderQueue 的各种接口基本一致，比如有合并队列 merge 函数，添加 renderable 的 AddRenderable 函数等等
- 同时类似于 RenderQueue 与 RenderQueueGroup 的关系，该类与 RenderPriorityGroup 拥有相同的关系，比如拥有 RenderPriorityId 到 RenderPriorityGroup，拥有 RenderPriorityGroup 的迭代器便于遍历 RenderPriorityGroup

1.3.4. RenderPriorityGroup

- 存放同个优先级 renderables 的场所，以 QueuedRenderableCollection 的形式组织
- 在同个类里面的 renderables 都是在同个 RenderQueueGroup 里面同时具有相同的优先级。
- 将 Solid object 按照 pass 排序（减少 render state 的改变）或者按照视觉深度升序 or 降序排列，而对于 Transparent object 总是降序
- Renderable 进一步细分 solid(及其不同排序类型)和 tranparent(及不同属性类型)，拥有各种添加 renderable 的方法，为 addXXXRenderable，如 addSolidRenderable 为添加 Solid 的方法，而 addTransparentRenderable 为添加 Transparent 的方法

  综上所述，需渲染的物体分别经过 RenderPriorityGroup、RenderQueueGroup 分类后，由 RenderQueue 统一管理。

- 公有接口主要有：
  - 得到细分后基本物体 collection 的接口比如：
    - getSolidXXX() const;
    - getTransparentXXX() const;

- set&get&default 的下层组织方式（下面类说明）
  - addOrganisationMode
  - resetOrganisationModes
  - defaultOrganisationMode
- 如上所述各种队列都有的接口比如 merge，clear 等

1.3.5. QueuedRenderableCollection

类型别名有：

```
typedef vector<RenderablePass>::type RenderablePassList;

typedef vector<Renderable*>::type RenderableList;

typedef map<Pass*, RenderableList, PassGroupLess>::type PassGroupRenderableMap;
```

变量根据用途有不同命名，但类型大致为以上三种

- RenderPriorityGroup 有 5 个成员变量 mSolidsBasicm、SolidsDiffuseSpecular、mSolidsDecal、mSolidsNoShadowReceive、mTransparents，都是 QueuedRenderableCollection
- QueuedRenderableCollection 是存储 Renderable 和 Pass 的最终场所。通过多种排序实现 Renderable 和 Pass 的有序化。
  - 其中，排序包括
    - 小于排序
    - 深度递减排序
    - 基数排序。
  - 组织 renderable 和 pass 的方式包括
    - 按 Pass 分组
    - 按与 camera 距离升序
    - 按与 camera 距离减序
  - 以上按照组织方式和排序方式进行组合
- 选择不同的 visitor，函数原型

```
void acceptVisitor(QueuedRenderableVisitor* visitor, OrganisationMode om) const;
```

内部由 switch 根据 om 决定用哪个，实现如下：

```
switch(om)
{
case OM_PASS_GROUP:
    acceptVisitorGrouped(visitor);
    break;
case OM_SORT_DESCENDING:
    acceptVisitorDescending(visitor);
```

```
        break;
    case OM_SORT_ASCENDING:
        acceptVisitorAscending(visitor);
        break;
    }
```

简单地看其中一种 visitor 访问元素的实现：

```
void QueuedRenderableCollection::acceptVisitorGrouped(QueuedRenderableVisitor*
visitor) const
{
      //选择特定 list
    RenderableList* rendList = ipass->second;
    RenderableList::const_iterator irend, irendend;
    irendend = rendList->end();
      //对每个元素调用 visitor 进行访问
    for (irend = rendList->begin(); irend != irendend; ++irend)
    {
      visitor->visit(*irend);
    }
}
```

1.3.6. QueuedRenderableVisitor

- 虚类一个，使用前需继承
- QueuedRenderableVisitor 是按访问者模式设计的抽象接口。整个渲染过程实质上是通过访问者模式来完成的，也就是说，最终的渲染任务是交到 QueuedRenderableVisitor 手中
- 同时也意味着，如果想要遍历 QueuedRenderableCollection 中的 item，就必须实现访问者模式，因为 collection 中内部的组织方式决定于使用的排序方式
- 此类提供根据访问元素的不同，提供各类 visit 接口（重载函数）
  - Renderable
  - Pass
  - RenderablePass（一个由 Renderable 和 Pass 组成的 structure）

