

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，让每个游戏物体持有Shader和显存的引用，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。

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。

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。

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。实现了诸如第一人称摄像机、第三人称摄像机、2D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，例如使用观察者模式搭建摄像机和场景内游戏物体的联系。并使用专业所学的大量数学知识，如摄像机视景体的计算，使三维世界中存在的各种各样的计算更加严谨。

# 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，以及3D场景所需元素。

## 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

 （2-1）

其中右边I表示原始光强，Color表示光的颜色。

点光源

 （2-2）

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

 （2-3）

（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

 （2-4）

ɑ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

 （2-5）

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

 （2-6）

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

 （2-7）

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

 （2-8）

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

 (2-9)

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：

 (2-10)

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：

 （2-11）

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。

 (2-11)

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

 （2-12）

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;

 （2-13）

### 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 GLSL（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 显存（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。

# 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：

资源管理类

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。

场景管理器

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。

底层渲染器

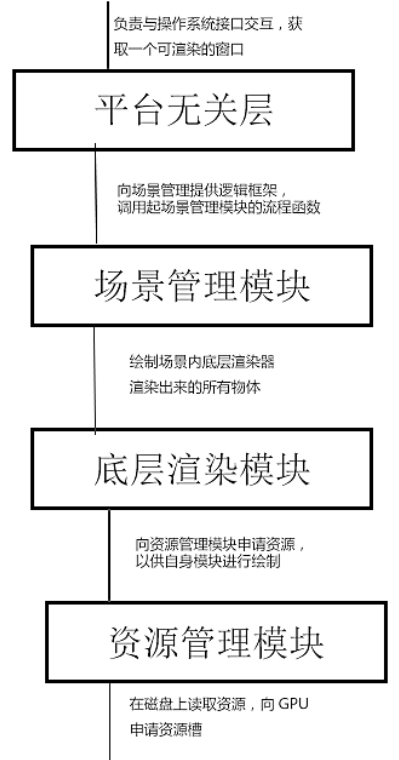
只管渲染一个游戏物体需要做的事情，所需的美术资源由资源管理模块负责加载，底层渲染器使用这些美术资源绘制图像。

平台无关层

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

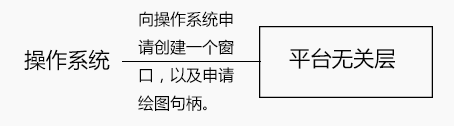
平台无关层的功能相当于main函数，其具体职能如下：

①注册并弹出一个OpenGL渲染窗口。

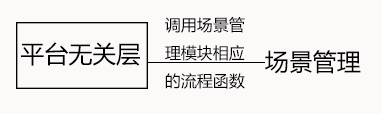
②调用起场景管理模块的逻辑框架。

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：

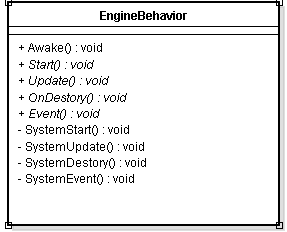
①向外和操作系统交互



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。

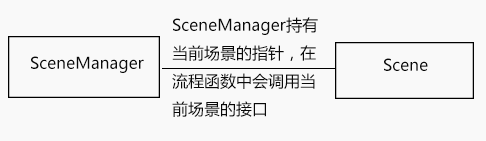
②收集当前帧需要绘制的顶点数据，进行一系列处理后将顶点数据传到GPU。

③提供场景注册、场景切换功能。

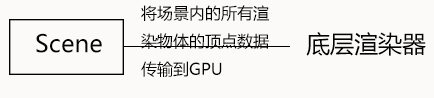
④管理OpenGL状态机的全局状态，例如是否开启alpha混合、是否开启深度测试等。

