西南科技大学

Southwest University of Science and Technology

本科毕业设计（论文）

题目：基于 OpenGL 的图形图像渲染引擎

学生姓名：王必宇
学生学号：5120140514
专　　业：信息与计算科学
指导教师：马新
学院(部)：理学院

教务处制表

基于 OpenGL 的图形图像渲染引擎

摘要

渲染技术是游戏引擎技术的核心，它包括从 CPU 传数据到 GPU，最后完成绘制的过程。然而渲染技术又是基于计算机图形学的，实现优秀的渲染引擎能使开发人员忽略渲染的过程而只关心逻辑开发，它的实现好与坏直接决定了一款视频游戏、电影产品的质量。

本人在介绍 3D 图形学基础知识的同时基于 OpenGL 技术架构、实现了一套包括资源管理器、底层渲染器、场景管理器的图形图像渲染引擎。资源管理器实现了对加载图像、向 GPU 传递着色器程序、加载模型资源等操作，主要职能在于减少对 CPU 和 GPU 时间和空间的浪费，优化渲染引擎效率；多场景管理模块管理所有单一场景，提供在任意情况下进行场景切换的接口。

底层渲染器是渲染技术的核心，它负责单个场景的渲染、管理工作。实现了包括三维世界中常用的天空盒，第一人称摄像机、第三人称摄像机，2D 摄像机，地面，模型，粒子系统以及更底层的 VertexBuffer，ElementBuffer，FrameBuffer，Shader 等核心模块。渲染中使用了裁剪算法，提高引擎渲染效率。

对于各子模块的实现技术，在论文实现中提供了详细说明。例如场景管理的裁剪算法，是将摄像机视口以外的模型全部裁掉，避免进行不必要的渲染；天空盒即用一个立方体盒子包住摄像机，并贴上 2D 纹理，使摄像机转动任意角度均会看到最外面的天空盒贴图。

关键字： 渲染引擎，OpenGL，C++，3D 场景，计算机图形学。

GRAPHICS AND IMAGE RENDING ENGINE BASED ON OPENGL

ABSTRACT

Rendering technology is the core of game engine technology. It includes the process of transferring data from CPU to GPU and finishing the drawing process.However, rendering technology is based on computer graphics, the implementation of a good rendering engine can make developers ignore the process of rendering and only care about logical development. Its good or bad implementation directly determines the quality of a video game and movie products.

This paper introduces the basic knowledge of 3D graphics and implements a graphics and image rendering engine which includes resource manager, bottom renderer and scene manager based on OpenGL technology.The resource manager implements the operations of loading images, transferring shader programs to GPU, loading model resources, etc. The main function is to reduce the waste of time and space of CPU and GPU, and to optimize the efficiency of rendering engine.The scene management module manages all single scenarios and provides an interface for scenario switching under arbitrary circumstances.

The underlying renderer is the core of rendering technology, which is responsible for the rendering and management of a single scene.It includes the sky box, the first person camera, the third person camera, the ground, the model, the particle system and the lower layer of the core modules, such as VertexBufferBull, ElementBuffer, Fframework, Shader and so on.Clipping algorithm is used to improve the efficiency of engine rendering.

For the implementation of each sub-module technology, the paper provides a detailed description.For example, the scene management clipping algorithm is to cut all the models outside the camera's view to avoid unnecessary rendering. The sky box covers the camera with a cube box and pastes 2D textures.Make the camera rotate at any angle to see the outermost sky box map.

Key words ： rendering engine, OpenGL , C ++, 3D Scene, Computer Graphics

第一章绪论

本章节主要介绍游戏引擎中渲染技术，以及渲染技术国内外发展概况，最后介绍笔者实现的渲染引擎的内容和实现及其意义。

1.1 游戏引擎渲染技术的阐述

游戏引擎顾名思义，是开发游戏的工具包，其中包含多个模块，例如渲染引擎，第三方软件开发包和中间件（例如集成 STL、物理、角色动画、人工智能等），平台独立层，游戏资源管理器，脚本系统等。在任何游戏引擎中，渲染引擎都是最大、最复杂的模块之一。

渲染引擎的架构大致都分为低阶渲染引擎，高阶视觉效果和后期处理两个部分。

低阶渲染引擎只关心三维世界物体的绘制。其工作基于对底层图形 SDK 如 OpenGL 或 DirectX 的封装。例如通常会对 Shader 程序和显存进行逻辑封装来模拟 CPU 上分配内存和向内存写入数据，然后运行程序的效果。然后在对 SDK 封装之后，还会将一些三维世界的常用组件进行封装，例如材质、摄像机、光照、纹理、地面、天空盒、模型，用来模拟真实的三维世界。

高阶视觉效果包括粒子系统，动画系统，地形系统等各种丰富三维世界表现的子模块。后处理则包括高动态范围光照、全屏抗锯齿等细节上优化表现效果的技术。最后再经由场景图或剔除优化去限制提交的图元数量以优化渲染引擎性能。

1.2 渲染技术国内外发展概况

20 世纪 50 年代，第一台显示器在 MIT 诞生，该显示器用类似于示波器的刷新式 CRT 来显示简单图形。这昭示着计算机图形学的萌芽。而后 60 年代的随机扫描显示器，70 年代的光栅扫描显示器，以及现在盛行的平板显示器的推出，均在展现计算机图形学在蓬勃发展的盛况。

1992 年，首个三维第一人称 FPS 游戏《德军总部 3D》发布。这款游戏由美国德克萨斯的 Software 公司制作，而后该公司又相继推出《雷神之锤》系列游戏，由于其所用引擎架构的相似，故被称为雷神之锤引擎家族。该引擎设计优化并且整洁，虽然略有过时，但是是纯 C 语言写的，其优秀的运行效率仍“老当益壮”。

1998 年，Epic Games 公司通过《虚幻》进军游戏行业，其所研发引擎 Unreal Engine 主宰游戏行业至今，虚幻引擎以其功能全面、渲染画面如其名“虚幻”著称，它几乎可以用来制作任意第一人称、第三人称的 3D 游戏。其最新版本 Unreal 4 提供非常方便的 UI 界面去制作 Shader，也提供了蓝图可视化这个完整的游戏脚本系统，又有名为 KIsmetde 的 UI 界面供编写游戏逻辑所用。但如此强大的引擎也带来了一些问题：由于其渲染效果的强大而导致其工具学习门槛较高，且使用费用较贵。

2005 年，Unity Technologies 公司发布 Unity 引擎。往往是游戏的优秀推动其发展游戏所用的引擎，而 Unity 引擎是通过引擎本身起家的。其最大的特点即跨 23 个平台，以及拥有非常友好的 UI 界面能让程序开发者，游戏策划，游戏美术都能参与到游戏研发中。渲染效率虽然在发布之初不敌虚幻，但其最新版本 Unity 2018 的渲染效果近乎可以认为其已经赶上了虚幻引擎。自中国手游产业爆发的 2014 年以来，几乎 99%的中国 3D 手游都出自该引擎，这就足以说明 Unity 引擎的优秀。该引擎融入了行业领先的实时全局光照技术 Enlighten，内置 UI 组件 UGUI 并广泛支持外部拓展（由此产生了另一款非常出名的 NGUI 拓展），内建强大的地形编辑器能让低级硬件亦可流畅运行广阔茂盛的植被景观，着色器编写使用 ShaderLab 语言并同时支持 GLSL、Cg 语言。海量的优势让其在当前引擎市场占据主导地位。但 Unity 引擎也有其缺点：工具数量有限，所以开发商必须给自己创作工具；做复杂和多样化的效果比较耗时。

其他例如 OGRE（开源）、Irrlicht（开源）、GameBryo Lightspeed、CryEngine 也都是非常优秀的引擎，也都在游戏引擎发展史上画上重要的一笔色彩，游戏开发者们由足够的理由尊敬前辈们做出的贡献。

1.3 本文研究的意义和内容

渲染引擎是渲染技术和软件工程的结合产物。渲染技术说到底其本质是计算机图形学技术，渲染引擎的发展也会带动诸如人工智能中的图像识别，计算机视觉等方向的发展。渲染引擎同时也是一个软件，其架构无处不渗透着软件工程的思想，架构的优良性直接影响渲染效率。又由于计算机图形学归根结底是数学在图形领域的应用，渲染技术的发展同样也有助于数学学科的发展。

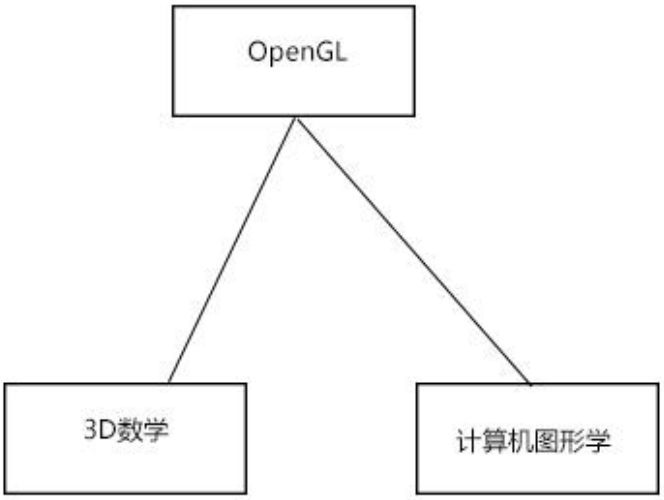
当今游戏行业内两大引擎：Unreal 和 Unity 优点虽足，但也有不少缺点：Unreal 的价高难学以及 Unity 的渲染效率不尽人意，笔者希望能有一款底层渲染引擎，能在使用难度低的前提下尽可能地提升渲染效率。同时希望能对一些特别的物体提供渲染支持，使其适合渲染某一种物体。

故笔者结合专业所学数学知识以及自学的计算机图形学知识和编程知识，设计并实现了一款图形图像渲染引擎，这款渲染引擎采用 C++语言开发，图形设备接口使用 OpenGL，能实现对三维世界的部分还原。实现摄像机、天空盒、模型、地面的逻辑封装。并对渲染过程中可能出现的问题实现了解决方案，例如使用摄像机裁剪算法避免可能会遇到的硬件性能问题，使用离屏绘制技术加强表现效果等。同时在各模块设计中适当加入了设计模式和软件工程思想，优化了渲染引擎的设计，使三维世界中存在的各种各样的计算更加严谨。

第二章 技术推演

本章将介绍笔者所作渲染引擎所涉及的技术和理论知识，仔细阅读能够更加理解本论文所讲述的技术原理。三维渲染引擎主要涉及到的知识点包括计算机图形学，3D 数学，OpenGL。

它们的关系是这样的：



OpenGL 封装了部分 3D 数学和计算机图形学的知识，可以理解为一种程序设计语言或者说一套工具，但是开发者仍需具备一些重要知识才能开发 3D 程序。

3D 数学是一门学科，线性代数、高等代数是它的子集，它还包含诸如图元可见性检测、相交检测等等知识。

计算机图形学也是一门学科，泛泛地说就是要讨论如何在计算机上绘各种各样的图。因为命令计算机绘出的图完全取决于开发者。可以使贴合实际的，也可以是虚幻的。

下面我们来讨论引擎中用到的这些知识。

2.1 计算机图形学

2.1.1 渲染流水线

从我们在 CPU 端输入顶点数据开始，就进入 GPU 的控制范围了。GPU 将执行一系列操作：顶点着色器->曲面细分着色器->几何着色器->裁剪->屏幕映射->三角形设置->三角形遍历->片元着色器->模板测试->深度测试->混合->屏幕映射。一般认为可编程完全控制的是顶点着色器和片元着色器。顶点着色器通常用于实现顶点的坐标变换和逐顶点着色；片元着色器

根据上一步插值后的片元信息，输出该片元的颜色，这里可以完成很多渲染技术，例如纹理采样技术。其他阶段均为弱控制阶段或完全不可控阶段，弱控制包括通过命令去开启或关闭某个功能等。

2.1.2 光栅化

我们输入的是顶点数据，那么是谁让一个顶点变成一个片元呢？实际上是光栅化在作用。光栅化把几何图元变成屏幕上的二维图像，它决定窗口中哪些整形栅格区被基本图元占用（一般是一系列三角形），并分配一个颜色值和一个深度值到目标区域。这个把物体的数学描述和与物体相关的颜色信息转换到屏幕上，用于对应位置的像素以及用于填充像素的颜色，这个过程称为光栅化。

2.1.3 材质与光源

光源类型大致可分为 3 种，分别是方向光（想象为太阳光，位置在无穷远处，光照不会随着距离而衰减，因为已经传播了无穷远的距离），点光源（想象为灯泡），聚光灯（想象为舞台的聚光灯）。点光源在图形学中通过常亮衰减因子、线性衰减因子、平方衰减因子来模拟。聚光灯的本质也是点光源，不过加入了照射方向和内锥角度、外锥角度来描述。

下面给出光照颜色的着色计算公式（Phong 着色模型研究成果）：

方向光源

$$I_{later} = I_{before} * Color \qquad (2\text{-}1)$$

其中右边 I 表示原始光强，Color 表示光的颜色。

点光源

$$I_{later} = \frac{I_{before} * Color}{k_c + k_l * d + k_q * d * d} \qquad (2\text{-}2)$$

其中 kc 表示常数衰减因子，kl 表示线性衰减因子，kq 表示平方衰减因子。d 表示着色点到光源的距离。距离越远光线衰减越严重。

聚光灯

（1）情况一：着色点位于外锥的外边，照射不到

$$I_{later} = 0 \qquad (2\text{-}3)$$

（2）情况二：着色点位于外锥的里边，内锥的外边，光强根据角度衰减

$$I_{later} = \frac{I_{before} * Color}{k_c + k_l * d + k_q * d * d} * \frac{(\cos\theta - \cos(\phi/2))^p}{\cos(\alpha/2) - \cos(\phi/2)} \qquad (2\text{-}4)$$

α表示外锥角度，Φ表示内锥角度。p 表示指数因子。θ表示光和人的夹角，即如果夹角大于外锥则根本照射不到。

（3）情况三：着色点位于内锥里边

$$I_{later} = \frac{I_{before} * Color}{k_c + k_l * d + k_q * d * d} \qquad (2\text{-}5)$$

当着色点位于内锥时，聚光灯可近似认为是点光源。

反射类型表示物体对同一种光的着色情况，图形学中认为有三种类型，分别是环境光反射（来自四面八方的光，没有确切来源，而是各种光经过各种物体反射的结果），漫反射（粗糙表面发生的反射）和镜面反射（光照射到镜子上发生的反射现象）。在计算机图形学中，我们将这 3 种反射称为物体的材质，即可以认为材质是物体表面对光的反射颜色。

下面给出反射类型的着色计算公式（Phong 着色模型研究成果）：

环境光反射

$$I_{\text{ambient}} = I_{light} * M_{ambient} \qquad (2\text{-}6)$$

M 表示材质对环境光的吸收，等式右边 I 表示环境光的光强。

漫反射

$$I_{diffuse} = I_{light} * M_{diffuse} * \max((\overrightarrow{normal} \bullet \overrightarrow{direction}),0) \qquad (2\text{-}7)$$

M 表示材质对漫反射颜色的吸收，normal 表示照射平面的法线，direction 表示光的照射方向。当法线与照射方向夹角为 0° 时，光强最大；夹角为 90° 时，光强最小；夹角＞90° 时，漫反射光对该表面不起作用（此时小于 0 故 max 函数令其等于 0）。

镜面反射

$$I_{specular} = I_{light} * M_{specular} * \max((\overrightarrow{normal} \bullet \overrightarrow{direction}),0)^p \qquad (2\text{-}8)$$

M 表示材质对镜面反射颜色的吸收。p 是指数因子，镜面反射与漫反射的不同只在于，镜面反射通过指数形式加快等式右边值的衰减，贴合实际生活发生的镜面反射。

无论是任何类型的光源，都会有环境光属性、漫反射属性、镜面反射属性使得对材质的的计算能够实现，材质也会有对应的 3 个属性和光源进行计算。

2.1.4 2D 纹理与纹理坐标

2D 纹理其实就是一张贴图，也就是 bmp、jpg 等类型的图片，三维世界中的颜色大多数都是贴图带来的，贴图把没有任何颜色而只有位置信息的模型变得有颜色。

三维世界中需求把 2D 的纹理贴到一个物体上，这就有了纹理坐标（或者叫 uv 坐标）的概念。纹理坐标以左上角为(0,0)点，右下角为(1,1)点，是属于一个顶点的属性，用来指示这个点需要贴 2D 的哪个对应的点。比如希望在矩形上贴一个 2D 纹理只需要在左上角写(0,0)，左下角(0,1)，右下角(1,1)，右上角(1,0)就可以把一个 2D 纹理贴到矩形的游戏物体上了。

2.2 3D 数学

这里讲述一些 3D 世界中涉及的数学方面问题，向量和矩阵的运算属于 3D 数学的基础，故论文中不会过多提及，而会将更多注意力放在渲染方面的话题。

3D 数学所要解决的最根本问题，是如何将 3D 的游戏世界显示在 2D 的显示器上。

2.3.1 坐标系

坐标系有两种：左手坐标系和右手坐标系，OpenGL 使用的是右手坐标系。右手掌心面向自己，大拇指向右，食指向上，其它三个指头指向自己，这就是右手坐标系。大拇指表示坐标系 x 轴正方向，食指表示 y 轴正方向，其它三指表示 z 轴正方向。

下面介绍一些三维世界常用的坐标系。

世界坐标系：

想象为地球的经纬度，每个点在世界坐标系都有一个唯一的标识(x,y,z)，用于在三维世界中唯一标识一个位置。

物体坐标系：

以物体自己左上角或重心为零点的坐标系，用于方便地表示物体 A 在物体 B 哪个位置。

摄像机坐标系：

摄像机就好像人的眼睛，摄像机自己位于坐标系零点，用于表示一些可见性相关问题，例如：某个物体是否在屏幕上？两个物体重合，谁在前面？

2.3.2 物体变换

变换物体意味着，该物体的所有点被移动到一个新的位置，使用同一坐标系描述变换前后点的位置。在引擎中，有 3 种方式进行变换，分别是：矩阵，四元数，欧拉角。由于绝大多数引擎以及 OpenGL、DirectX 均采用矩阵形式实现变换，下面将讲述用矩阵进行物体变换的方法。在解释完平移、旋转、缩放后，会说明如何用这 3 个矩阵表示物体在某个坐标系下的位置。

平移变换

$$M_{pos}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ t_x & t_y & t_z & 1\end{bmatrix} \qquad (2\text{-}9)$$

Tx,ty,tz 表示变换后物体的下标。

旋转变换

我们可以沿着任意轴进行旋转，若是沿自身 x,y,z 轴旋转就简单得多，下面给出绕任意轴旋转的矩阵：

$$M_{rotation}=\begin{bmatrix}n_x^2(1-\cos)+\cos\theta & n_xn_y(1-\cos\theta)+n_z\sin\theta & n_xn_z(1-\cos\theta)-n_y\sin\theta\\ n_xn_y(1-\cos\theta)-n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_yn_z(1-\cos\theta)+n_x\sin\theta\\ n_xn_z(1-\cos\theta)+n_y\sin\theta & n_yn_z(1-\cos\theta)-n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta\end{bmatrix} \qquad (2\text{-}10)$$

其中θ表示旋转角度，n 表示旋转轴的方向向量。

缩放变换

一般在游戏引擎中，会将物体沿着自身的坐标轴进行缩放。缩放的本质起始就是，将物体的每个点的 x,y,z 值分别进行缩放。下面给出沿着自身坐标轴进行缩放的矩阵：

$$M_{scale}=\begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{bmatrix} \qquad (2\text{-}11)$$