1.3.7. RenderQueueInvocation

此类可以让用户根据 viewpoint 来自定义渲染顺序，方便实现特效等，具体功能有：

- 决定是否 skip shadows
- 改变 solids 顺序
- 阻止 ogre 控制渲染状态 render state

同时也可以定义子类以获得更多的控制

其他的类似于 renderQueue，只不过后者自动处理 shadow，自动根据 pass 分组处理 solid，然后再处理 transparent objects 等。

```
class _OgreExport RenderQueueInvocation : public RenderQueueAlloc
    {
    protected:
        /// 目标QueueGroupID
        uint8 mRenderQueueGroupID;
        /// 相当于该类的ID，用在监听器中
        String mInvocationName;
        /// Solids 排序方式
        QueuedRenderableCollection::OrganisationMode mSolidsOrganisation;
        /// 是否关闭shadows处理
        bool mSuppressShadows;
        /// 是否关闭OGRE的 render state 管理?
        bool mSuppressRenderStateChanges;
    public:
        /** 构造函数
       使用GroupID和Invocation的名字够在
        */
        RenderQueueInvocation(uint8 renderQueueGroupID,
            const String& invocationName = BLANKSTRING);
        virtual ~RenderQueueInvocation();
```

```
        /// 得到 queue group id
        virtual uint8 getRenderQueueGroupID(void) const { return mRenderQueueGroupID; }

        /// 得到invocation 名字(如果没有设置则为空)
        virtual const String& getInvocationName(void) const { return mInvocationName; }

        /** 设置组织模式 */
        virtual void setSolidsOrganisation(
            QueuedRenderableCollection::OrganisationMode org)
        { mSolidsOrganisation = org; }

        /** 得到组织模式 */
        virtual QueuedRenderableCollection::OrganisationMode
            getSolidsOrganisation(void) const { return mSolidsOrganisation; }

        /** mSuppressShadow的set和get函数
        */
        virtual void setSuppressShadows(bool suppress)
        { mSuppressShadows =  suppress; }
        virtual bool getSuppressShadows(void) const { return mSuppressShadows; }

        /** mSuppressRenderStateChanges的set和get函数
        */
        virtual void setSuppressRenderStateChanges(bool suppress)
        { mSuppressRenderStateChanges =  suppress; }
        virtual bool getSuppressRenderStateChanges(void) const { return
mSuppressRenderStateChanges; }
        /** 对特定group进行调用，于此控制权会重新交到SceneManager，具体见上文动态分析*/
        virtual void invoke(RenderQueueGroup* group, SceneManager* targetSceneManager);
    };
```

1.3.8. RenderQueueInvocationSequence

- 一个包含 RenderQueueInvocation 线性序列的类，其中的所有 RenderQueueInvocation 均属于同一个 viewport
- 这个类的主要目的的保证 RenderQueueInvocation 实例在关闭之后会被删除，但该类被清除的时候，所绑定在这类的所有 RenderQueueInvocation 或者用户自定义的 RenderQueueInvocation 子类均会被删除
- 变量成员：
  - `String mName`
  - 一个 Invocation 的 list
    - `RenderQueueInvocationList mInvocations`
- 函数成员：
  - Name 的 get 函数
    - `const String& getName(void) const { return mName; }`
  - 两个添加 RenderQueueInvocation 的 add 函数，其中第一个相当于临时构造一个新的并且返回其指针
    - `RenderQueueInvocation* add(uint8 renderQueueGroupID, const String& invocationName);`
    - `void add(RenderQueueInvocation* i);`
  - 得到列表大小的函数
    - `size_t size(void) const { return mInvocations.size(); }`
  - 清除列表函数
    - `void clear(void);`
  - 得到某个 Invocation 的 get 函数
    - `RenderQueueInvocation* get(size_t index);`
  - 根据所以删除某个 invocation 的函数
    - `void remove(size_t index);`
  - 得到迭代器方便迭代调用
    - `RenderQueueInvocationIterator iterator(void);`
- 上面两个类被调用流程

```
RenderQueueInvocationSequence* seq;

RenderQueueInvocationIterator invocationIt = seq->iterator();

while (invocationIt.hasMoreElements())

   {

       RenderQueueInvocation* invocation = invocationIt.getNext();

       uint8 qId = invocation->getRenderQueueGroupID();

       invocation->invoke(queueGroup, this);

   }
```