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），其交互方式如下：

①SceneManager和Scene的交互

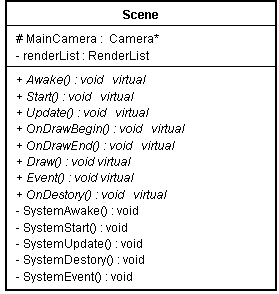


②Scene和底层渲染器的交互



### 3.4.1 Scene

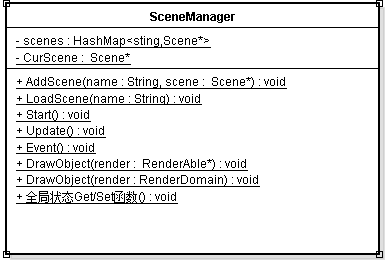
一个Scene表示单个场景，一个场景有初始化、更新、销毁的过程，同一时刻只会有一个场景，渲染引擎渲染的实际上就是该场景内的所有物体，Scene类是所有场景的基类，继承Scene并重写Scene的虚函数就可以完成场景的渲染工作。Scene类定义如下：



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。

### 3.4.2 SceneManager

定义一个Scene并不能够被运行，还需要将Scene注册到SceneManager里边，SceneManager管理所有Scene的逻辑，下面给出类的定义：



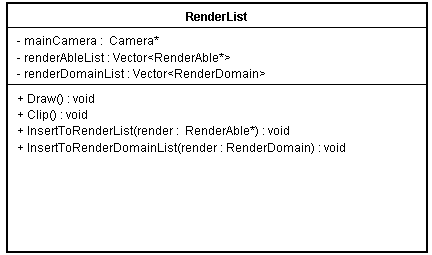
SceneManager使当前场景的逻辑能够正常运行，场景通过AddScene(string name,Scene\* scene)函数被加载到SceneManager中管理起来。在LoadScene(string name)函数中执行卸载上一场景和初始化新的场景，在Update()中执行当前场景的流程函数，在Event()中将事件转发给当前场景。是在单场景之上的多场景管理器。DrawGameObject(RenderAble\* render)和DrawGameObject(const RenderDomain& render)提供渲染游戏物体的功能，填入合适参数后，参数所代表的可渲染对象会被正确渲染。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。

### 3.4.3 RenderList

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。

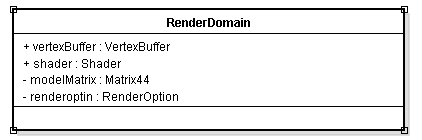
在场景中需要被渲染的一系列顶点数据会进入RenderList。RenderList会与一个摄像机进行关联，来对一个摄像机看到的物体进行一系列处理。然后通过测试的顶点才会传到GPU。RenderList的定义如下：



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额方式进入渲染列表。

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。

## 3.5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。

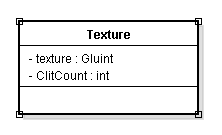
也就是说，资源管理模块的数据流是这样的：

下面讲述具体设计。

### 3.5.1 GPU程序的内存模型

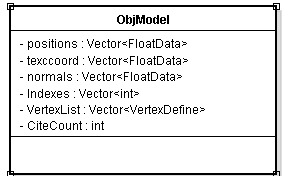
### 

### 3.5.2 纹理类型的内存模型

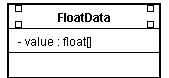
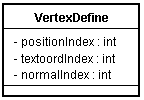


所有GPU资源通过模拟引用计数的形式实现管理，GLuint类型成员表示该资源在GPU的标识，CiteCount 表示当前引用数。并提供一些必要的函数让程序能够正常运行。