其中 k 为缩放因子，可以对每个轴用不一样的缩放因子。

应用矩阵变换（在右手坐标系中）

我们需要将 3 大矩阵相乘，以拿到物体在世界坐标系中的位置，然后交给 GPU 进行绘制。这就关系到三大矩阵如何组合在一起来表示。

$$\mathrm{WorldTransform} = \mathrm{M_{position}} * \mathrm{M_{Rotation}} * \mathrm{M_{scale}} \qquad (2\text{-}11)$$

旋转要最先起作用，因为旋转总是相对于原点而言的。比如说，如果你先旋转，后平移，那么物体能够自转。但反之如果你先平移后旋转，那么这个对象不会自转，而是会绕世界坐标系原点旋转，这和我们通常的需求就不一样了。

2.3.3 投影

三维世界中常用的投影有两种：正交投影和透视投影。正交投影给玩家一种 2D 化的眼睛，它看到的东西不会产生近大远小的感觉。而透视投影则反之，会模拟三维世界中人的眼睛看物体近大远小的效果。这两种投影也是通过矩阵形式来实现的，下面给出正交投影和透视投影矩阵。

正交投影

$$M_{Orth}=\begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far-near} & 0 \\ 0 & 0 & \frac{\mathrm{near}}{\mathrm{far\text{-}near}} & 1 \end{bmatrix} \qquad (2\text{-}12)$$

其中 width，height 分别为视口宽、高（单位为像素），near 为眼睛能看到的最近点，far 为眼睛能看到的最远点。

透视投影

yScale=cot(fov/2);fov 为眼睛的可见角度。

xScale=yScale*height/width;

$$M_{Perspectiv\ e}=\begin{bmatrix} \mathrm{xScale} & 0 & 0 & 0 \\ 0 & \mathrm{yScale} & 0 & 0 \\ 0 & 0 & \frac{far}{far-near} & 0 \\ 0 & 0 & \frac{\mathrm{-near*far}}{\mathrm{far\text{-}near}} & 1 \end{bmatrix} \qquad (2\text{-}13)$$

2.3.4 视景体与 3D 裁剪

人在现实世界通过眼睛接收外部所见，那么在游戏中玩家的眼睛就是摄像机，负责把游戏中玩家看到的东西，输出到屏幕上。摄像机可见的范围我们称之为视景体，是一个横截面呈梯形的六面体，摄像机的近视点是梯形的上底，远视点是梯形的下底，梯形的高就是 far-near，两条斜边的延长线交点就是摄像机的可视角度。远视平面的宽高就是摄像机视口的宽高。

由于摄像机是引擎开发者以逻辑进行封装的一个类，并非真正存在于 GPU 的一个物体，所以要绘制哪些物体都是由引擎底层决定的。但是如果摄像机看不到其他地方的物体，那么这个不被摄像机观察到的物体就没有必要进行绘制，这就产生了使用裁剪算法的需求，裁剪算法判断一个物体在不在摄像机可视范围内，在的话则把这个物体的顶点信息传输到 GPU 进行绘制，不在的话则不传输到 GPU 以节省 CPU->GPU 的传输数量。

视景体有 6 个面，实际上我们只需要对近裁剪面（近视点）进行裁剪，因为如果是上下左右或者远裁剪面定义的半空间内，则接受它，否则应该直接拒绝进行渲染。在近裁剪面过程中，可能会出现裁剪平面将三角形分割为多边形的情况，此时应该将其再次切割，然后再进行裁剪。

裁剪算法应该在渲染列表传递数据到 GPU 时被运行,更准确地说是应该在世界坐标空间或者相机坐标空间或投影坐标空间被运行，执行流程用伪代码表示：

```
//输入所有需要渲染的三角形，在这里进行裁剪，输出裁剪后在视景体内的三角形
foreach(triangle in inputTriangle)
{
Set<Triangle> inside=Sprite(triangle,nearPlane);
If(inside.size()==2)
{
output.insert(inside.at(0));
output.insert(inside.at(1));
}
Else if(inside.size()==1)
{
output.insert(inside.at(0));
}
}
```

通过裁剪的三角形会被添加到 Output，执行流水线下一步骤。

2.3 OpenGL

OpenGL 可以理解成 GPU 上的 C 语言，它本质和 C 语言一样并非是一门语言，而是一个标准，OpenGL 委员会提出标准后让硬件厂商和操作系统去实现这套标准，从而实现绘图功能。OpenGL 做的事情实际上就是在 CPU 和 GPU 之间搭建一个桥梁，开发者可以通过 OpenGL 在 CPU 端操作 GPU 数据。下面会讲述一些使用 OpenGL 开发的具体概念和用法。

2.3.1 OpenGL 指令

OpenGL 本身是一个巨大的状态机，开发者通过命令去改变 OpenGL 的当前状态，例如是否开启深度测试？是否开启 Alpha 混合？用什么颜色擦除缓冲区等等。

2.3.2 Shader（OpenGL Shader Language）

在计算机图形学里面，有说到这样两个名词：顶点着色器和片元着色器，顶点着色器处理顶点数据，片元着色器处理顶点经过光栅化之后的片元。而顶点着色器和片元着色器可以组成一个 GPU 程序，从 CPU 传过来的顶点数据经过顶点着色器，进行光栅化变成片元，然后传到片元着色器。Shader 在 OpenGL 中这样编译成可执行程序：

①读取 VertexShader，FragmentShader 的字符数据（const char*）到内存。

②将 Shader 源码放到 GPU，并分别编译 VertexShader，FragmentShader。

③创建一个 GPU 程序，将两个 Shader 绑定到 GPU 程序上，进行链接。

有几个顶点则会调用几次顶点着色器，屏幕上有几个像素点则会调用几次片元着色器。

在 GPU 端写入变量的值是通过标识符的形式的，可以理解为每种类型的变量会有一个槽，槽内分为 1、2,、3、4、……，等很多孔，这些孔被称为 location，得到 location 后就可以通过 OpenGL 命令行，将内存里的数据传送到显存去。

2.3.3 Buffer（缓冲区）

有程序的地方必然有数据，在有独立显卡的机器上运行的 Shader 读取的是位于显存的数据。与内存相同，显存也需要开发者手动申请。申请之后就可以在显存里面写数据了。Shader 在绘制的时候会到当前正在使用的显存去取顶点数据，当然这需要开发者告诉 GPU 怎么去读显存里的数据。

缓冲区也是分类型的，引擎中实现的缓冲区有 3 种：

①VertexBuffer：直接存储顶点数据，GPU 拿到一个数据就绘制一个顶点。

②ElementBuffer：让 GPU 通过索引方式去拿当前 VertexBuffer 的顶点数据。

③FrameBuffer：俗称离屏绘制，开辟一个新的“画板”进行作画，可以实现画中画的效果。

OpenGL 标准非常庞大，并非三言两语可以讲完的。这里讲到的只是引擎中用到的部分知识。而 OpenGL 实际上帮我们做的事情非常少，用得更多的还是计算机图形学和 3D 数学的知识。

第三章 引擎设计与架构

3.1 引入的外部库

引擎将需要使用的外部库以及代码中使用的宏定义写在 GameDefine/Main/ggl.h 中，目前认为需要使用以下第三方库，后续如有需要将不断扩充：

①C++ STL 库。使用 STL 的容器、算法、迭代器、IO 库进行更方便的开发。

②glfw 平台无关库，能让 OpenGL 在不同操作系统下均能正确地弹出窗口。

③glew 库，由于部分操作系统（比如 windows）上只实现了 OpenGL 1.1 也就是固定功能管线，而 OpenGL 2.0 以上才出现可编程管线。在实际开发中我们需要更高级别的特性，故需要引入此库。

④FreeImage 库，由于现在已存在的图片格式较多，不可能一一对这些图片编写解码程序，故引入此库进行图片的解码。

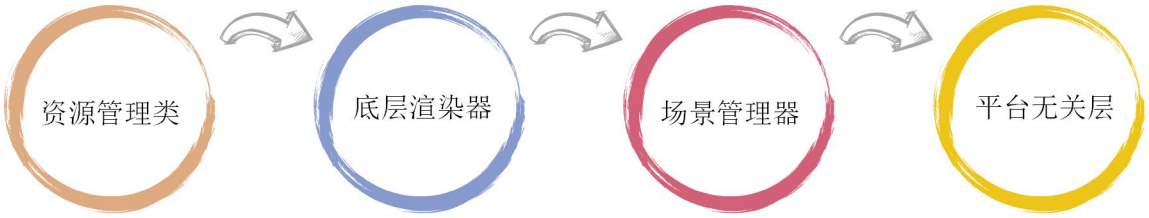
⑤glm 库，这是一套 OpenGL 的数学库，提供向量、矩阵等数学功能支持。

3.2 总体架构

在平台无关层之上是引擎的总体，这里分为三大模块：资源管理模块，底层渲染模块，多场景管理模块。当然还有更底下的平台无关层，不过这和渲染技术关系不大。

资源管理模块提供对磁盘资源加载到内存、以及申请 GPU 资源等操作，它将提供渲染素材供底层渲染器使用；底层渲染器装载了渲染引擎所有核心逻辑和算法，负责进行游戏物体的渲染工作；场景管理模块管理所有场景并知道当前场景是谁，它并不关心场景内如何实现，只执行当前场景的逻辑，所有场景须注册到 SceneManager 才能正常运行，向底层渲染器隐藏了当前场景和操作系统细节。

图示如下：



使用标准 C 语言 IO 函数和 OpenGL 标准加载卸载资源函数，只管申请资源和释放资源，而不关心其他的。

只管渲染一个游戏物体需要做的事情，所需的美术资源由资源管理模块负责加载，底层渲染器使用这些美术资源绘制图像。

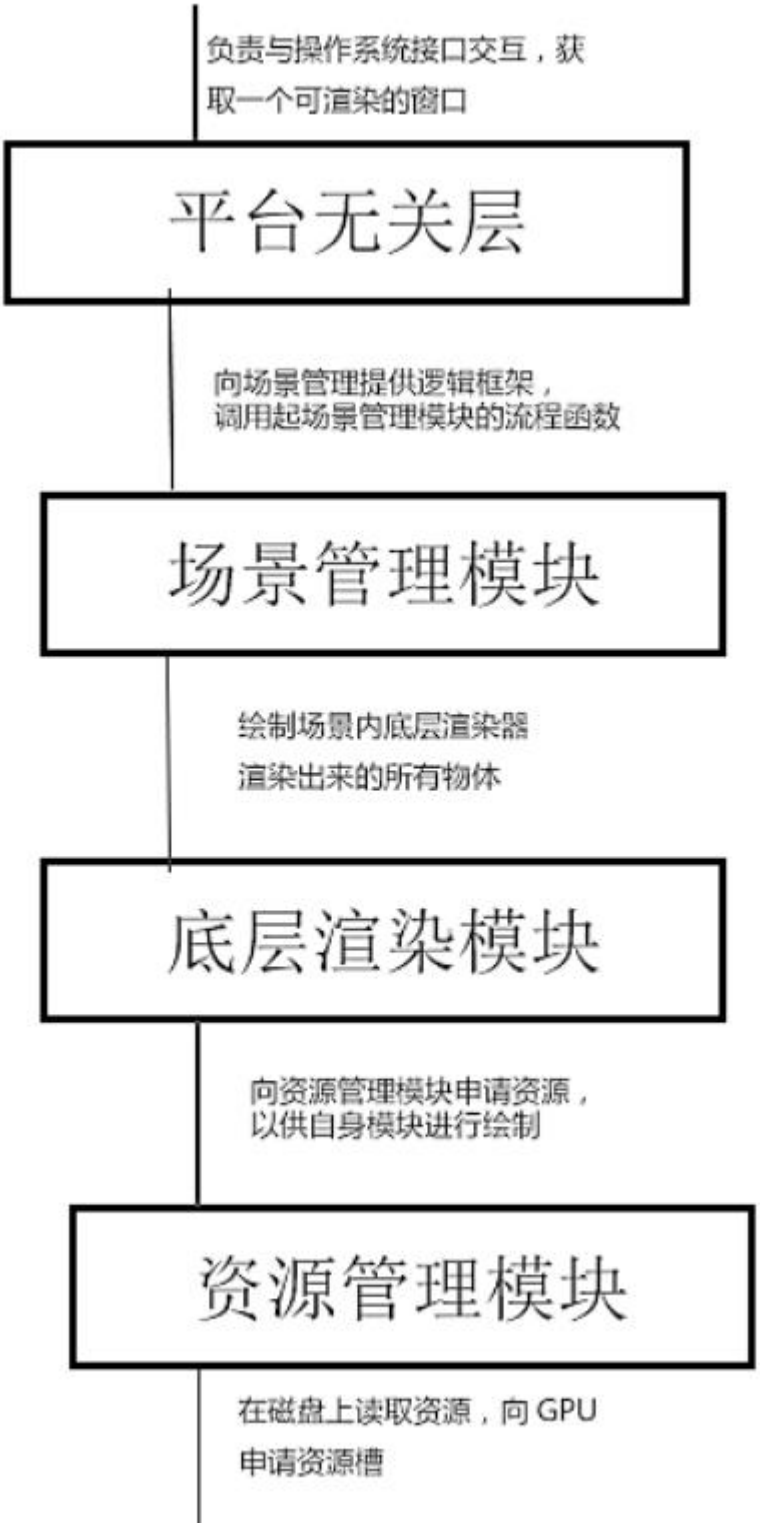
场景管理模块不知道一个场景是如何渲染的，但持有所有场景的指针并知道哪个是当前场景。能够从全局角度操作当前场景。

只管调多场景管理模块的流程函数，而不关心当前运行的是哪个场景。

这样就定义了引擎的整体工作模式。资源管理类会随着所需资源类型的增加而扩充，但不会影响其它模块已经实现的功能。底层渲染器会随着功能的逐渐强大而要求资源管理器能够提供新的美术资源，并不断完善其内部实现，但完善自身内部实现并不会使其它模块受到任何影响。场景管理模块的，仅限于加强对当前场景的管理支持而提供新的接口，一个物体

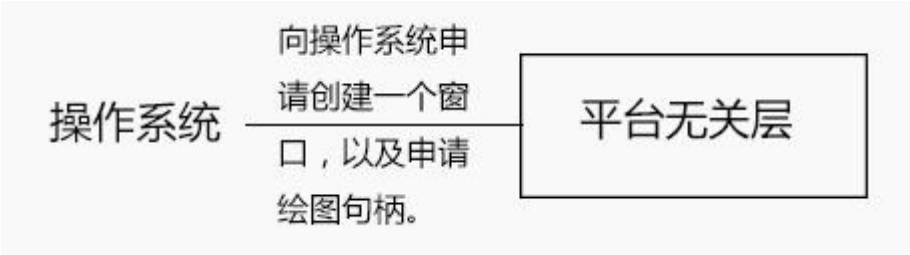
的逻辑如何更新、资源如何加载它都不会关心。平台无关层只管显示一个窗口即可，它的更新几乎是没有的，就算有也与其它模块没任何联系。这样，就设计了一个高内聚低耦合的引擎整体架构。

层次关系示意图如下：



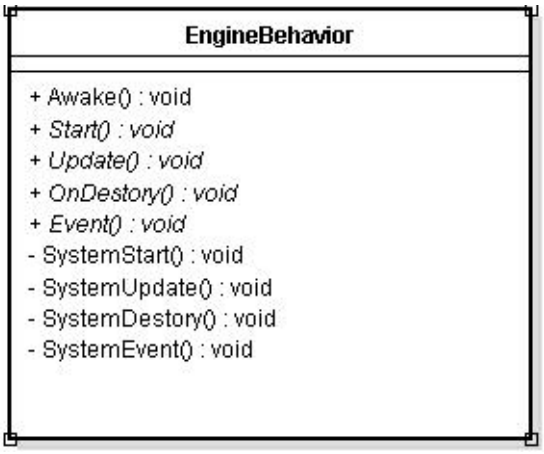
3.3 平台无关层

平台无关层的功能相当于 main 函数，其具体职能如下：
①注册并弹出一个 OpenGL 渲染窗口。
②调用起场景管理模块的逻辑框架。
平台无关层仅仅和场景管理模块交互，具体交互方式是这样的：
①向外和操作系统交互



②向内和场景管理交互

平台无关层仅仅包含一个类，类的设计图如下：



它需要利用 OpenGL 跨平台特性来实现，定义一个类，在类的初始化函数中进行窗口的初始化，其接口设计如下：

其中 Awake()函数是由操作系统的 main 函数调用，调用该函数后应该能实现对 OpenGL 渲染窗口的初始化。

游戏循环开始函数 SystemStart()在 Awake()最后被调用，执行开始逻辑，在 SystemStart()之后将开启软件的无限循环直到游戏退出。

游戏更新函数 SystemUpdate()被 SystemStart()调用，执行每帧更新需要做的事情，比如交换缓冲区。

消息处理函数 SystemEvent()监听由操作系统发到进程的消息，并把消息转发给

SceneManager 类的消息处理函数和 EngineBehavior 的派生类的消息处理函数。

清理全局资源函数 SystemDestory()被 SystemStart()调用，在退出游戏时执行窗口销毁时的资源清理。

另外还有几个虚函数 Start()、Update()、OnDestory、Event;是空函数，让开发者继承 EngineBehavior 类并重写这些虚函数，C++的多态特性就得以大显神威，调用到未来人写的方法。在这些方法里，开发者可以实现自己的全局性质的逻辑。比如注册自己的场景到 SceneManager，点击某地方时关闭游戏等。

这样就实现了一个游戏的初始化->游戏循环->监听消息->游戏循环->销毁的过程。

3.4 场景管理

场景管理在全局概念上管理所有场景及其渲染，它不是一个单一的类，其职能如下：

①执行当前场景的初始化、触发事件、更新、渲染、销毁过程。

②收集当前帧需要绘制的顶点数据，进行一系列处理后将顶点数据传到 GPU。

③提供场景注册、场景切换功能。

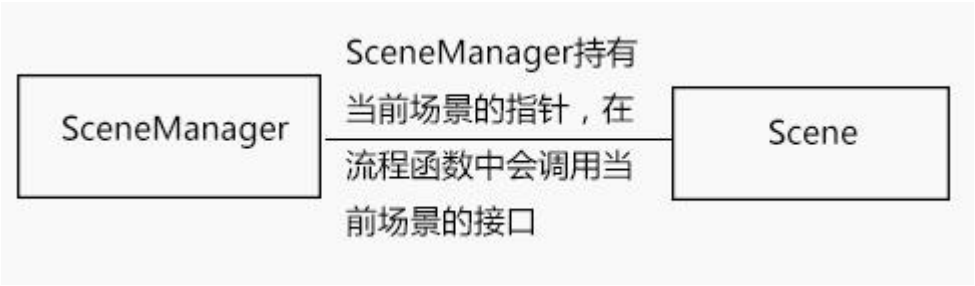
④管理 OpenGL 状态机的全局状态，例如是否开启 alpha 混合、是否开启深度测试等。

场景管理模块内部氛围 SceneManager 和 Scene，分别对应全局场景和单个场景（值得一提的是 Scene 是所有场景的基类）。下面介绍这些元素。

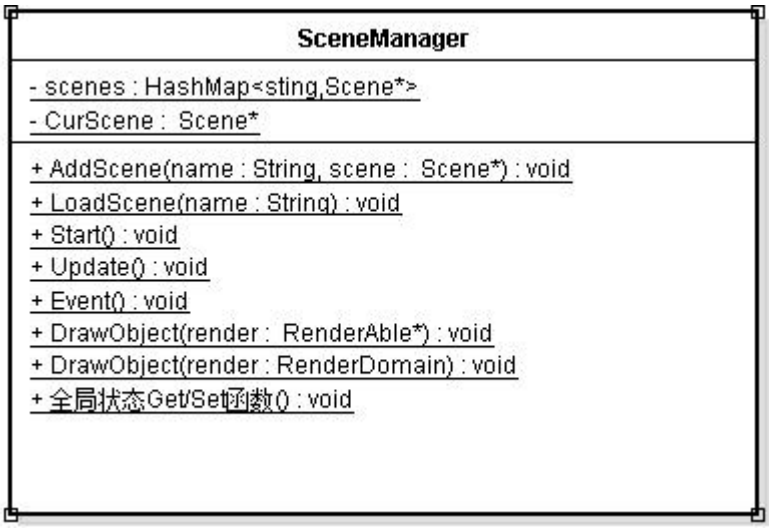
3.4.1 SceneManager

SceneManager 管理所有 Scene 并知道谁是当前 Scene。管理方法是通过注册、加载、销毁 Scene 的接口来实现。

SceneManager 和 Scene 的交互



下面给出类的定义：



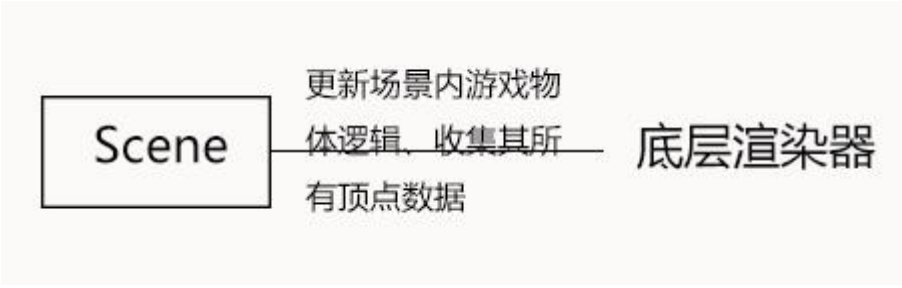
SceneManager 使当前场景的逻辑能够正常运行，场景通过 AddScene(string name,Scene* scene)函数被加载到 SceneManager 中管理起来。在 LoadScene(string name)函数中执行卸载上一场景和初始化新的场景，在 Update()中执行当前场景的流程函数，在 Event()中将事件转发给当前场景。是在单场景之上的多场景管理器。
DrawGameObject(RenderAble* render)和 DrawGameObject(const RenderDomain& render)提供渲染游戏物体的功能，填入合适参数后，参数所代表的可渲染对象会被正确渲染。其中前者适用于更方便地对 RenderAble 的基类进行渲染，后者则是更通用的函数，只需要构造一个逻辑上的 RenderDomain 对象并输入正确参数就可以被渲染。

由于 OpenGL 是一个状态机，它只知道当前状态是什么，并且设定以后如果不去修改，就一直会是某个状态。故 SceneManager 也承担了管理 OpenGL 当前状态的职责，需要被关心的状态会被添加到全局状态中，被 SceneManager 知晓当前状态并管理起来，同时提供接口让外部设置和获取。

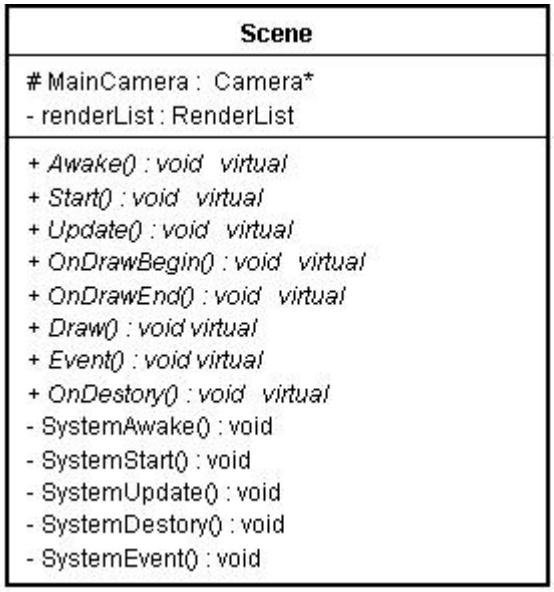
3.4.2 Scene

一个 Scene 表示单个场景，一个场景有初始化、更新、销毁的过程，同一时刻只会有一个场景，渲染引擎渲染的实际上就是该场景内的所有物体，Scene 类是所有场景的基类，继承 Scene 并重写 Scene 的虚函数就可以完成场景的渲染工作。

Scene 和底层渲染器的交互



Scene 类定义如下：



其中带 System 字样函数是引擎内部定义函数，不供用户重写的。其余虚流程函数都应该是空函数，供用户虚函数重写以完成逻辑开发。一个场景应该至少有一个摄像机和一个渲染列表，当然也可以有多个摄像机和与之对应个渲染列表。

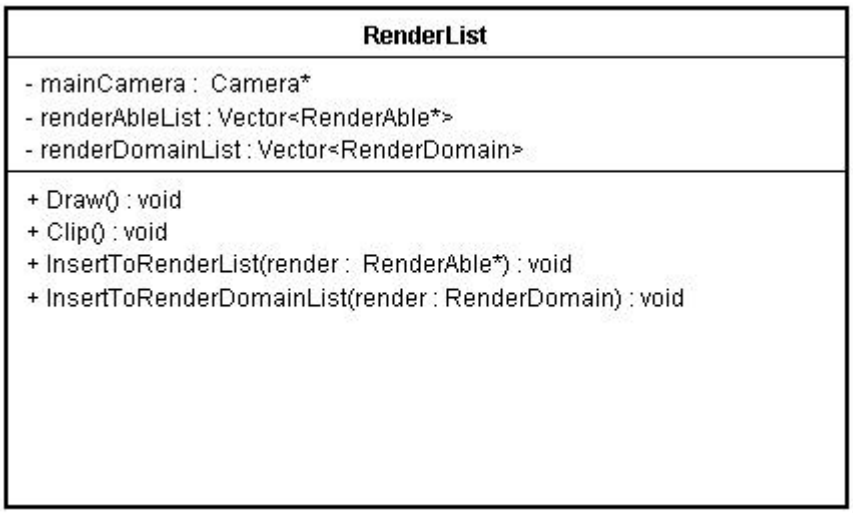
3.4.3 渲染列表

渲染流水线绘制的起点是 CPU 端输入的顶点集。在引擎中，这些顶点在开发者绘制命令的后并不会直接进入 GPU 开始绘制，而是会进入到渲染列表中，待其进行一系列处理后，再进行绘制。

渲染列表被组合在 Scene 中，一个场景默认会拥有一对摄像机-渲染列表组合。每个游戏物体的 Draw 函数会将渲染信息插入到当前场景的渲染列表中。

一个单渲染列表的场景渲染会发生这样的事情：

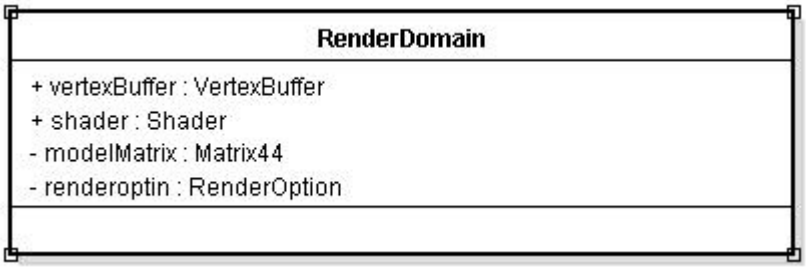
在场景中需要被渲染的一系列顶点数据会进入 RenderList。RenderList 会与一个摄像机进行关联，来对一个摄像机看到的物体进行一系列处理。然后通过测试的顶点才会传到 GPU。RenderList 的定义如下：



游戏物体进行渲染时会发生裁剪->剔除->绘制的过程。每次绘制结束后会清空渲染列表，以待下一帧绘制的时候添加下一帧需要被绘制的游戏物体。

在类中其中最关键的是容器 std::vector<RenderAble *>。RenderAble 是所有需要绘制的物体的基类，拥有足够多的信息供 GPU 进行绘制。其绘制类型、所需全局环境将通过自身变量的形式告诉 RendList，RendList 经过适配后将顶点数据传到 GPU 端进行绘制。这个类在后面会详细介绍。

std::vector<RenderDomain>是一个更加通用的容器，其中 RenderDomain 类定义如下：



如定义所见,这个类是一个渲染数据的逻辑模型,只持有顶点数据、Shader、ModelMatrix 和渲染选项，也就是说也可以有较为特殊的类不需要继承 RenderAble 也可以被渲染，他可以通过这种较为特殊的方式进入渲染列表。

也就是说，对于 RenderDomain 的渲染可以这样做：

渲染列表的函数会在 Scene 被调用，来完成当前场景中游戏物体的渲染。

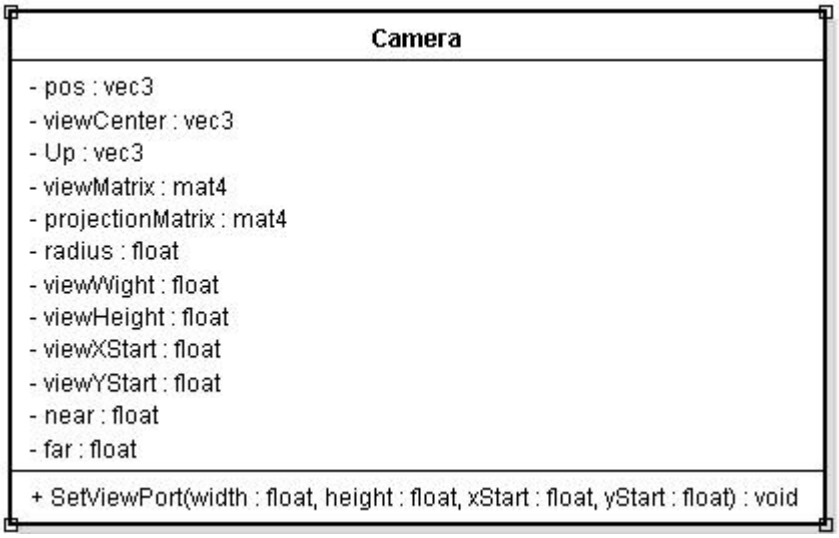
3.4.4 摄像机

摄像机相当于玩家的眼睛，而开发者希望玩家作为主角观察游戏世界，还是希望玩家作为第三者观察主角，这都取决于开发者对摄像机逻辑信息的更新。

摄像机在引擎中的作用是这样的：

摄像机类保存自己的值，然后被 RenderList 读取进行加工处理，然后传递到 GPU 作为变量进行绘制。

摄像机类定义如下：



一个摄像机需要知道位置、视点、Up 向量，还需要有一个模型视口矩阵 viewMatrix 和投影矩阵 ProjectionMatrix 和视景体 Frustum 以确定渲染范围，然后为了定义视口还需要视野的信息等。这些信息熟悉图形学的读者应该非常清楚，是一个摄像机必须有的属性。这些属性都将被 RenderList 读取，用于游戏物体的绘制。

SetViewPort 函数定义视口大小，该函数在 Camera 的构造中被用默认的变量值调用，在此函数中需要初始化视口矩阵和投影矩阵的值。

3.5 资源管理

资源管理类管理所有 CPU 端和 GPU 端分配的系统资源。它的具体职能如下：

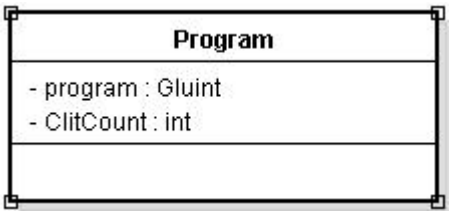
①读取磁盘上外部资源的能力，并能够将其转化为提供底层渲染器需要的资源类型进行返回。

②对于分配的内存和显存提供管理方案，不再使用的资源能够被正确释放。

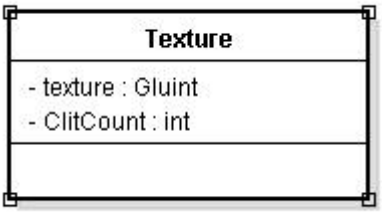
也就是说，资源管理模块的数据流是这样的：

下面讲述具体设计。

3.5.1 GPU 程序的内存模型

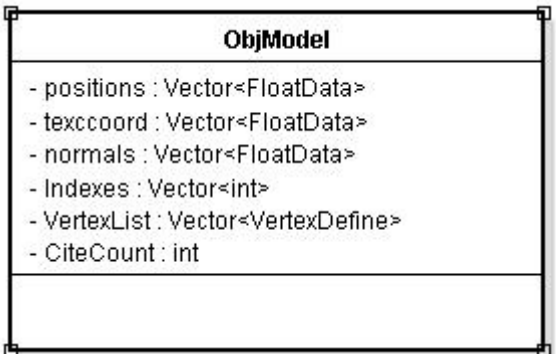


3.5.2 纹理类型的内存模型

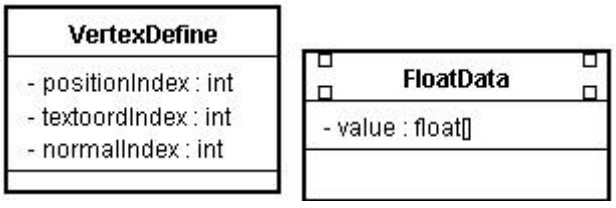


所有 GPU 资源通过模拟引用计数的形式实现管理，GLuint 类型成员表示该资源在 GPU 的标识，CiteCount 表示当前引用数。并提供一些必要的函数让程序能够正常运行。

3.5.3 obj 文件的内存模型



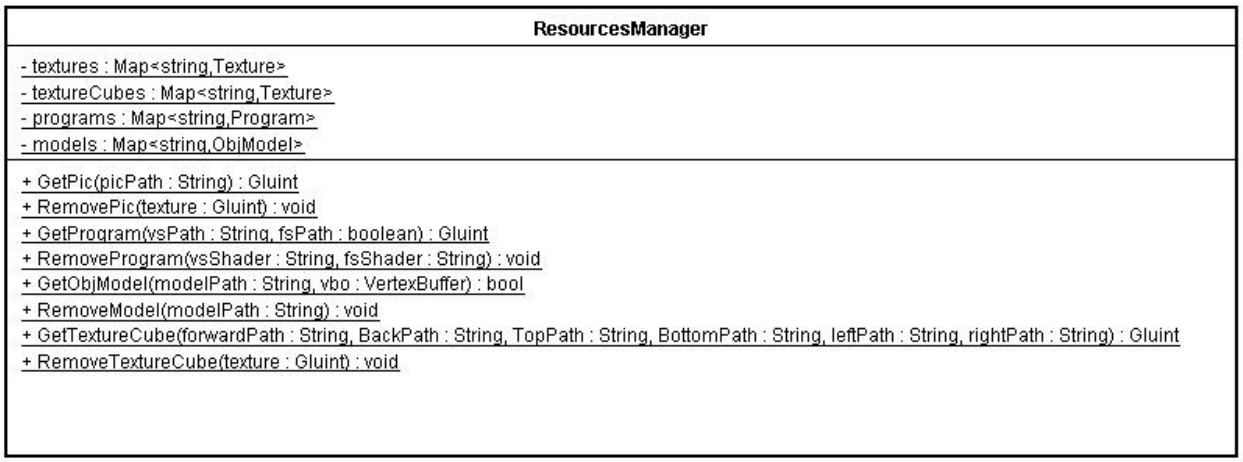
其中泛型成员定义：



obj 类型的模型文件格式是顶点数据+绘制指令的形式，故可以将 Obj 文件的定义保存在这里。

3.5.4 资源管理类接口定义

目前需要管理的只有 4 种资源，后续版本更新后将不断增加。



管理的方式均为文件名+内存模型。由于一个游戏中可能有多个地方用了同样的纹理，所以应该避免其反复读取磁盘上的文件和申请显存。Shader 同理，因为往往同类型的游戏物体会使用同样的 Shader，而通过 IO 加载 Shader 到内存，并送到显存编译、链接同样是非常耗时的工作。而 Obj 文件往往就是一个几十万行的指令集合，每次加载模型就让 IO 去读取、解码是非常耗时的工作，故在读取一次后便持有经过解析的 Obj 文件的引用，再次获取时只需要把内存中的数据返回即可。

对于每种资源提供加载、卸载功能，随着游戏物体的增加，资源管理类的接口也将不断增加，但这并不会影响已有的接口，也就不需要改动任何代码了。

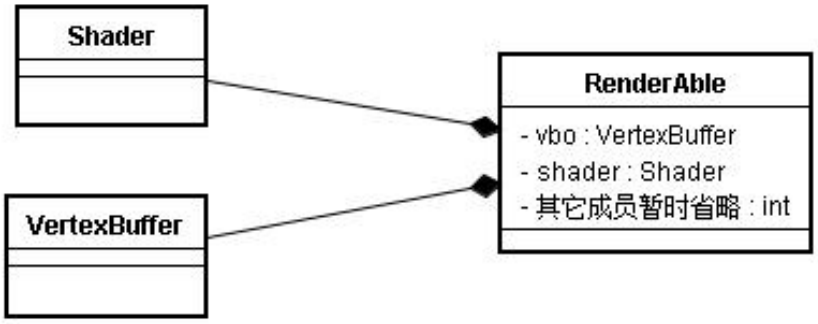
3.6 底层渲染——SDK 相关部分

底层渲染模块是最为复杂的，总体上划分为 SDK 相关部分和 SDK 无关部分。

SDK 相关部分职能如下：

①在 CPU 抽象出 GPU 相关信息。

该模块会直接持有底层渲染器所申请的大部分资源，它在架构中的职能是这样的：被之后会介绍的可渲染对象的基类所组合，将 SDK 部分的代码封装到 VertexBuffer 和 Shader 中，实现 SDK 相关部分和 SDK 无关部分的解耦合。

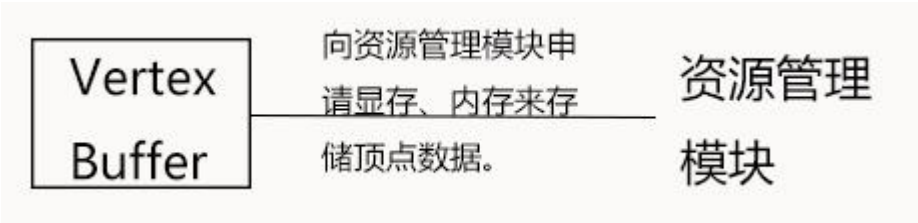


其中 RenderAble 是可渲染对象的基类。这幅 uml 图的意思就是说，VertexBuffer 和 Shader 会被可渲染对象的基类组合。

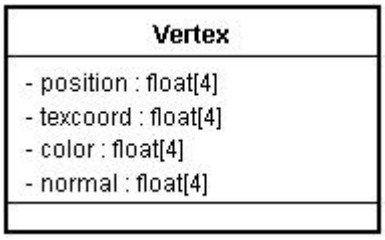
3.6.1 VertexBuffer

VertexBuffer 可以理解成一个 GPU 上的数组，数组的元素是顶点数据，VertexShader 所需的数据就会从这里取。VertexBuffer 用最直观的方式存储顶点数据，顶点和 VertexBuffer 的元素一一对应。

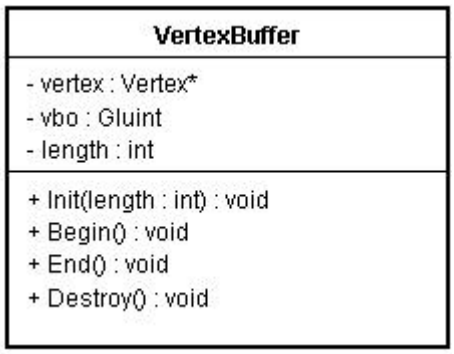
VertexBuffer 在引擎中充当组件的角色，下面是 VertexBuffer 和资源管理模块的关系：