### 3.5.3 obj文件的内存模型



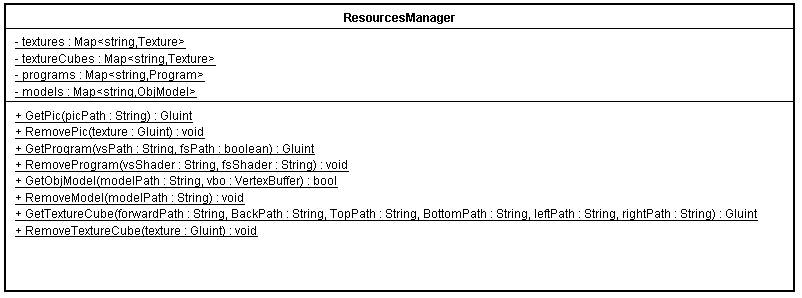
其中泛型成员定义：



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

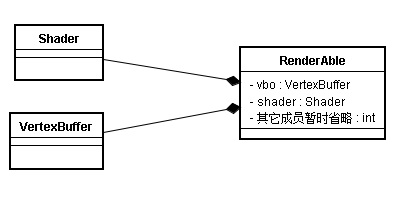
底层渲染模块是最为复杂的，总体上划分为SDK相关部分和SDK无关部分。

SDK相关部分职能如下：

①在CPU抽象出GPU相关信息。

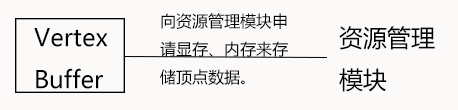
该模块会直接持有底层渲染器所申请的大部分资源，它在架构中的职能是这样的：

①底层渲染模块中，SDK相关部分与SDK无关部分的关系：

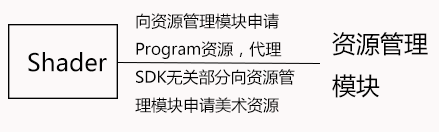


其中RenderAble是可渲染对象的基类。这幅uml图的意思就是说，VertexBuffer和Shader会被可渲染对象的基类组合。

②VertexBuffer和资源管理模块的关系：



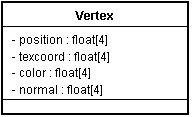
③Shader和资源管理模块的关系：



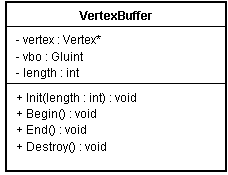
### 3.6.1 VertexBuffer

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。

首先是顶点数据的定义



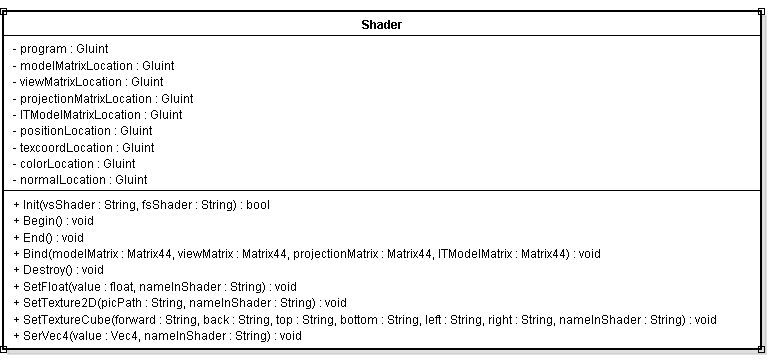
VertexBuffer的定义：



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。

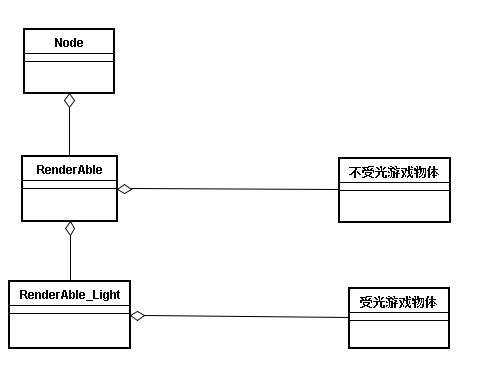
SDK无关部分职能如下：

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。

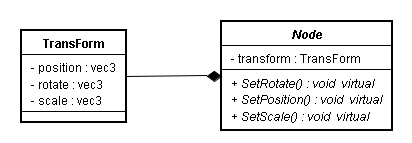
①SDK无关部分的继承关系是这样的：



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

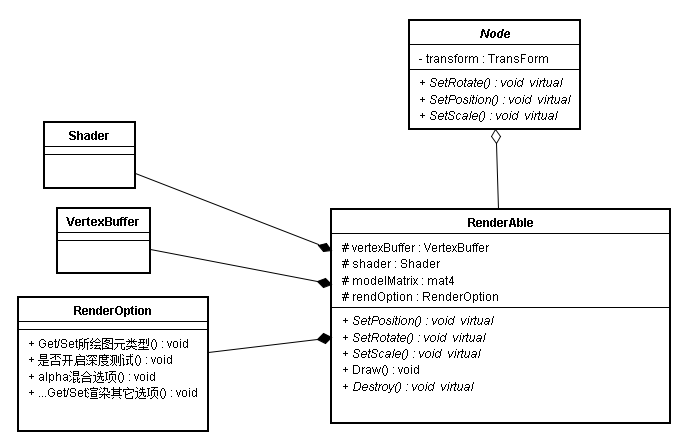
首先是所有游戏物体的一个共同的基类Node。



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。

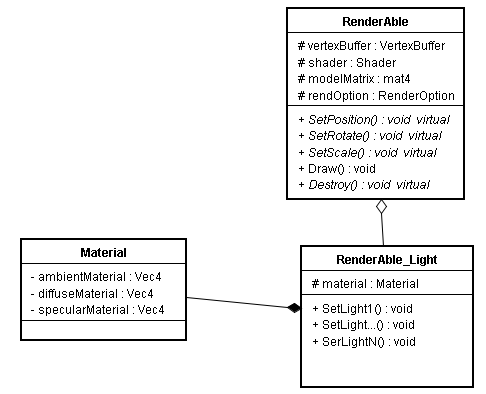


RenderAble类提供了足够多的抽象，它的职能是提供足够多的信息供RenderList使用，RenderOption是传递给RenderList的渲染选项，它随时可以被扩充以适配更复杂的情况。

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，CPU上的灯只负责将数据存储并告诉Shader。

enum LIGHT\_TYPE

{

LIGHT\_INVALID=0,无效灯

LIGHT\_DIRECTION=1,方向光

LIGHT\_POINT=2,点光源

LIGHT\_SPOT=3,聚光灯

};

首先是灯光基类，灯光基类只存储类型和颜色信息，并且需要明确拒绝系统提供构造函数。而且需要是一个抽象类使其传入到其它受光物体的函数。

class Light:public Node

{

public:

virtual int GetType() const = 0;

virtual void SetAmbientColor(float r, float g, float b, float a=1)=0;

virtual void SetDiffuseColor(float r, float g, float b, float a = 1) = 0;

virtual void SetSpecularColor(float r, float g, float b, float a = 1) = 0;

virtual const vec4& GetAmbientColor() const = 0;

virtual const vec4& GetDiffuseColor() const = 0;

virtual const vec4& GetSpecularColor() const = 0;

virtual void SetConstAttenuation(float v) = 0;

virtual float GetConstAttenuation() const = 0;

virtual void SetLinearAttenuation(float v) = 0;

virtual float GetLinearAttenuation() const = 0;

virtual void SetQuadricASttenuation(float v = 0);

virtual float GetQuadricASttenuation() const = 0;

virtual void SetExponent(float v) = 0;

virtual float GetExponent() const = 0;

virtual void SetCutoff(float v) = 0;

virtual float GetCutoff() const = 0;

protected:

Light();

protected:

LIGHT\_TYPE m\_Type;

LightColor m\_Color;//3种颜色

};

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，然后将所有不能被外界调用的函数设为私有。

class DirectionLight:public Light

{

public:

DirectionLight();

virtual inline void SetAmbientColor(float r, float g, float b, float a = 1)

{

m\_Color.SetAmbientColor(r, g, b, a);

}

virtual inline void SetDiffuseColor(float r, float g, float b, float a = 1)

{

m\_Color.SetDiffuseColor(r, g, b, a);

}

virtual inline void SetSpecularColor(float r, float g, float b, float a = 1)

{

m\_Color.SetSepcularColor(r, g, b, a);

}

virtual inline const vec4& GetAmbientColor() const

{

return m\_Color.GetAmbientColor();

}

virtual inline const vec4& GetDiffuseColor() const

{

return m\_Color.GetDiffuseColor();

}

virtual inline const vec4& GetSpecularColor() const

{

return m\_Color.GetSepcularColor();

}

virtual inline int GetType() const

{

return m\_Type;

}

protected:

};

点光源会出现衰减，但没有光的半角信息（相关函数被设置为私有），故定义如下：

class PointLight:public DirectionLight

{

public:

PointLight();

virtual inline void SetConstAttenuation(float v)

{

m\_ConstAttenuation = v;

}

virtual inline float GetConstAttenuation() const

{

return m\_ConstAttenuation;

}

virtual inline void SetLinearAttenuation(float v)

{

m\_LinearAttenuation = v;

}

virtual inline float GetLinearAttenuation() const

{

return m\_LinearAttenuation;

}

virtual inline void SetQuadricASttenuation(float v)

{

m\_QuadricASttenuation = v;

}

virtual inline float GetQuadricASttenuation() const

{

return m\_QuadricASttenuation;

}

protected:

float m\_ConstAttenuation = 1;//设置常数衰减因子

float m\_LinearAttenuation = 0.1f;//设置线性衰减因子

float m\_QuadricASttenuation = 0;//设置平方衰减因子

};

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，故定义如下：

class SpotLight :public PointLight

{

public:

SpotLight();

public:

virtual inline void SetExponent(float v)

{

m\_Exponent = v;

}

virtual inline float GetExponent() const

{

return m\_Exponent;

}

virtual inline void SetCutoff(float v)

{

m\_Cutoff = v;

}

virtual inline float GetCutoff() const

{

return m\_Cutoff;

}

protected:

private:

float m\_Exponent = 90;//聚光灯不衰减角度

float m\_Cutoff = 180;//聚光灯可见角度

};

### 3.7.8 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，这都取决于摄像机的类型。文中所述引擎实现了两种摄像机：第一人称摄像机和第三人称摄像机。至于2D摄像机只是把投影矩阵进行了变换，本质上不属于摄像机类型的变换，故不赘述。

首先是第一人称摄像机，它将玩家视为游戏主角去观察这个世界，使用第一人称摄像机的游戏例如穿越火线（CF）。下面给出第一人称摄像机的定义：

class Camera\_1st

{

public:

Camera\_1st();

public:

virtual void Update();

void SetViewPortSize(float width, float height);

public:

void MoveToLeft(bool isMove);

void MoveToRight(bool isMove);

void MoveToTop(bool isMove);

void MoveToBottom(bool isMove);

virtual void MoveToFront();

virtual void MoveToBack();

void SetMoveSpeed(float speed);

virtual void Pitch(float angle);//绕自己的X轴旋转

virtual void Yaw(float angle);//绕世界坐标系y旋转

void RotateView(float angle, float x, float y, float z);

public:

inline const glm::vec3& GetPosition() const{ return this->m\_Position; };

inline void SetPosition(const vec3& position){ this->m\_Position = position; };

inline glm::mat4 GetViewMatrix() const{ return this->m\_ViewMatrix; };

virtual inline void SetTarget(const vec3\* target){ this->m\_ViewCenter = \*target; };

inline glm::mat4 GetProjectionMatrix() const{ return this->m\_ProjectionMatrix; };

inline float GetWidth() const{ return this->m\_ViewportWidget; };

inline float GetHeight() const{ return this->m\_ViewportHeight; };

inline float GetRadius() const{ return this->m\_Radius; };

inline Frustum GetFrustum() const{ return this->m\_Frustum; }

public:

protected:

glm::vec3 m\_Position;//摄像机位置

glm::vec3 m\_ViewCenter;//目标视点

glm::vec3 m\_Up;//发射出去的方向向量

glm::mat4 m\_ViewMatrix;

glm::mat4 m\_ProjectionMatrix;

Frustum m\_Frustum;

float m\_Radius = 50.0f;//视野

float m\_ViewportWidget = 800;//视口大小

float m\_ViewportHeight = 600;//视口大小

float m\_Near = 0.1f;//最近距离

float m\_Far = 1000;//最远距离

};

摄像机属于特殊的游戏对象，不需要继承Node类，而是定义自己的位置、方向变量而不是组合Transform。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。

### 3.7.9粒子系统

粒子系统是通过绘制一些小颗粒物体（不规律的顶点集）并使其运动来表现特效，它不受光照；有最大数量；根据纹理来显示其外观；有一个中心位置，所有粒子根据粒子系统中心位置发生Transform偏移。ParticleSystem是所有粒子系统的基类，其定义如下：

class ParticleSystem:public RenderAble

{

public:

ParticleSystem();

public:

virtual void Update();

protected:

};

构造函数中需要实现渲染选项的默认设置，例如绘制单位是点、点的大小等。然后不做其他任何事情。Update函数在每一帧更新粒子的位置。引擎实现过程中将定义粒子系统基类，以绘制不同类型的粒子。例如旋涡状、烟花散射状、火焰向上蔓延状、雾化特效的雾气等。

# 编码实现

## 4.1 平台无关层

## 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。

### 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;

static POINT m\_OriginalPos;//记录按下时的位置

static bool m\_IsRotate;//是否正在旋转

static AlphaBlendInfo m\_Blend;//alpha混合状态

static bool m\_IsDepthTest;//是否开启深度测试

static ProgramPointSize m\_IsProgramPointSize;//是否由程序控制点的大小

static ScissorState m\_ScissorState;//裁剪开启状态

static float m\_ViewWidth;

static float m\_ViewHeight;

①注册和加载/卸载场景实现如下：

void SceneManager::AddScene(string name,Scene\* scene)

{

if (scene == nullptr)

{

return;

}

m\_mScene.insert(std::pair<string, Scene\*>(name, scene));

}

void SceneManager::LoadScene(string name)

{

auto it = m\_mScene.find(name);

if (it != m\_mScene.end())

{

if (m\_CurScene == it->second ||it->second==nullptr)

{

return;

}

if (m\_CurScene != nullptr)

{

Scene\* temp = m\_CurScene;

m\_CurScene = nullptr;

cout<<endl;

temp->OnDesrory();

temp->SystemDesrory();

}

cout << endl;

it->second->SystemAwake();

it->second->Awake();

it->second->SetViewPortSize(m\_ViewWidth, m\_ViewHeight);

it->second->SystemStart();

it->second->Start();

}

}

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

void SceneManager::Update()

{

Time::SetDeltaTime();

if (m\_CurScene != nullptr)

{

m\_CurScene->SystemUpdate();

m\_CurScene->Update();

m\_CurScene->SystemOnDrawBegin();

m\_CurScene->OnDrawBegin();

m\_CurScene->Draw3D();

m\_CurScene->SystemDraw3D();

m\_CurScene->SystemDraw2D();

m\_CurScene->Draw2D();

m\_CurScene->SystemOnDraw2DOver();

m\_CurScene->SystemOnDrawOver();

m\_CurScene->OnDrawOver();

}

}

void SceneManager::DrawGameObject(RenderAble\* render)

{

m\_CurScene->m\_3DRendList.InsertToRenderList(render);

}

void SceneManager::DrawGameObject(const RenderDomain& render )

{

m\_CurScene->m\_3DRendList.InsertToRenderList(render);

}

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）。

绘制函数会将输入的参数插入到当前场景的合适的摄像机的渲染列表，也就是说游戏物体不需要知道是哪个渲染列表在渲染它，而SceneManager会处理好一切。

③事件处理实现如下

int SceneManager::Event(UINT message, WPARAM wParam, LPARAM lParam,HWND hwnd)

{

if (m\_CurScene == nullptr)

{

return 0;

}

switch (message)

{

case WM\_KEYDOWN:

m\_CurScene->OnKeyDown(wParam);

break;

case WM\_KEYUP:

m\_CurScene->OnKeyUp(wParam);

break;

case WM\_MOUSEMOVE:

if (m\_IsRotate)

{

//记录点击后的偏移坐标

POINT currentPos;

currentPos.x = LOWORD(lParam);

currentPos.y = HIWORD(lParam);

ClientToScreen(hwnd, &currentPos);

//偏移坐标=现在坐标-点击时候的坐标

int deltaX = currentPos.x - m\_OriginalPos.x;

int deltaY = currentPos.y - m\_OriginalPos.y;

m\_CurScene->OnMouseMove((float)deltaX, (float)deltaY);

SetCursorPos(m\_OriginalPos.x, m\_OriginalPos.y);