下面给出 VertexBuffer 的定义，VertexBuffer 可以理解成顶点数据的集合，首先是顶点数据的定义：



VertexBuffer 的定义：

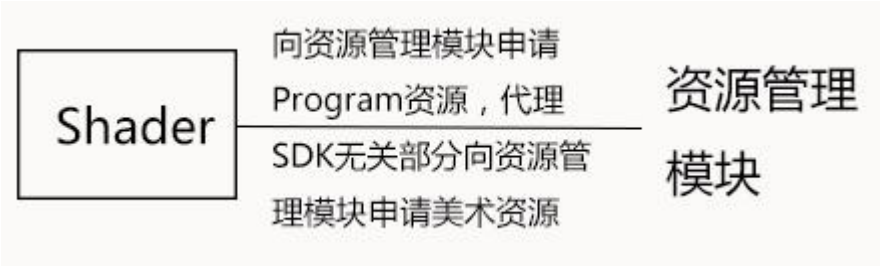


VertexBuffer 持有一个顶点数组 Vertex *m_Vertex 来在 CPU 存储实际的顶点数据。然后由 Begin()方法绑定该 VertexBuffer 为当前显存并把最新的内存数据更新到显存，Destroy 函数会清理 VertexBuffer 所申请的资源。

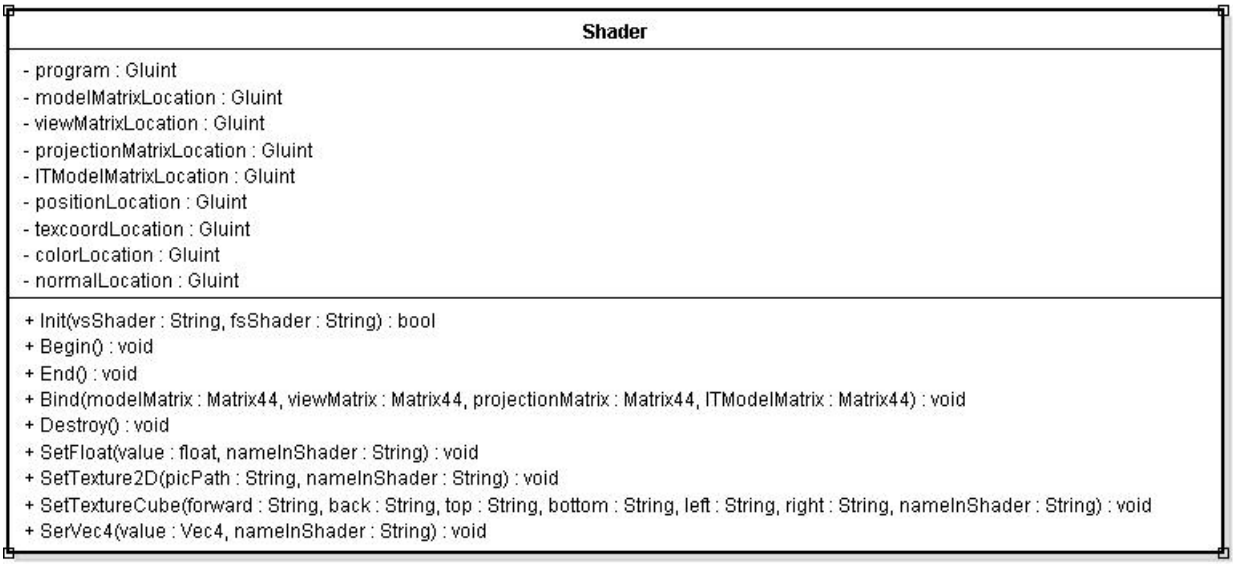
3.6.2 Shader

Shader 就是所谓的 GPU 程序，在该引擎中一个完整的 Shader=VertexShader+FragmentShader，一个 Shader 会到当前指定的缓冲区（显存）去读顶点数据，然后进行绘制。

Shader 在引擎中也是充当组件的角色，复杂管理 GPU 上的程序，给出 Shader 和资源管理模块的关系：



Shader 定义：



Shader 类最关键的设计是提供了无限的适配性，通过 SetXXX 接口设置这些值然后每次

绘制前引擎会为开发者设置值的大小。例如 void SetTexture2D 函数会将值存储，作为 Shader 的属性会在每次绘制前被传递。通过上述方式可以传递任意格式的任意变量名的值到 GPU，大大拓展了 Shader 通用性。

每次绘制前（Bind 函数中）会将 map 中最新的值取出，传递给 GPU。在 Bind 函数中，Shader 类会根据各个变量的 Location 去更新各项数据。例如 MVP 三大矩阵，以及 Position、Color、Normal、Texcoord 信息。

3.7 底层渲染——SDK 无关部分

SDK 无关部分定义逻辑概念上的游戏物体，它可以是模型、可以是地面、也可以是粒子系统特效。

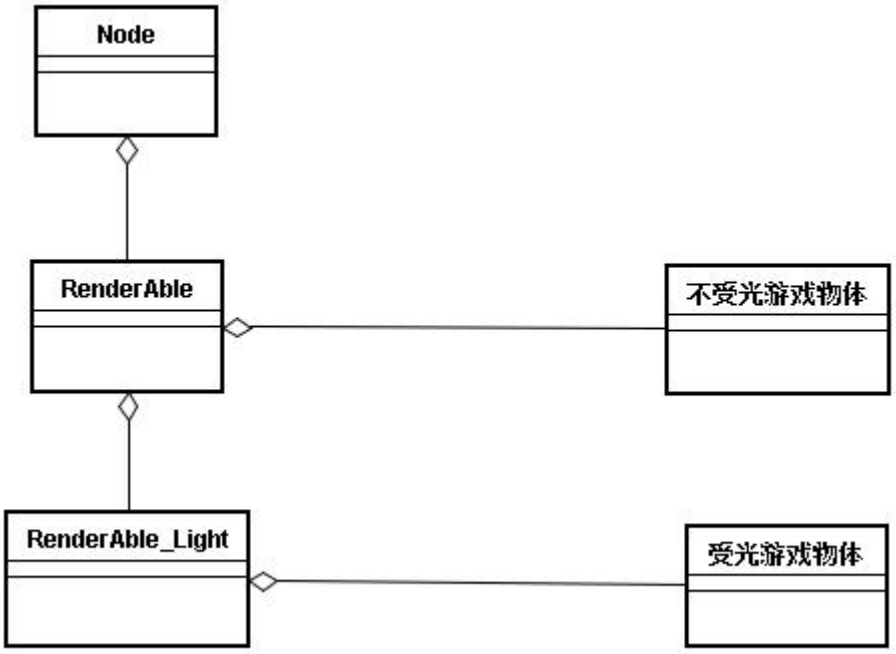
SDK 无关部分职能如下：

①关心当前物体顶点数据及其纹理的获取。

②物体的逻辑更新。

SDK 无关部分包括但不限于摄像机、模型、天空盒、地面、粒子系统等等通过对顶点数据进行逻辑封装产生的对象等。它们各不相同，但又有一些能够被抽象出来的共性。该模块的设计是将共性作为基类，下方逻辑游戏物体可以进行无限的延伸。

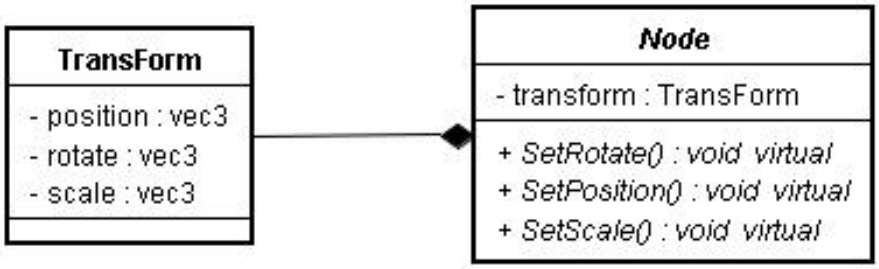
①SDK 无关部分的继承关系是这样的：



也就是说，引擎对不渲染物体、不收光游戏物体、受光游戏物体做了分类，它们都分别有一个基类，通过继承这些基类获得不同的属性，然后再通过自身初始化、更新逻辑获得不同特性。

3.7.1 基类

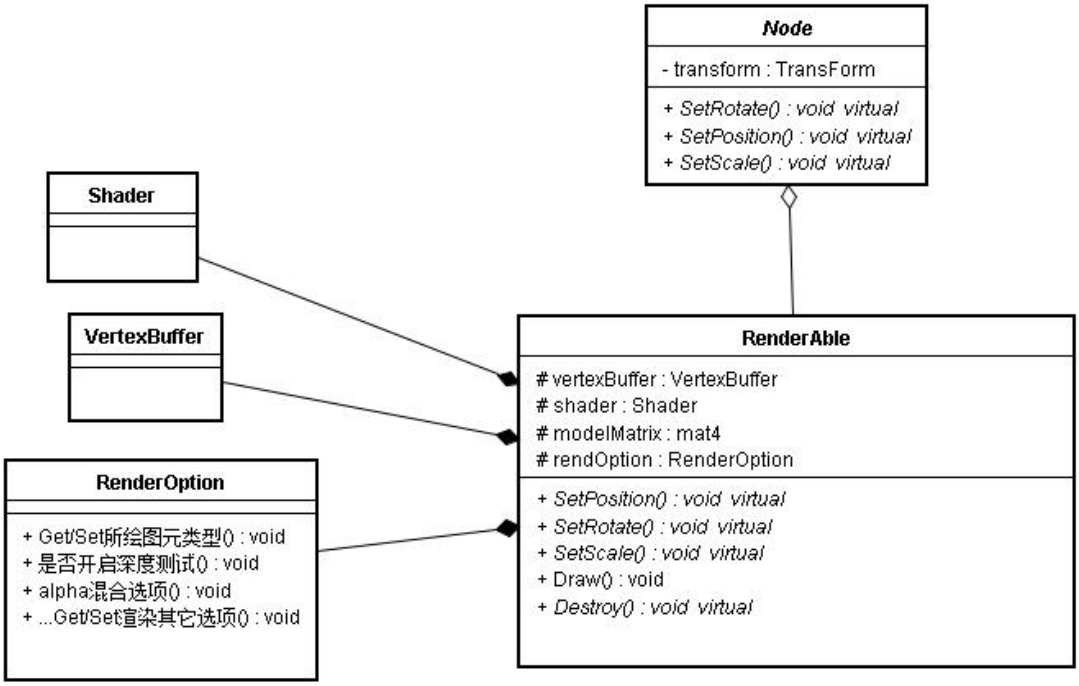
首先是所有游戏物体的一个共同的基类 Node。



可以看见基类里面有一个位置信息，因为无论是摄像机、光源而或是模型，都会有一个位置信息。

3.7.2 渲染基类

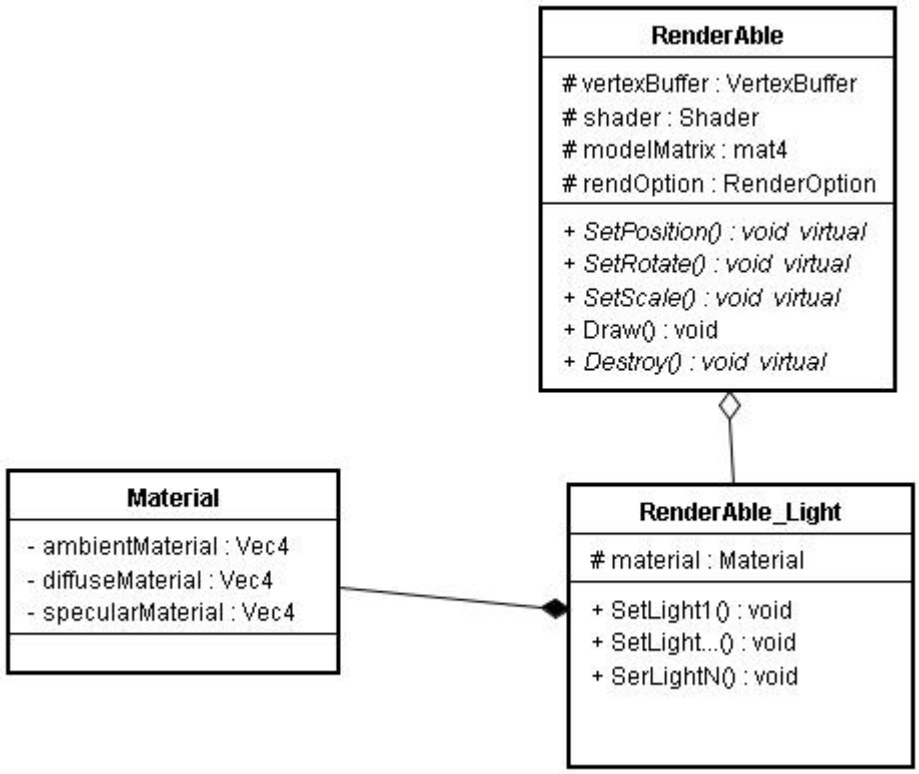
要被渲染的游戏物体需要继承先前提到的 RenderAble 类，下面给出该类的定义。



RenderAble 类提供了足够多的抽象,它的职能是提供足够多的信息供 RenderList 使用，RenderOption 是传递给 RenderList 的渲染选项，它随时可以被扩充以适配更复杂的情况。主要信息是 VertexBuffer ，Shader ，ModelMatrix,RenderOption 这 4 个绘制必备信息。

其中 SetPos、SetRotate、SetScale 对 Node 提供的虚函数进行了重写，因为在更新位置信息的同时，RenderAble 需要同步更新 modelMatrix 的值。Draw 函数负责将顶点数据传递到当前 Scene 的渲染列表，Destroy 函数会清理掉 Shader、VertexBuffer 所申请的资源。

3.7.3 受光对象基类



该类继承 RenderAble，也就是说在 RenderAble 的基础上需要接受常规光照的类拓展了一些新功能。需要接受接受常规光照的类则继承 RenderAble_Light 类，RenderAble_Light 类提供了默认的 Set 材质和灯光的实现。也就是说，继承该类的类可以接受任意盏灯，只需要提供灯的变量在 Shader 中的名称即可。

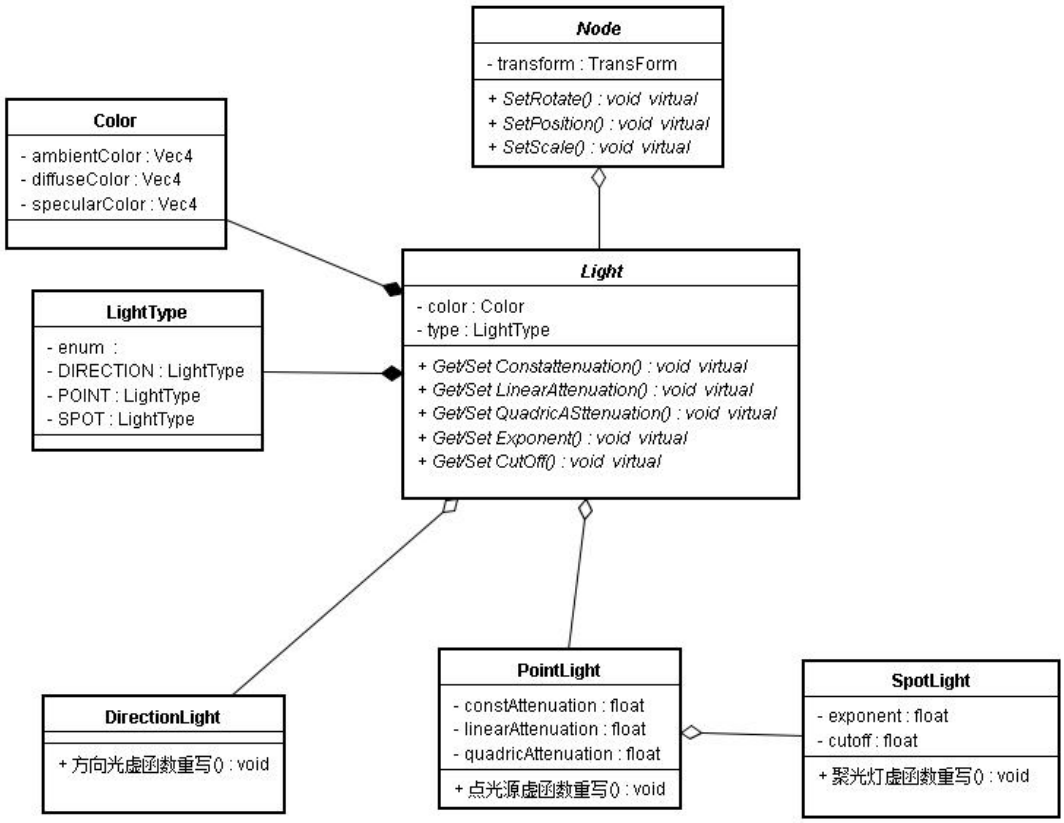
3.7.4 灯光

引擎中封装了 3 种灯光，通过枚举值决定实际类型，而实际处理灯光效果的代码在 Shader 中出现，灯类只负责存储数据，并将其数据的值并告诉 Shader，而不会进行任何渲染。

也就是说，其数据流如下：

灯光类的内存空间存储了这个灯的一些信息，然后开发者将该灯光信息传递给 RenderAble_Light 的子类，其子类就会去读取该灯光的信息，然后传递到 Shader 变量参与绘制。

灯光系统类设计如下：



首先是灯光基类,灯光基类只存储类型和颜色信息,不需要重写 Node 的位置函数,Light 类的类型明确指明和需要哪些参数。

然后是具体类的 3 种光，首先是方向光。方向光只需要告诉 Shader 它的类型是方向光即可。光源会出现衰减，但没有光的半角信息（相关函数被设置为私有）。聚光灯有灯的所有参数，会进行点光源式的衰减，还有光照射的距离限制，需要实现 Light 的全部函数。

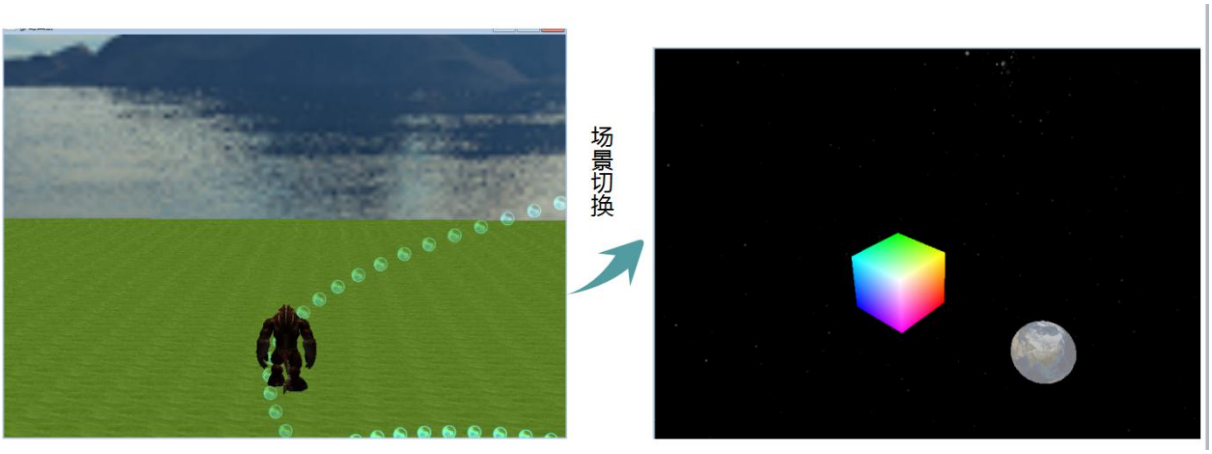
第四章 编码实现

4.1 总体效果预览

4.1.1 资源管理类

资源管理类正常运行情况——在初始化，销毁场景时能正常申请和清理掉已经申请但不需要的资源。

4.1.2 场景管理

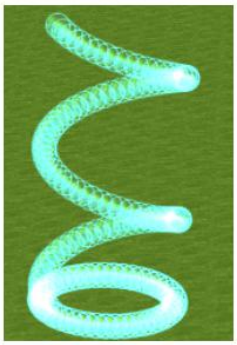


通过函数 SceneManager::LoadScene(string)，一句话使得场景进行切换，切换时会析构之前场景的所有资源。

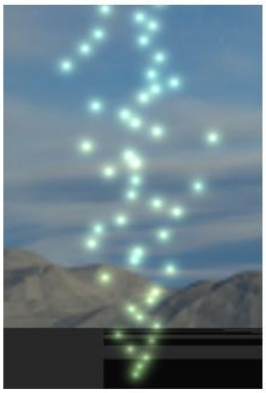
4.1.3 底层渲染（举例）

模型渲染

方向光从上方往下打在地球模型上



粒子系统——旋涡状粒子



粒子系统——萤火虫

4.2 场景管理

前面说到 SceneManager 负责为场景内的物体渲染提供框架，下面给出框架的具体实现及其结果。

4.2.1 SceneManager

SceneManager 有加载和卸载场景的能力，其管理是通过一个装载 Scene 地址的 std 的容器 std::map<string, Scene*> SceneManager::m_mScene 实现，在注册的时候提供场景名称，然后就可以在 SceneManager 中随时加载和卸载场景。等于是只要注册后，场景将永远

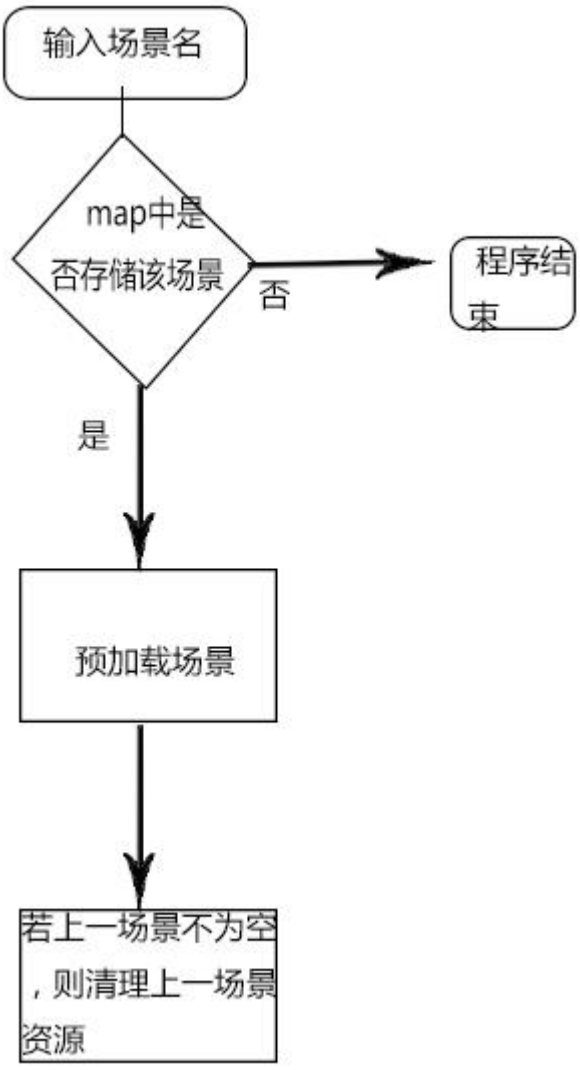
存在于游戏中，但是内存和显存中只会存放当前场景的大量信息，其它场景只维护少量信息保证其可以被再次加载。

回顾一下成员变量定义，随后分析其函数实现思想。

```
static std::map<string, Scene*> m_mScene;
static Scene* m_CurScene;
```

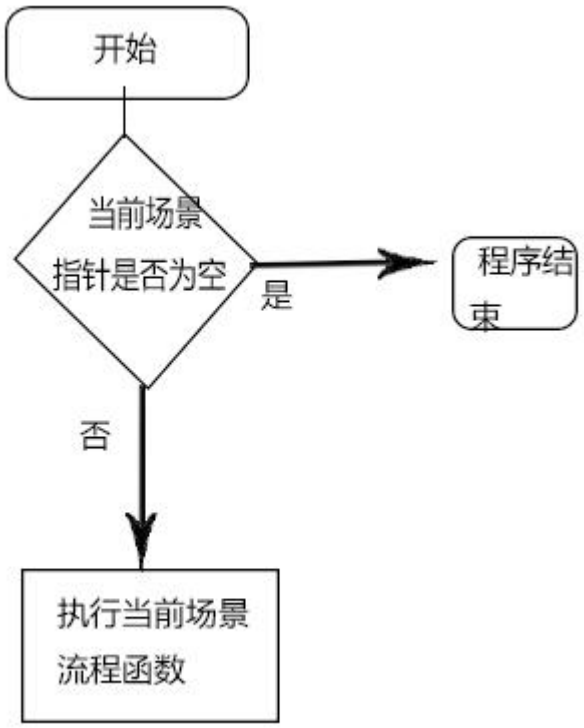
①加载/卸载场景

注册场景后，该场景的指针只会被简单地添加到 map 中，而该场景是什么样的 SceneManager 并不关心。在 LoadScene 方法中，首先将当前 Scene 卸载，然后会去加载新的 Scene 并让它成为新的当前 Scene。



②游戏循环实现如下

在 SceneManager 游戏循环中，首先会更新当前帧所用时间，然后会调用 Scene 基类的系统游戏循环函数和 Scene 派生类的游戏循环函数，执行当前场景的游戏循环。需要注意的是 SceneManager 的游戏循环包含局部的游戏循环（当前场景的循环）和全局的游戏循环（例如时间戳）。



4.2.2 Scene

Scene 是所有场景的基类，它需要做的事情是初始化一个场景最基本的东西，并定义流程的虚函数让子类重写。

成员变量回顾：

```
Camera_1st *m_MainCamera;
RenderList m_3DRendList;
```

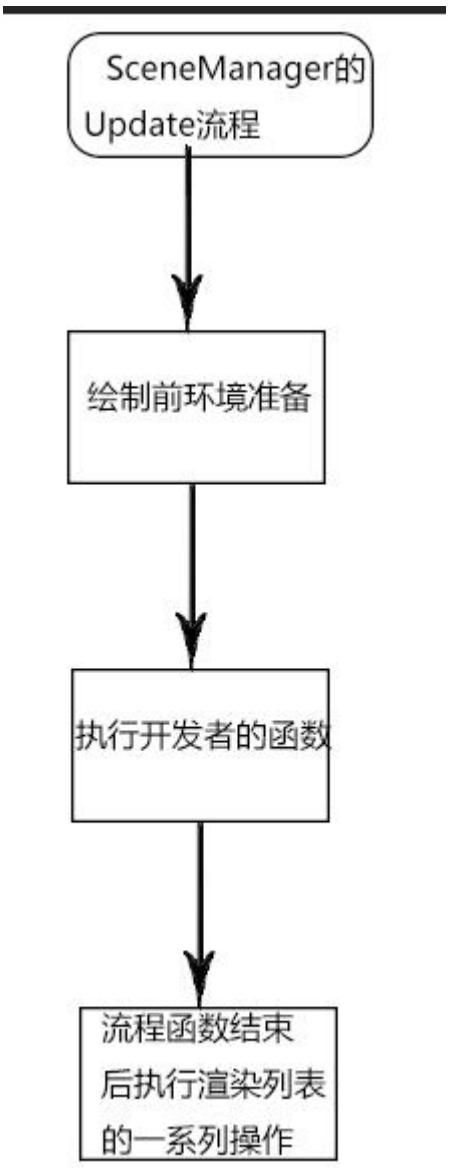
①System 初始化函数

场景开始时只完成一件事情：给 RenderList 提供默认的摄像机，使得渲染能够正常进行。

②System 游戏循环

系统的游戏循环主要完成以下几件事情：

- 在场景派生类开始绘制前把“画板”擦干净。
- 调用开发者重写的 virtual 流程函数。
- 在场景派生类绘制结束后执行渲染列表绘制函数。



③System 销毁场景

只需要清理场景基类申请的资源，并调用场景派生类重写的清理资源函数。

④其它非 System 函数（虚函数，让开发者重写）

类中所有虚函数都是提供给开发者的函数，全都是虚的空函数，供开发者重写实现自己场景的逻辑。

4.2.3 渲染列表

RenderList 负责绘制一个相机所看到的的所有游戏物体。对它做一些处理后将顶点数据统一传到 GPU。

成员变量回顾

```
Camera_1st *m_pMainCamera;//此渲染列表所用的摄像机
```

```
std::vector<RenderAble *> m_RendList;
std::vector<RenderDomain> m_DomainRenderList;
```

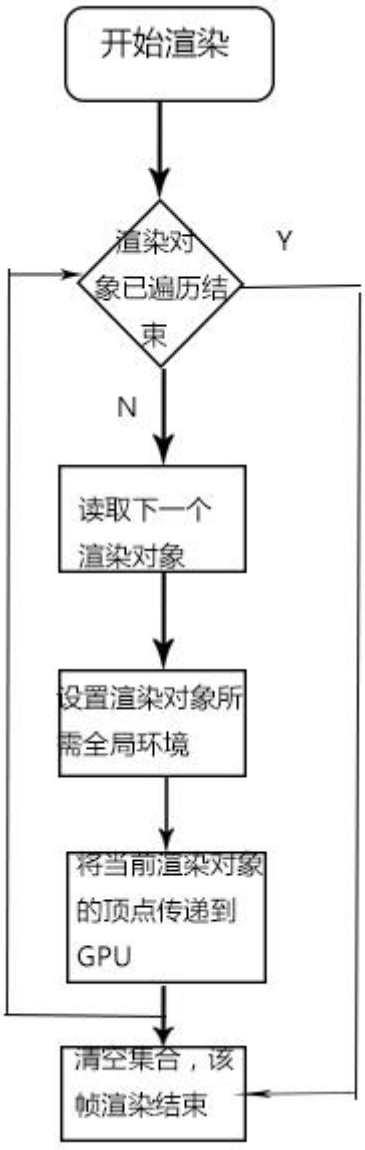
①渲染前期处理

- 裁剪：裁剪功能是 OpenGL 的一个全局状态，RenderList 中通过 SceneManager 去设置这个状态，设置状态的参数从该 RenderList 的参数中取出，该全局状态同样由 SceneManager 维护。开启裁剪后，在摄像机视景体以外的游戏物体都不会被绘制，降低了巨大的性能消耗。

②绘制

RenderList 绘制函数需要完成的工作：

- 遍历两组渲染列表
- 绘制每一个渲染对象时，根据对象成员中的渲染选项设置当前渲染参数、全局状态，并组织起渲染指令，一条条传输到 GPU。
- 清空两组渲染列表，等待下一次绘制。



4.2.4 摄像机

摄像机看到的即玩家看到的，故摄像机可以理解为玩家的眼睛，摄像机需要做的事情就是定义：玩家能看到哪些东西？玩家不能看到哪些东西？

成员变量回顾：

```
glm::vec3 m_Position;//摄像机位置
glm::vec3 m_ViewCenter;//视点
glm::vec3 m_Up;//Up 向量

glm::mat4 m_ViewMatrix;
glm::mat4 m_ProjectionMatrix;

float m_Radius = 50.0f;//视野范围
float m_ViewportWidget = 800;//视口大小
float m_ViewportHeight = 600;//视口大小
float m_ViewportXStart = 0;
float m_ViewportYStart = 0;
float m_Near = 0.1f;//最近距离
float m_Far = 1000;//最远距离
```

①构造

摄像机的构造只需要为它的成员变量设置初始的视口，并更新视口矩阵和投影矩阵。

②摄像机视口移动（上下左右）

让摄像机上下左右移动，只需要将摄像机的位置移动并且将视点位置移动即可，可以理解为人在平行移动，眼睛关注的视点也在平行移动。

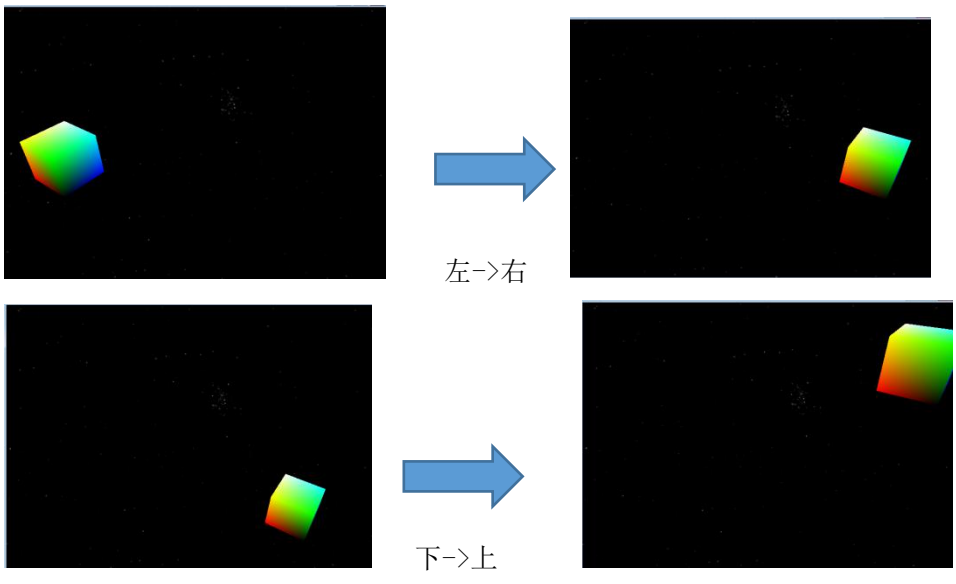
算法伪代码:

偏移量=移动方向*速度*时间

摄像机位置+=偏移量

摄像机视点+=偏移量

效果如下：



③视口视点移动（远近）

摄像机远近移动即人位置不变，但走进所观察的物体。算法即先计算出视点-摄像机的方向向量，然后摄像机的位置随着方向向量移动，同时视点位置做相对运动，即可实现摄像机的远近变换。

算法伪代码：

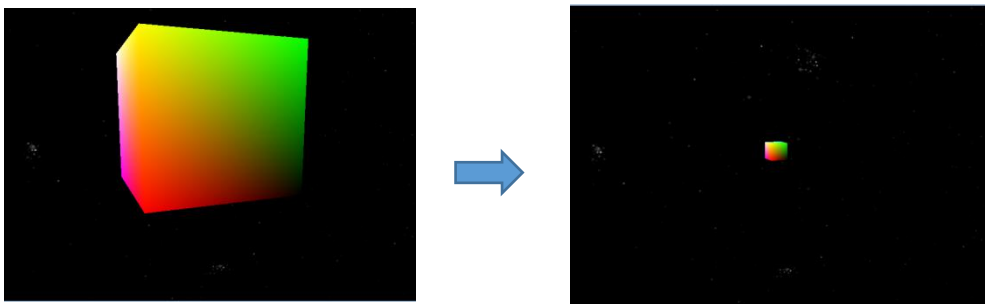
向前的方向向量=视点坐标-摄像机坐标

偏移量=向前方向*速度*时间

摄像机位置+=偏移量

摄像机视点+=偏移量

效果如下：



④摄像机旋转

摄像机旋转分为绕 X 轴进行的上下旋转，和绕 Y 轴进行的左右旋转。引擎中的实现方式是用一个抽象的函数 RotateView(float 角度，vec3 轴)处理摄像机绕任意轴的旋转，旋转的思想基于以下公式矩阵：

$$\begin{bmatrix} C+A_x^2(1-C) & A_xA_y(1-C)-A_zS & A_xA_z(1-C)+A_yS \\ A_xA_y(1-C)+A_zS & C+A_y^2(1-C) & A_yA_z(1-C)-A_xS \\ A_yA_z(1-C)-A_yS & A_yA_z(1-C)+A_xS & C+A_z^2(1-C) \end{bmatrix}$$

其中 C 表示角度的 cos 值，S 表示角度的 sin 值，A 和 A 的下标表示旋转轴的分量。

有了该公式矩阵，给出摄像机绕任意轴旋转的算法如下：

视线方向=摄像机视点-摄像机位置

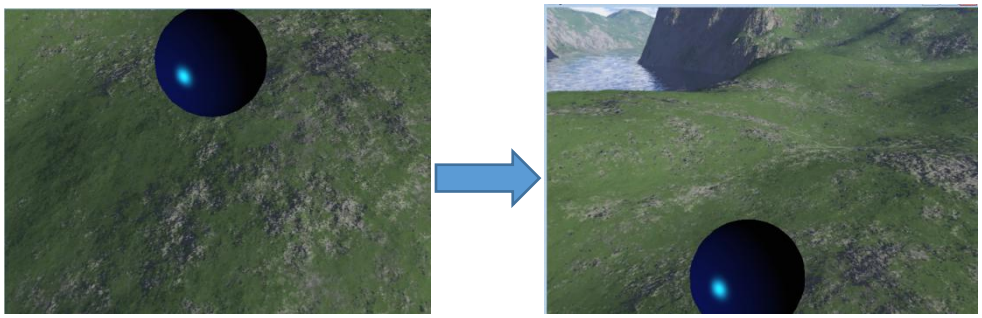
新的视点的 x 分量=点乘（矩阵第一行，视线方向）

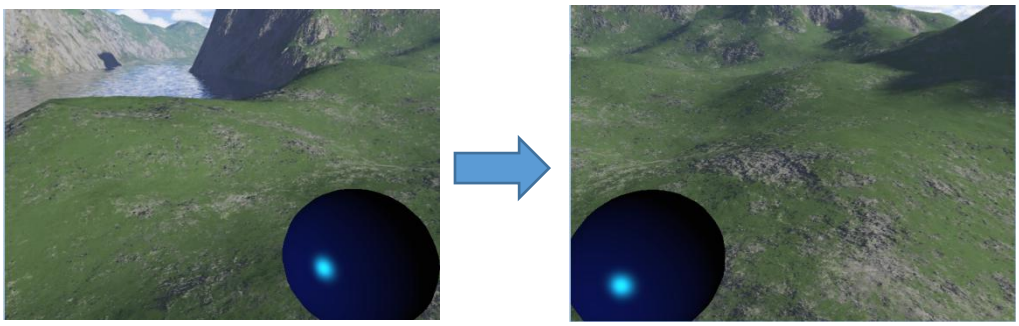
新的视点的 y 分量=点乘（矩阵第二行，视线方向）

新的视点的 z 分量=点乘（矩阵第三行，视线方向）

摄像机视点=摄像机位置+新的视点

效果如下：





4.3 资源管理

资源管理为底层渲染器提供资源。实现效果如下：

下面给出具体实现。

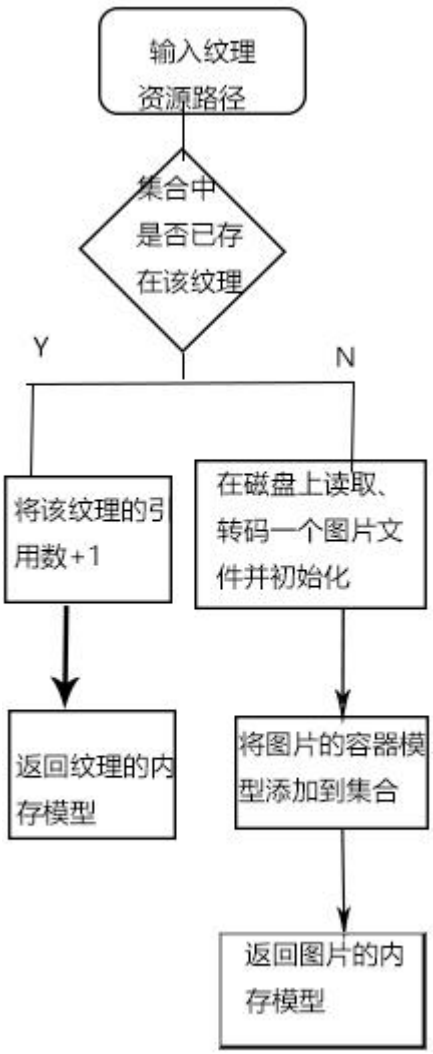
4.3.1 纹理类型管理

所有在 GPU 中申请的纹理都会以 unsigned int 类型返回，资源管理器在没有任何游戏物体在引用该资源的时候会将资源析构。

相关容器：

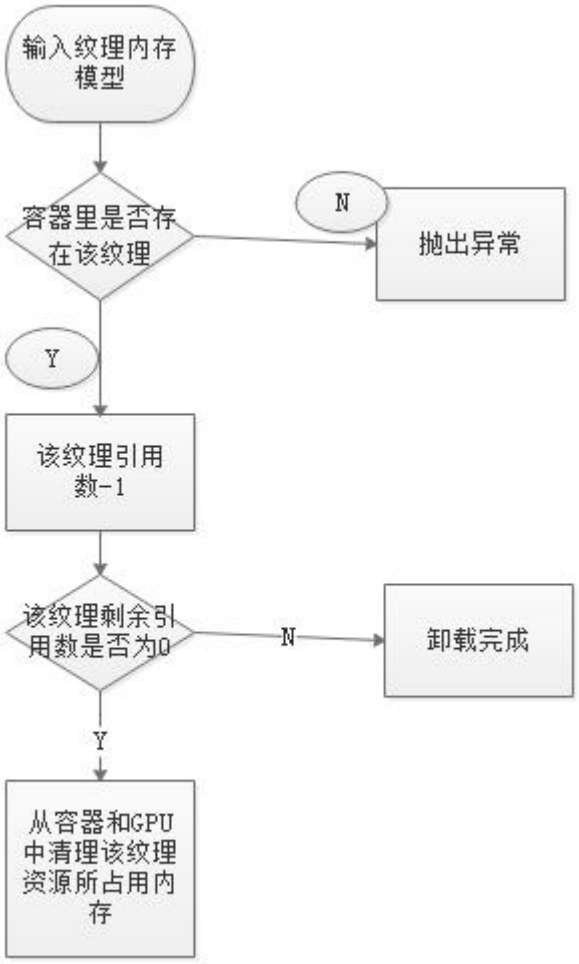
```
static std::map<string, Texture> m_mTexture;
static std::map<string, Texture> m_mTextureCube;
```

①获取纹理资源



存储纹理类型的容器维护资源所在文件路径、纹理引用数、纹理标识符的信息，当引用数为 0 时资源会从 GPU 中卸载。这样有几个好处：不需要反复向 GPU 申请资源，节省了显存和 CPU->GPU 通信的时间代价；将申请资源的函数封装在 ResourceManager，外界不需要调用相关函数，实现申请资源的函数和使用资源的函数解耦合。

②析构纹理资源



卸载函数采用引用计数方式，从容器中找到该纹理并将其引用数减少，当引用数为 0 时从集合和 GPU 卸载该纹理。

③从磁盘上加载图片简介

从磁盘加载图片使用了外部库 FreeImage，简单来说流程就是，加载图片的字节数据到内存->解码图片的字节数据->在 GPU 申请一块槽来存放该纹理->为纹理设置参数->将纹理的字节数据传输到显存->清理内存上的的纹理字节数据。

可以看到这个过程是非常耗时的，从磁盘到内存，从内存到 GPU，如果大量进行纹理读取将非常影响实时引擎效率。故该函数只会在纹理不存在于显存的时候被调用。

④立方体纹理简介

立方体纹理需要 6 个面来构造，创建资源的方式和 2D 纹理类似，先在在 GPU 申请一块槽来存放该纹理->为纹理设置参数，然后重复执行 6 次以下步骤实现磁盘上的图片传出到 GPU：先加载图片的字节数据到内存->解码图片的字节数据->将纹理的字节数据传输到显存正确的位置->清理内存上的的纹理字节数据。需要注意的是，传输图片的字节数据到显存的时候需要设置正确的参数，才能把图片正地放到立方体某个面上。

4.3.2 GPU 程序管理

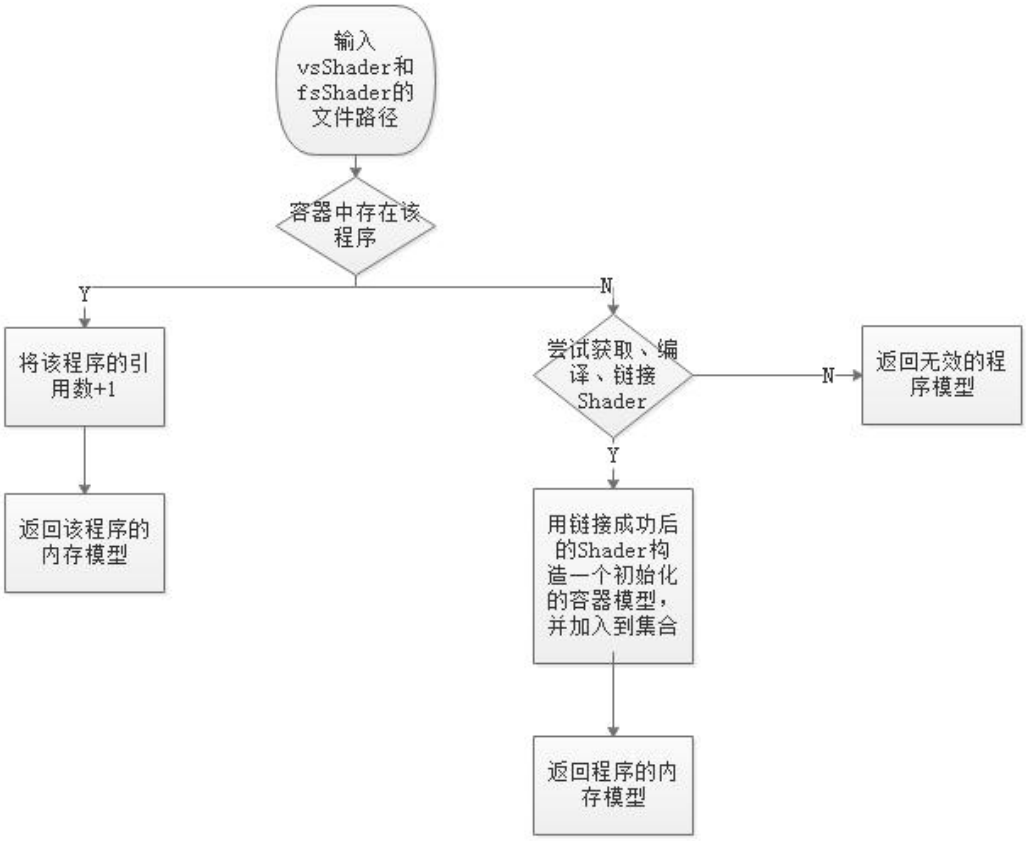
在引擎中，GPU 程序=VertexShader+FragmentShader。ResourceManager 提供了获取程序的方法以及内部编译、链接程序的指令集。

相关容器：

```
static std::map<string, Program> m_mProgram;
```

①获取、编译 GPU 程序

获取函数首先检测该 GPU 程序是否已经送到 GPU 编译完成了，若内存中有该 GPU 程序的信息则直接把该 GPU 程序在 GPU 的标识符返回；如果没有编译过该 Shader 则从磁盘分别将顶点 Shader 和片元 Shader 读取、编译、链接并拿到 GPU 返回的标识符储存起来。



②卸载 GPU 程序

卸载函数同样采用引用计数方式，从容器中找到该 Shader 并将其引用数减少，当引用数为 0 时卸载该 Shader。

③编译、链接多个 Shader 成 GPU 程序简介

编译 Shader 就是把对应类型的 Shader 送到 GPU 去编译，编译不通过则返回报错信息供调试。链接就是把编译好的 Shader 绑定到 GPU 程序上，并将多个 Shader 链接成一个程序，类似 C 语言的链接。

4.3.3 OBJ 模型管理

引擎使用了.obj 格式的模型，obj 模型的定义包括顶点法线、纹理坐标、位置信息以及绘制指令信息。

相关容器：

```
static std::map<string, ObjModel> m_mModel;
```

①OBJ 模型的获取、卸载和缓存方式。

模型资源的管理和其它资源相同，都是通过引用计数的形式。但是模型资源是通过缓存实际的绘制指令来管理的。

obj 模型文件中有 4 类数据头需要关注：v 开始的 vt 纹理坐标信息、vn 法线信息、v 位置信息以及 f 开头的绘制指令。在初次加载模型时会将这些信息解释后写入到用作输出参数的 vbo，然后再将模型数据缓存到 map。缓存的意义是用一个模型大小的内存来以避免频繁 IO 所消耗的时间。

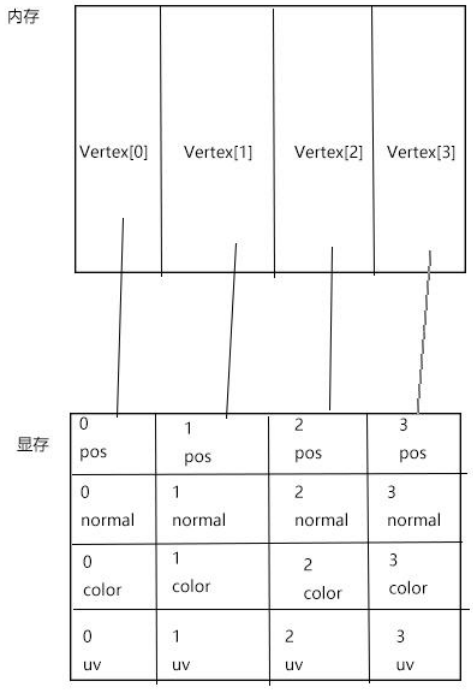
4.4 底层渲染——SDK 相关部分

SDK 相关部分包括抽象出来的显存、Shader，它们是作为组件被组合在 RenderAble 中的，复杂和 OpenGL SDK 的交互。下面一一介绍。

4.4.1 VertexBuffer

VertexBuffer 表示顶点数据集，是 GPU 一块显存的抽象。在绘制时指定自己为当前的 VertexBuffer，则接下来 Shader 会到这块区域去读 attribute 标志的数据。

其关联模式是这样的：



成员变量回顾：

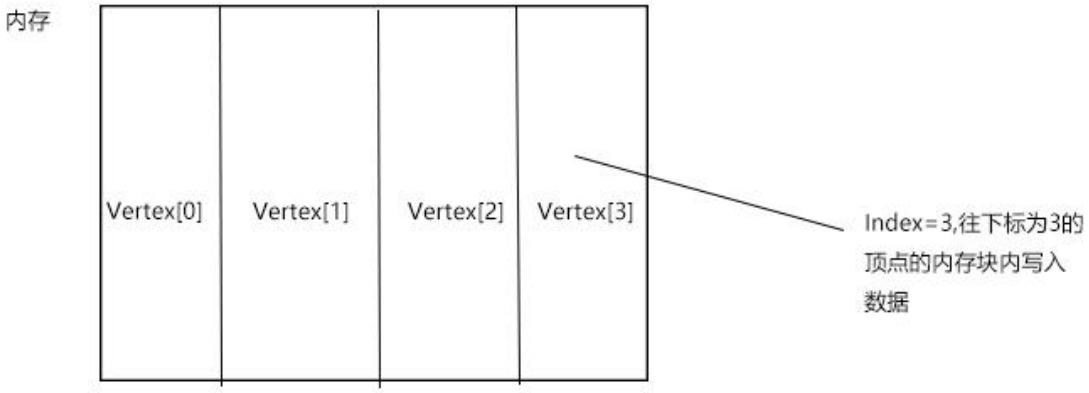
```
GLuint m_Vbo = _INVALID_ID_;
Vertex *m_Vertex;
int m_Lenth = INVALID;
bool m_IsInit = false;
```

①初始化

初始化的时候会在内存开辟一块内存、一块显存用于存储顶点数据，内存端用于供开发者进行逻辑设置和显存端用于 Shader 去读数据来进行绘制。在 Init 函数被执行之后内存就开辟好了，可以往里面写数据了。

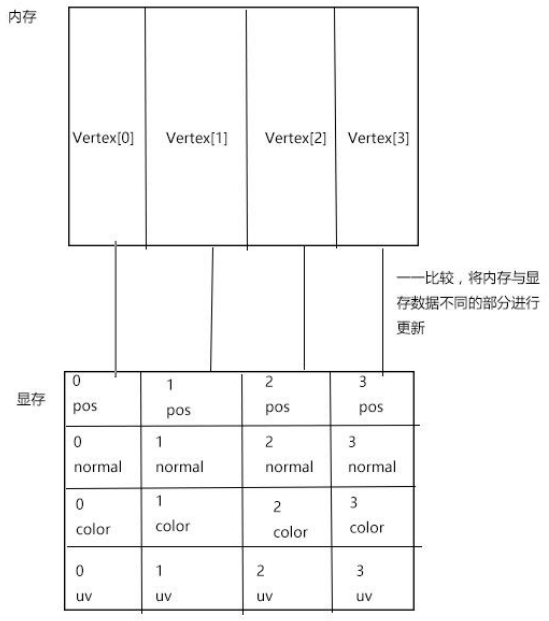
②设置顶点数据

SetNormal 函数往 index 对应的内存块写入数据，此时该 index 对应的顶点数据已经变化了。需要注意的是，在设置顶点数据的时候，改变的是内存的数据，也就是说：



③绘制前更新数据。

绘制前需要将内存中最新的顶点数据更新到显存对应位置，这里采用增量更新的形式，只将有变动的部分更新到显存，其它不变。这样就能实现将显存中最新的顶点数据更新到内存。



4.4.2 Shader

Shader 在这里指的是完整的 GPU 程序，即 VertexShader+FragmentShader，主要涉及的函数封装就是对于 Shader 变量的设置。

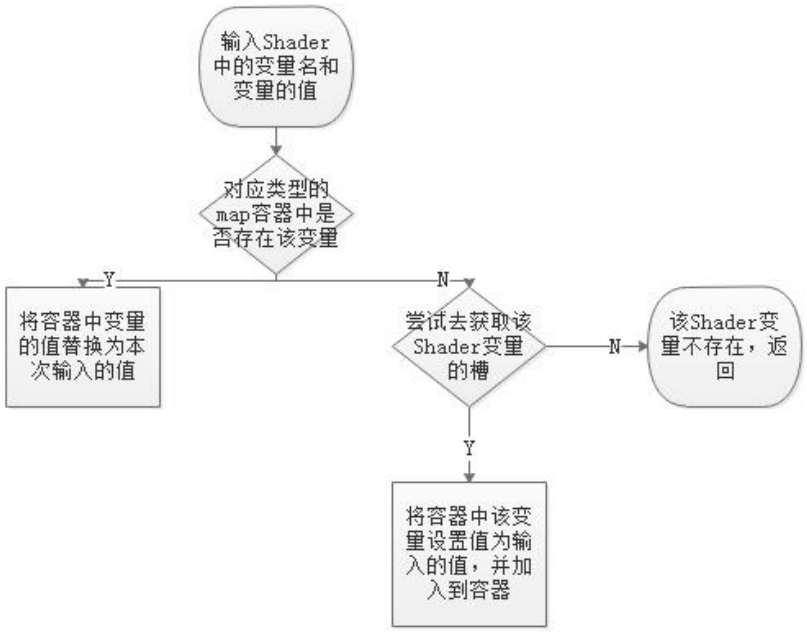
①Shader 初始化

Shader 初始化需要提供顶点着色器和片元着色器的名称，然后会让资源管理器去磁盘上读取该 Shader，接着就是根据 Shader 名字获取一些必备的 attribute 变量和 Uniform 变量的 Location。这些 Shader 名字也会提供接口让外部设置。

②Shader 变量设置及使用

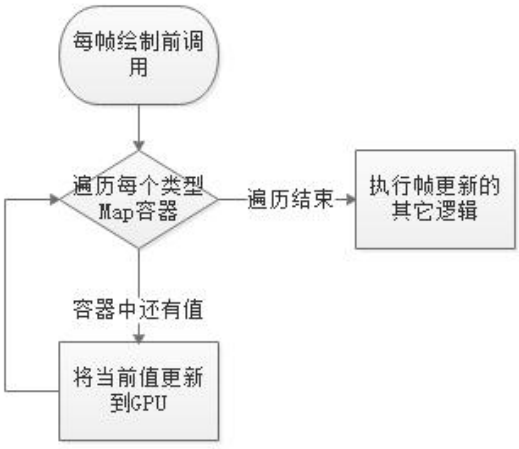
只能设置一些固定的变量是不够的，还需要有更加兼容的接口，能根据类型设置变量，才能满足不同 Shader 的需求。为此，引擎提供了两个接口实现这个功能。我们来看这两个功能的实现方式。

变量的设置实现：



在 Shader 类实现中，为其设计了若干个 map 容器用来存储变量在内存，而 set 着色器的变量，实际上就是将变量的内存模型存储到 map，以供后续向 GPU 传递变量的值。具体做法就是看 map 中是否已经有该变量了->有的话就直接更新它的值，没有的话就尝试去获取 Location 并设置值。

变量的更新实现：



数据的更新发生在在每帧绘制之前，Shader 会去查看容器中是否由值，如果有那么就去更新它的值为 map 中最新的值。

4.5 底层渲染——SDK 无关部分

SDK 无关部分就是指引擎内置的游戏物体，经过分层后它不需要过多地关心 OpenGL 的细节，而专注于描述这个可绘制物体的特点。

4.5.1 抽象的光照算法

要计算光照，首先要知道：我们所看见的颜色，实际上是光照射在物体表面，物体表面反射出来的颜色进入到我们眼睛，我们才能看见颜色。再结合第二章所讲述的三维世界光照模型，引擎实现了对方向光、点光源、聚光灯颜色计算的算法。

①方向光颜色计算函数

首先是方向光，方向光有一个照射方向，且不需要计算衰减。所以只需要计算方向光源对于该物体材质的影响就能实现正确的着色。具体计算方式就是实现基于第二章提出的理论基础：环境光反射颜色=光源环境光颜色*材质对于环境光颜色的吸收；漫反射颜色=光源漫反射颜色*材质对于漫反射颜色的吸收*法线和光照射方向的向量内积；镜面反射颜色=光源镜面反射颜色*材质对于镜面反射颜色的吸收*眼睛方向与光反射方向的 N 次幂。

给出方向光计算的算法：

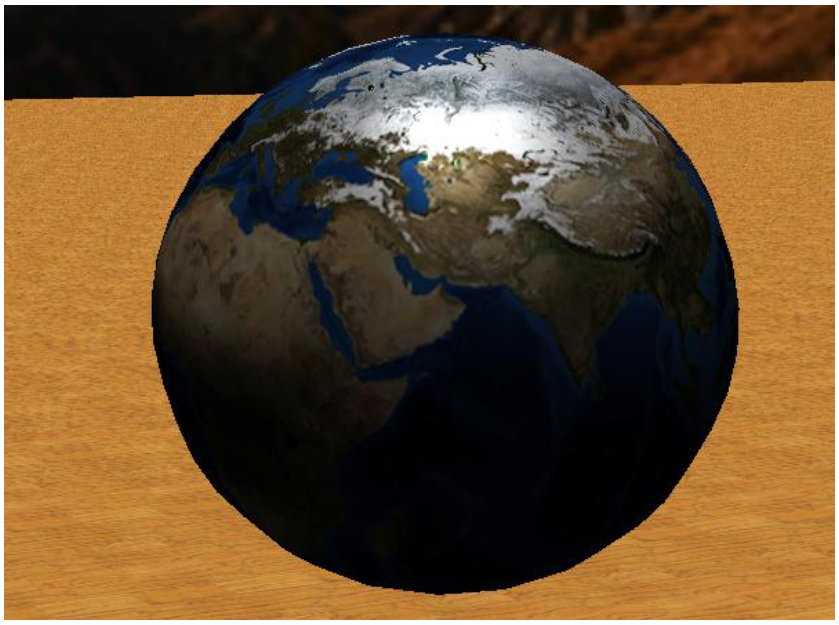
环境光着色=环境光颜色*物体环境光材质

漫反射着色=漫反射颜色*物体漫反射材质*点乘（顶点法线，光的方向）

镜面反射颜色=镜面反射颜色*物体镜面反射材质*点乘（视线方向，反射光方向）的 N 此幂

最终颜色=环境光着色+漫反射着色*镜面反射着色

方向光着色效果：



方向光从正上方打在地球模型上（太阳光）

②点光源颜色计算

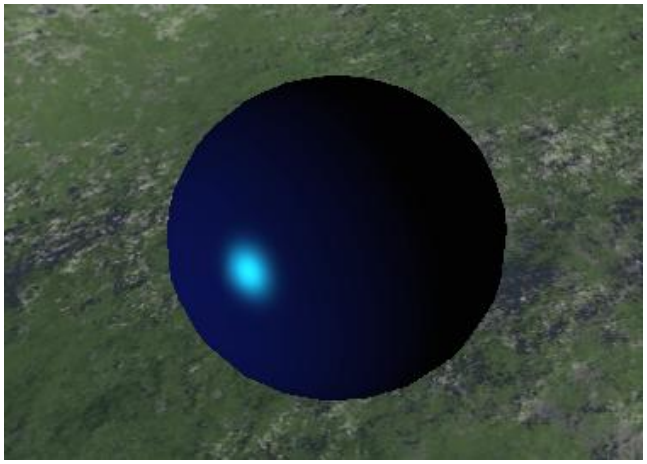
点光源和方向光逻辑类似，这里只点出它们不同的地方——漫反射光会衰减。引擎中实现的点光源只在漫反射光实现了光的衰减效果，实际上在环境光、镜面反射也进行衰减也可以，但是实际渲染出来的效果并不能使人满意。衰减方式就是去计算一个衰减系数，再原先计算的颜色的基础上乘上一个衰减系数，就得到点光源的颜色。

基于方向光的算法，给出点光源的算法：

衰减因子=1/(常衰减因子+线性衰减因子*着色点和光源的距离+平方衰减因子*着色点和光源的距离的平方）

最终颜色=（环境光颜色+漫反射颜色+镜面反射颜色）*衰减因子。

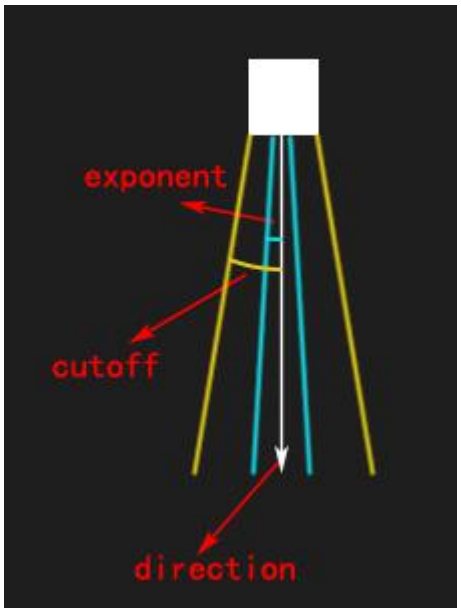
着色效果：



点光源打在小球上形成光斑

③聚光灯颜色计算函数

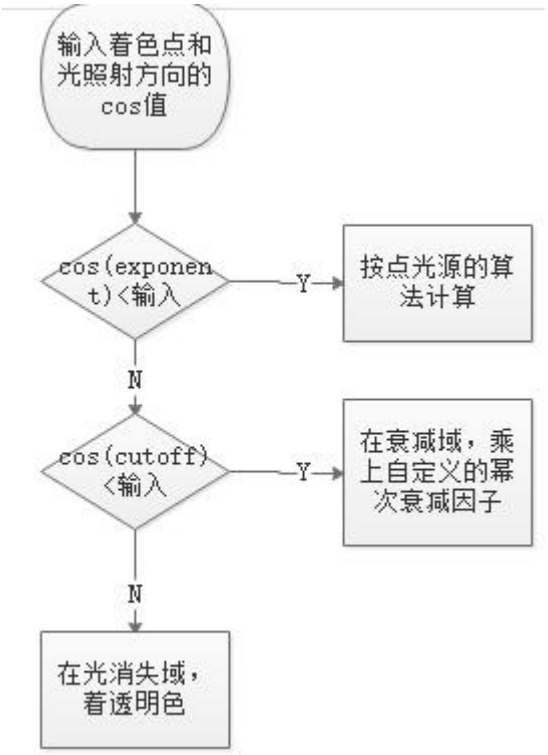
聚光灯颜色的计算是最为复杂的。它分为三个区域：光强不变域、光强衰减域和光强消失域。



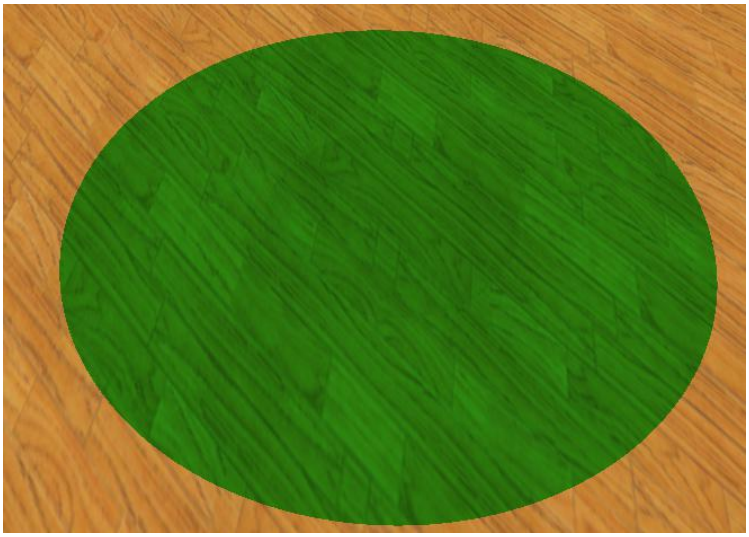
光强不变域的颜色也就是和漫反射光是一样的;光强衰减域则需要计算聚光灯的衰减系数,这里使用了简化的计算方法:使用物体和光照射方向的夹角余弦取 N 此幂做为衰减系数,得到衰减系数后用衰减系数乘上漫反射光的颜色即可；如果在光强消失域则不显示颜色。

基于点光源的算法，给出聚光灯的算法：

1、计算出着色点到光源的延长线和 direction 方向的夹角余弦。



聚光灯着色效果：

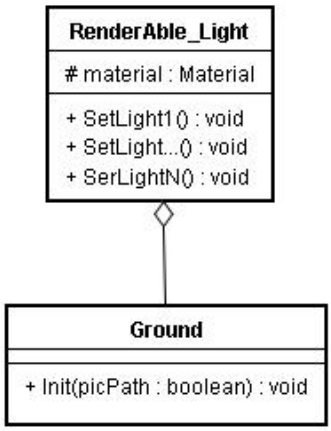


聚光灯打在地面上形成 3 个区域

4.5.3 地面

地面实现就是一张巨大的贴图，法线向上，纹理坐标对应四个顶点。它继承自 RenderAble_Light 且仅需要实现初始化方法以及引擎内置的地面 Shader。

地面类定义：



地面绘制效果：

①地面初始化

地面初始化函数对 VertexBuffer 和 Shader 执行初始化，并设置渲染选项和材质。实际上地面所需的属性在 RenderAble 全部覆盖了，故不需要添加额外的成员变量。

地面初始化最关键的部分是设置地面的顶点信息。引擎中的地面只需要 4 个顶点组成一个法线向上的矩形。

②地面着色

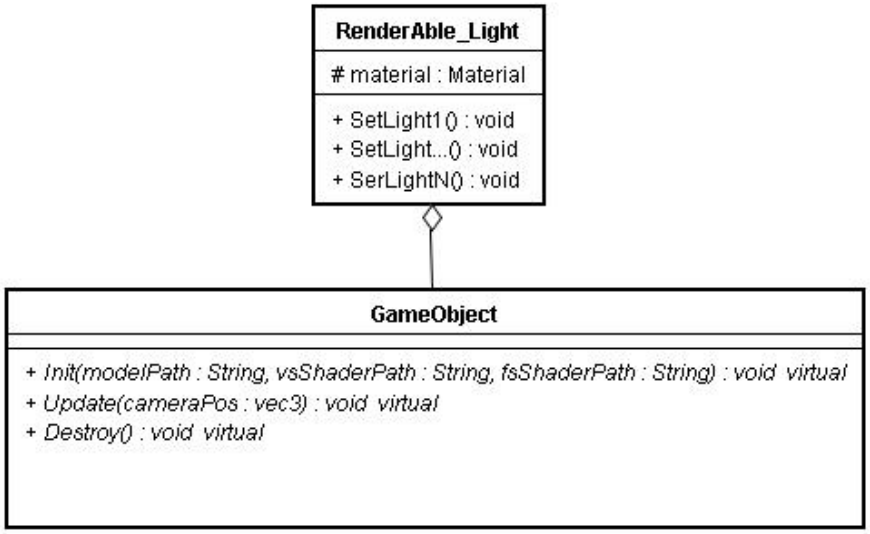
地面的顶点着色器只处理顶点位置信息，颜色、法线、纹理坐标信息只做必要处理后就扔给片元着色器，让片元着色器负责颜色。

我们在地面的片元着色器处理颜色信息。光照部分调用了 4.5.1 讲述的光照算法拿到每个像素的光照颜色，再用 GLSL 内置的贴纹理函数拿到每个像素的贴图颜色，将它们相乘拿到最终的颜色。

4.5.4 模型

引擎使用的模型格式为.obj，通过加载磁盘上的文件，可以拿到模型的顶点位置、纹理坐标、法线信息，并放到资源管理类，随后就可以从资源管理类获取模型。

模型类定义：



模型绘制效果：

模型绘制——牛头人

①初始化/析构模型数据

模型的加载不需要 GameObject 去关心，它只需要命令 ResourceManager 去获取模型资源。成功获取后需要设置其材质、渲染选项的信息。而模型的析构需要析构属于其基类 RenderAble 的部分，外加 GameObject 特有的模型信息。

②Shader 部分介绍

模型实现了效果高低不同的 4 种 Shader：在顶点部分计算颜色的 Blin、Phong 光照算法，在片元部分计算颜色的 Blin、Phong 光照算法。在 4.5.1 节提出的算法属于抽象算法，在任意阶段使用则代表的意义不同。

顶点着色器只会被调用顶点数量次，也就是说精密度会不够高，效果不如在片元阶段好，但是片元着色器会被调用像素数量词，也就是说被调次数过多会导致性能的下降。引擎中提供了不同精密度、不同效率的 Shader 供开发者选择。

在第二章提到的光照算法属于 Phong 光照模型，而 Blin 光照模型是对其的简化，精密度会略微降低但是性能会大大提高。

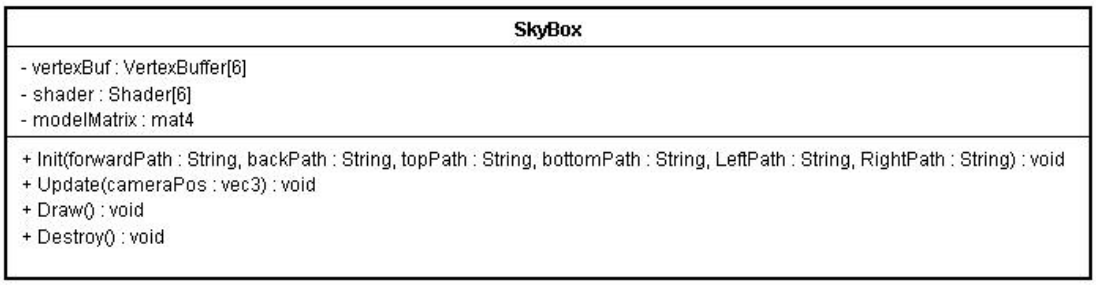
Blin 光照和 Phong 光照的不同只有一点——在镜面反射光计算阶段，Phong 光照会使用 GLSL 内置函数 reflect 计算反射光的方向——vec3 reflectDirection=normalize(reflect(-L,N));而 Blin 光照模型是利用视线方向+光的方向的向量和，来模拟反射光的方向，一个是复杂计算一个是简单的向量求和，性能消耗会大大降低。引擎还实现了 Blin 光照/Phong 光照-逐像素渲染。逐像素渲染即将 Blin 光照和 Phong 光照放到片元着色器去算，这样效率会有所降低但是渲染的真实感会强很多。引擎提供了这 4 种算法，开发者可以结合实际需求考虑所用的 Shader 类型。

4.5.6 天空盒——六面离散形式

引擎中实现了两种天空盒：6 面体和立方体。6 面体顾名思义，就是直接把 6 张贴图拼成一个盒子罩在摄像机上，这样的话需要向 GPU 提交 6 次顶点数据。

天空盒绘制效果（两种类型天空盒渲染效果一致）：

6 面离散形式天空盒定义：



①初始化

六面离散形式的天空盒就是将 6 个面的分别并放到合适的位置，使其正好拼成一个盒子，6 个面相互之间没有联系，仅仅是在 SkyBox 类中将他们组合到一起。6 个面都有自己的顶点信息以及贴图，并都会申请一个 Shader（当然这里只是引用计数）。6 个面都设置完成后 SkyBox 会直接管理这 6 个类。当然，既然存在 6 个面那么每帧绘制天空盒就会提交 6 次渲染数据。

②逻辑更新

要保证天空盒永远在无穷远处，那么就需要这个套在“脑袋“——即摄像机上的盒子永远跟着脑袋保持相对静止。那么一个可行的做法就是，让天空盒跟着摄像机移动。

引擎在每一帧更新天空盒的位置为摄像机当前的位置，保证天空盒永远跟着摄像机移动。

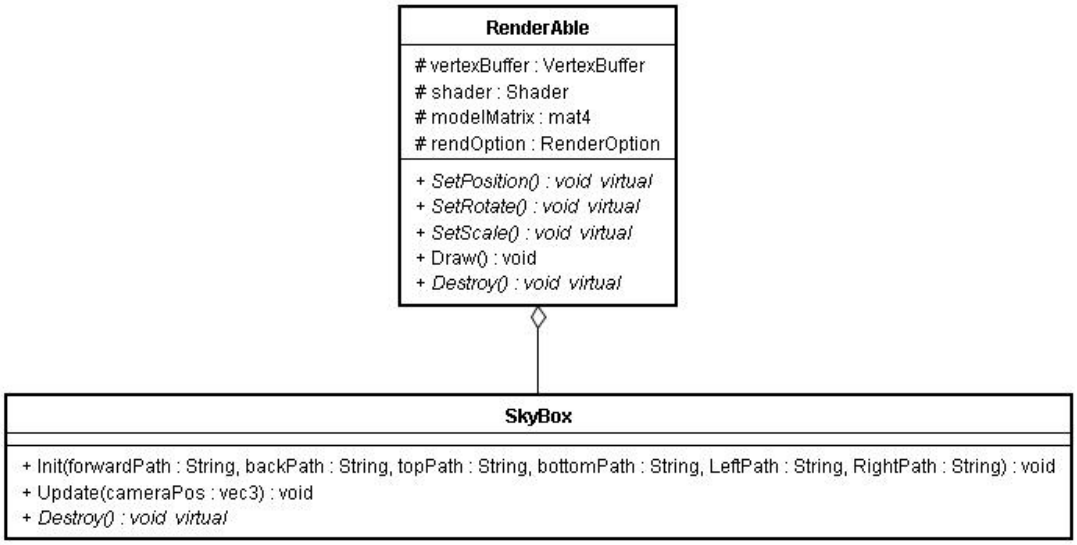
③绘制

既然要绘制 6 个面，那么就要走特殊的渲染列表。特殊的渲染列表需要构造一个虚拟的 RenderDomain 对象，使用自身成员构造虚拟的 RenderDomain 对象后传入渲染列表即可。

4.5.7 天空盒——立方体形式

立方体就是一个展开是 6 个相连接的面，将面拼起来能形成正方体的盒子。GLSL 中有定义相应的数据结构，我们只需要申请该数据结构的贴图，并正确赋值，立方体天空盒就能够被绘制。

立方体天空盒的类定义：



①初始化

立方体天空盒获取一个 Cube.obj 模型，并为其 Shader 设置 CubeMap 属性，在绘制的时候就可以将立方体纹理传到 GPU 了，下面关注立方体天空盒的 Shader 部分。

②着色器

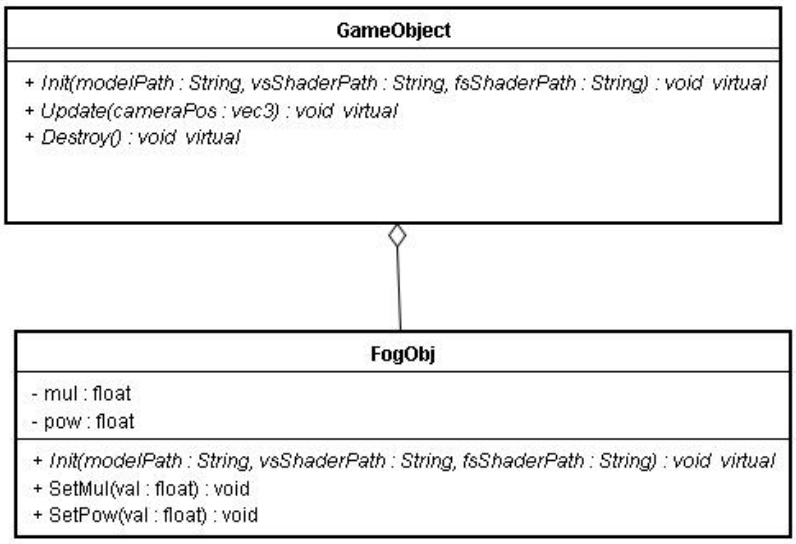
顶点着色器部分，将-pos 作为纹理坐标传递到着色器，这是因为立方体天空盒的纹理实际上要贴在内表面，而-pos 可以取得其内表面 pos。

在片元着色器直接调用 GLSL 内置函数 texture,将在 Shader 中设置的 samplerCube 类型和在顶点着色器设置的纹理坐标传递进去，就可以实现立方体纹理贴图。

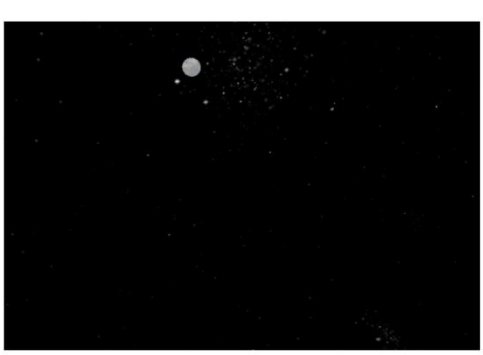
4.5.8 雾化的游戏物体

一个模型的雾化意味着它的身上蒙着一层灰色，引擎通过片元着色器使物体蒙上灰色，并且这个灰色随着距离的减小而变弱，随着距离增加而变浓，下面给出具体实现。

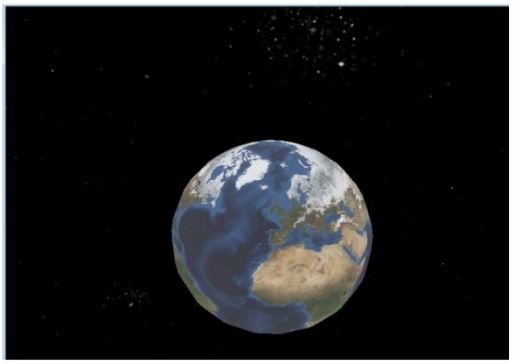
类定义：



渲染效果：



雾化Obj——远处时



雾化Obj——近处时

①初始化

在初始化时只需要在原初始化的基础上，给成员变量衰减因子 mul、幂次数 pow 赋初值即可。

②着色算法

片元着色器将灰色（0.7,0.7,0.7,1.0）和顶点着色器计算出来的颜色做 alpha 混合，其 alpha 值计算根据距离来决定。

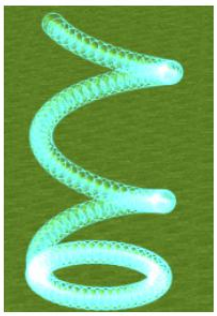
Alpha=(物体与摄像机的距离*mul)的 pow*2 次幂，再将 Alpha 限制在 0.2~1.0 的范围。

用上述式子计算出来的 alpha 值、原本的颜色、灰色做 alpha 混合，即可得到雾化的模型颜色。

4.5.9 粒子系统

引擎中的粒子系统是抽象的，即只提供了接口，具体怎么运动由开发者实现，同时引擎内部提供了笔者默认实现。

渲染效果：



粒子系统——旋涡状粒子

粒子系统——萤火虫

下面给出实现实例：

①旋涡状粒子

首先粒子系统的每一个顶点就是一个粒子。在这里旋涡状粒子给每一个顶点设置位置和颜色，同时在全局层面上，给该粒子系统设置 Shader 以及其他属性。

更新函数让整个粒子系统旋转，这样将使所有粒子同步旋转。然后为每个粒子设置“法线”，这里的法线被用来做其他事情了，在 Shader 就会看到具体使用方式。改变后的粒子系统会呈向上旋转状。

这里将法线当做位置信息去计算，顶点位置=法线+位置，这就实现了粒子系统物体偏移的效果。

第五章 引擎改进与优化

本章将对前文讲述的引擎中已经存在的架构、实现做总结，不会讨论还未实现的部分。

5.1 资源管理模块

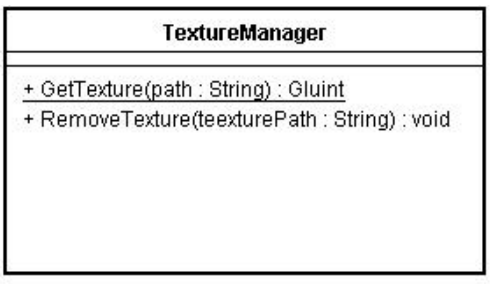
资源管理模块通过引用计数的方式，使用 STL 的 map、vector 容器管理资源。对外提供 Get/Remove 方式获取和卸载资源，同时会增加资源管理模块内的引用计数值。下面对部分有待改善的资源的管理方式加以讨论。

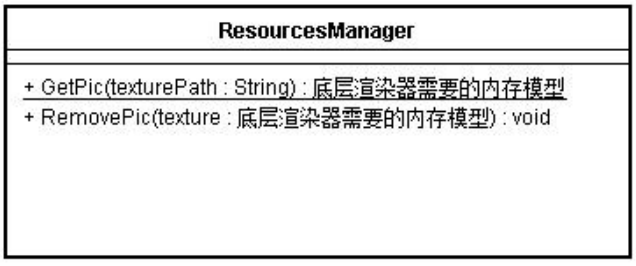
5.1.1 全局层面的资源管理

在目前的引擎架构中，将所有资源（纹理、模型、Shader 等）都放在了一起管理，这不是一个好的选择。因为如果后续的版本迭代中，出现了更多的被管理资源：音频管理、脚本管理等，这将使得资源管理模块变得庞大且冗余，非常不利于引擎的后续功能迭代。

更好的方案是单独写一个类型的管理器，例如：原本底层渲染器需要获取资源，资源管理器<->底层渲染器这样的交互模型拆分，拆分为纹理管理<->资源管理<->底层渲染的形式。此时获取纹理资源的调用堆栈大致是这样的：底层渲染器向资源管理器 Get 某个 Path 的纹理->资源管理器向纹理管理器请求获取 Path 路径的纹理->纹理管理器做处理后返回对应纹理->资源管理器做处理后将纹理返回给底层渲染器。

这里还将使用了适配器设计模式，给出类图：





ResourcesManager 拿到 TextureManager 提供的 GluInt（实际上是 OpenGL 内部支持的内存模型）后，应将其转换为底层渲染器需要的内存模型返回。卸载资源时应该将底层渲染器的内存模型转换为 TextureManager 卸载接口所需的参数类型传入。也就是说，在资源管

理模块中，ResourcesManager 充当适配器的职责，而真正的资源管理应该放在 TextureManager 中进行。

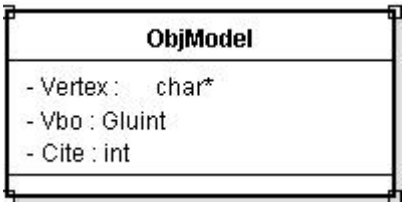
对于 Shader 管理、模型管理同样，这样的设计使得各类资源的管理分开，而对外只提供 ResourcesManager 中必要的接口，实现了一个高内聚、低耦合且易于维护的资源管理器总体架构。

5.1.2 模型资源管理子模块

在此前的设计中，在获取模型时，如果内存中已经有了对应的模型信息，则会再拷贝一份模型信息赋值给另一份 VertexBuffer，也就说，在内存和显存中，都会有一模一样的模型资源占用空间，但是实际上它们没有必要持有两份引用，在新的 ModelManager 中，应该管理的是模型的显存、内存信息。

新的执行流程如下：当一个 VertexBuffer 进来申请已经分配了显存和内存资源的时候，将先前分配的资源数据拷贝给它。也就是说，模型资源管理应该不再进行深拷贝，而是将先前分配的资源浅拷贝给 VertexBuffer，多个 VertexBuffer 持有的顶点指针指向同一块结存，持有的 GLuint 标识符指向同一块显存。

新的 struct ObjModel 类图如下：



该设计的复杂点在于：此时 ModelManager 处的资源信息必须与 BufferManager 同步，因为 BufferManager 才是真正申请、析构显存资源的地方，故需要谨慎设计使得 GPU 资源不会被错误析构。

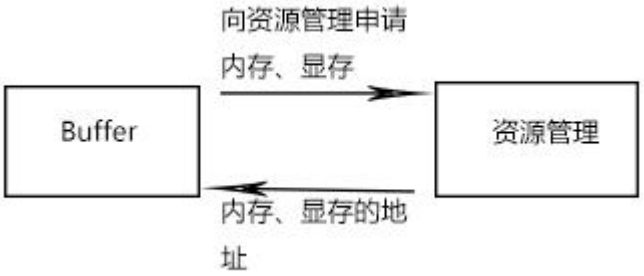
5.2 底层渲染模块

5.2.1 VertexBuffer、ElementBuffer、FrameBuffer

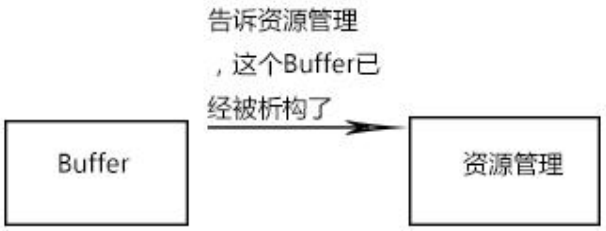
所有 Buffer 代表的都是显卡上的顶点数据和内存上的顶点数据的模型，在当前版本中，Buffer 类分配的内存资源都是在它们 Destory 的时候自行清理的，这样不好，应该将内存资源的释放也交给资源管理模块实现，自身应该只负责告诉资源管理模块析构它的资源。

也就是说：

申请 Buffer 应该是这样的



析构 Buffer 应该是这样的



所有的资源申请、清理应该全权交给资源管理复杂，而不应该在析构函数中清理相应的资源。

5.2.2 粒子系统

目前引擎的粒子系统是通过程序代码控制其移动的，这样不好，究其原因有以下几点：

①程序人员未必能完全理解美术人员的特效需求。

②美术人员未必会通过代码方式编写特效。

③用代码方式编写特效非常不方便。

综上，底层渲染器的特效模块应该有一套可视化工具，能够显示粒子系统参数并将即时的渲染效果展现在小窗口上。

该工具需要完成以下工作

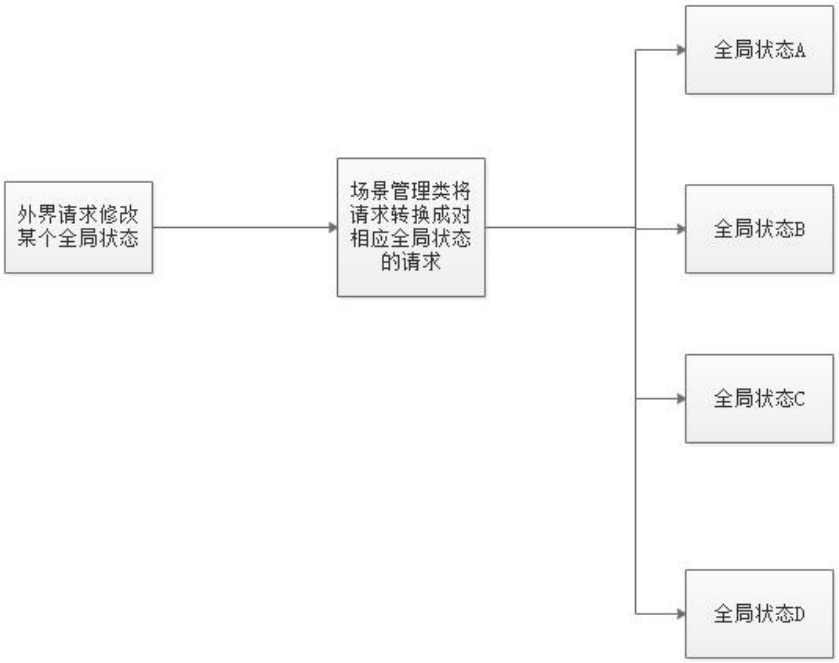
①制作弹窗工具，该弹窗工具应该能显示任意内容。

②在弹窗工具上嫩够显示当前特效结果以及所有可配置的粒子系统参数。

5.3 场景管理模块

5.3.1 全局状态管理

在引擎中，全局状态管理是被放在场景管理类中进行的，这样不利于引擎的后期维护：如果出现了更多的全局状态，那么场景管理类将变得异常庞大，且这样设计不符合：单一职责原则，故提出优化方案如下：



将设计出很多的场景管理类，完成全局状态和场景管理的解耦合，具体做法是：外界调用 SceneManager 的接口，SceneManager 调用对应某个状态的管理函数。这样就完成了集中但是相对耦合度较低的全局状态管理。

第六章 引擎版本展望

本章讨论引擎中还欠缺哪些应该要有的功能，后续几个版本的引擎将以此为基础进行功能迭代。

6.1 模型渲染

未来引擎应加入现在最流行的模型——.FBX 的渲染支持，并且引入关键帧动画、骨骼动画的支持。一个游戏或者说视频，其中的所有人物的表现不可能由平移、旋转、缩放能够完全表现的，故应该引入动画系统的支持，使得引擎更加完善。下面给出解决计划。

6.1.1 资源管理模块加入对动画系统解析

如今的模型管理仅能支持解析.obj 模型，但这还不够，资源管理模块应该提供对.fbx 类型以及与其绑定的关键帧动画、骨骼动画的解析，使底层渲染器能够拿到渲染所需的内存模型。

6.1.2 底层渲染器加入对动画系统的使用

引擎也应该让 GameObject 类型的对象派生出一个能使用动画系统的子类，其子类组合动画系统作为其组件，能够使用播放功能播放该模型所绑定的动画。

6.2 UI 渲染

一个游戏必然会带有 UI 界面，用于显示信息、相应用户点击等等，而 UI 也属于渲染技术的一员。未来引擎中也将加入这一内容。下面我们来谈加入 UI 系统所需的支持。

6.2.1 引入碰撞检测系统

要加入 UI 系统，需要加入对矩形区域的点击检测功能，该功能需要将玩家点击位置与一系列矩形碰撞体进行检测，来判定点击位置是否位于碰撞体内，如果在则触发点击事件。

6.2.2 底层渲染器加入 UI 渲染支持

进行 UI 渲染属于 SDK 无关层的内容，SDK 无关层需要引入这些类型：UILabel（文字）、UISprite（2D 精灵）、UITexture（2D 贴图），用于准备绘制 UI 所需的顶点数据。

6.3 节点管理

这是游戏引擎中很重要的一环，在引擎中也有预搭建相关框架——底层渲染器的 SDK 无关层中中所有物体的共同基类——Node 类，引擎将基于这个框架搭建节点管理。节点管理的意思就是说，让一个物体称为另一个物体的孩子节点。而一个父物体其下所有孩子节点会基于父物体局部坐标系进行坐标转换。

6.3.1 可选数据结构

①BVH，包围体层次结构。“这是包含一组物体的空间体，它要比所包含的几何物体形状简单得多，所以在使用包围体进行碰撞检测等操作的时候比使用物体本身更快。常见的包围体有 AABB（axis-aligned bounding boxes，轴对齐包围盒），OBB（oriented bounding boxes，有向包围盒），以及 k-DOP（discrete oriented polytope，离散定向多面体）”。

②BSP，二叉空间分割树。“在计算机图形学中，BSP 树有两种不同的形式：分别为轴对齐(axis-aligned)和多边形对齐(polygon-aligned)。要创建 BSP 树，首先用一个平面将空间一分为二，然后将几何体按类别划分到这两个空间中，随后以递归形式反复进行这个过程。这种树有一个非常有趣的特性，如果按照一定的方式对树进行遍历，那么会从某个视点将这棵树包含的几何体进行排序（对于轴对齐的方式来说，它是粗略的排序；对于多边形对齐方式来说，它的准确的排序）。在 Z-Buffer 问世之前，基于多边形对齐的 BSP 树成为 3D 游戏进行场景排序的最佳方案。”

③八叉树：类似轴对齐 BSP 树。沿着轴对齐包围盒的三条轴对其进行分割，分割点必须

位于包围盒的中心点，以这种方式生成 8 个新的包围盒。八叉树通过将整个场景包含在一个最小的轴对齐包围盒中进行构造，递归分割，直到达到最大递归层次或包围盒中包含的图元小于某个阈值，其分割过程如图 4 所示。八叉树的使用方式与轴对齐 BSP 树一样，可以处理同类型的查询，也可以用于遮挡裁剪算法中。由于八叉树是具有规则的结构，所以有些查询会比 BSP 树高效。

6.3.2 目标效果

①所有子物体跟随根节点进行移动、旋转、缩放，碰撞检测、可见性检测等其他行为。

第七章 总结与展望

本章在实现了全文所述引擎的基础上，对整个游戏引擎的技术性能进行必要的分析，并对全文进行总结。总结论文的整体主要工作并展望渲染引擎技术发展趋势。

7.1 论文总结

近年来，随着计算机的不断进步，人民生活品质的提高，更优秀的可视化需求使得计算机图形技术不断进步。其中游戏行业对计算机图形技术的推进可谓巨大。而渲染技术作为游戏客户端的底层框架，是实现优秀游戏作品的基石。本文将平日所学图形学知识汇总，设计和实现了一个 3D 游戏渲染引擎。本文主要工作如下：

首先本文讲述了游戏引擎的发展史，介绍了游戏引擎行业的现状，并分析了几款在当代主流的引擎和它们的优缺点。在第二章粗略介绍了计算机图形学、3D 数学以及 OpenGL 的相关知识。

其次架构和实现了一款 3D 游戏渲染引擎，此渲染引擎中包含场景管理、资源管理、底层渲染、平台无关模块。用代码实现了引擎，并展示了渲染后的效果图。在架构和实现过程中，借鉴和现有的国外著名引擎的思路，并对引擎中加入自己在生活中、实习中、学习中基类的知识和尝试。

最后，对该引擎的架构和实现进行了分析，总结不足并提出未来开发方向。

7.2 展望

虽然游戏引擎的发展正蓬勃向上，但国内计算机图形技术的发展似乎出现了断层——中国游戏市场呈百花齐放之势，在品类丰富的基础上，产品性能、画面不断突破，但其所用底层技术均为国外所开发。这更促进了国内引擎技术的发展，相信随着对渲染技术研究的不断嘉善，随着国内的游戏行业的蓬勃发展，游戏引擎技术也会迎来暖阳，未来的渲染技术也会使三维场景的渲染更加逼真和绚丽。

参考文献

[1] David H.Eberly;3D Game Engine:A Practical Approach to Rea-Time Computer Graphics.San Francisco,2001.中译本:《3D 游戏引擎设计：实时计算机图形学的应用方法（第二版）》,清华大学出版社，2013.
[2] Peter Shirley;Fundamentals of Computer Graphics，2001.中译本《计算机图形学基础（第二版）》，人民邮电出版社，2007.
[3] Stanley B. Lippman/Josee Lajoie/Barbara E. Moo;C++ Primer,2013.中译本《C++Primer(第五版)》，电子工业出版社，2013.
[4] Jason Gregory;Game Engine Architecture,2009.中译本《游戏引擎架构》，电子工业出版社，2013.
[5] Fletcher Dunn/Ian Parberry;3D Math for Graphics and Game Development,2002.中译本《3D 数学基础：图形与游戏开发》，清华大学出版社，2005.
[6] Andre Lamothe;Tricks of the 3D Game Programming Gurus-Advanced 3D Graphics and Rasterization,2003.中译本《3D 游戏编程大师技巧》,人民邮电出版社,2013.
[7] Richard S. Wright,Jr./Nicholas Haemel/Graham Sellers/Benjamin Lipchak;OpenGL SuperBible,Fifth Edition,2007.中译本《OpenGL 超级宝典（第五版）》，人民邮电出版社，2012.
[8] John Kessenich/Graham Sellers/Dave Shreiner;OpenGL Programming Guide(Ninth Edition),2017.中译本《OpenGL 编程指南（第九版）》，机械工业出版社，2017.
[9] 毛星云;《Window 游戏编程之从零开始》,清华大学出版社，2013.
[10] Eric Lengyel;Mathematics for 3D Game Programming and Computer Graphics,Third Edition,2012.中译本《3D 游戏与计算机图形学中的数学方法》，清华大学出版社，2016.
[11] Unity Technlogies;《Unity 5.x 从入门到精通》,中国铁道出版社,2016.
[12] Tomas Akenine-Moller/Eric Haines/Naty Hoffman;《Real-Time Rendering,Third Edition》,A K Peters,Lid,2008.
[13] Scott Meters;Effective C++:55 Specific Ways to Improve You Programs and Designs,3rd Edition,2010.中译本《Effective C++:改善程序与设计的 55 个具体做法（第三版）》,电子工业出版社，2011.
[14] Samuel R. Buss;3D Computer Graphics,A Mathematical Introduction with OpenGL,2003.中译本《3D 计算机图形学（OpenGL 版）》,清华大学出版社，2006.
[15] Sanjay Madhav;Game Programming Algorithms and Techniques,2014.中译本《游戏编程算法与技巧》，电子工业出版社，2016.
[16] Romain Marucchi-Foino;Game And Graphics Programming for IOS and Android With OpenGL ES 2.0 ,2012.中译本《OpenGL ES 2.0 游戏与图形编程 ——适用于 IOS 和 Android》，清华大学出版社，2014.
[17] Nicolai M.Josuttis;The C++ Standard Library,2015.中译本《C++标准库（第二版）》，电子工业出版社，2015.
[18] Unity 公司;《Unity5.x/2017 标准教程》,人民邮电出版社,2018.
[19] 丁克宁;北京交通大学硕士论文《3D 游戏引擎的设计与实现》,2011.

附录 A 使用本引擎

本引擎是完全开放源代码的，作者已经将源代码在 gitHub 进行开源：
GitHub 地址：https://github.com/zhanmengao/Main/tree/master/OpenGL/Shader
在该 GitHub 中存有引擎的所有版本，选择最新版本下载即可。