}

break;

case WM\_RBUTTONDOWN:

m\_OriginalPos.x = LOWORD(lParam);

m\_OriginalPos.y = HIWORD(lParam);

ClientToScreen(hwnd, &m\_OriginalPos);

SetCapture(hwnd);

ShowCursor(false);

m\_IsRotate = true;

break;

case WM\_RBUTTONUP:

m\_IsRotate = false;

SetCursorPos(m\_OriginalPos.x, m\_OriginalPos.y);

ReleaseCapture();

ShowCursor(true);

break;

case WM\_MOUSEWHEEL:

m\_CurScene->OnMouseWheel(HIWORD(wParam));

break;

case WM\_PAINT:

//绘制

m\_CurScene->Update();

break;

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

default:

return (DefWindowProc(hwnd, message, wParam, lParam));

break;

}

return 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

④全局状态管理实现

void SceneManager::SetBlendState(AlphaBlendInfo info)

{

if (m\_Blend.AlphaBlend == info.AlphaBlend)

{

if (m\_Blend.Type != info.Type)

{

glBlendFunc(GL\_SRC\_ALPHA, info.Type);

}

return;

}

m\_Blend.AlphaBlend = info.AlphaBlend;

if (m\_Blend.AlphaBlend)

{

glEnable(GL\_BLEND);//alpha混合

glBlendFunc(GL\_SRC\_ALPHA, info.Type);

}

else

{

glDisable(GL\_BLEND);//alpha混合

}

}

AlphaBlendInfo SceneManager::GetBlendState()

{

return m\_Blend;

}

其实就是把真正进行设置的OpenGL命令封装到SceneManager里面，SceneManager在内存中通过变量存储了当前状态。好处是如果有可能要改变当前状态，可以省去向GPU发送查询指令消耗的时间；而如果需要改变当前状态但是其实当前状态和需要设置的状态一致时，可以省去发送改变状态指令消耗的时间。

其它状态管理方式和此处相同，故省略。

### 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

void Scene::SystemStart()

{

m\_3DRendList.SetCamera(m\_MainCamera);

SceneManager::SetClearColor(vec4(0, 0, 0, 1));

}

场景开始时需要设置默认的清除颜色，以及给RenderList提供默认的摄像机。

②System游戏循环

void Scene::SystemUpdate()

{

m\_MainCamera->Update();

}

void Scene::SystemOnDrawBegin()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

}

void Scene::SystemDraw3D()

{

m\_3DRendList.Clip();

m\_3DRendList.Draw();

}

void Scene::SystemOnDrawOver()

{

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glUseProgram(0);

}

③System销毁场景

void Scene::SystemDesrory()

{

if (m\_MainCamera != nullptr)

{

delete m\_MainCamera;

m\_MainCamera = nullptr;

}

}

④其它非System函数

public:

virtual bool Awake();

protected:

virtual void Start();

virtual void Update();

virtual void OnDrawBegin();

virtual void Draw3D();

virtual void Draw2D();

virtual void OnDrawOver();

virtual void OnDesrory();

virtual void SetViewPortSize(float width, float height);

virtual void OnKeyDown(char KeyCode);

virtual void OnKeyUp(char KeyCode);

virtual void OnMouseMove(float deltaX, float deltaY);

virtual void OnMouseWheel(int direction);

这些事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 RenderList

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;//SDK无关层输入的顶点数据会进到这

std::vector<RenderDomain> m\_DomainRenderList;//特殊的输入

①裁剪

void RenderList::Clip()

{

SceneManager::SetScissorState(ScissorState(1, m\_pMainCamera->GetXStart(), m\_pMainCamera->GetYStart(), m\_pMainCamera->GetWidth(), m\_pMainCamera->GetHeight()));

}

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。

void SceneManager::SetScissorState(const ScissorState& scissor)

{

if (m\_ScissorState != scissor)

{

m\_ScissorState = scissor;

if (m\_ScissorState.m\_IsScissor)

{

glEnable(GL\_SCISSOR\_TEST);

glScissor(m\_ScissorState.xStart, m\_ScissorState.yStart, m\_ScissorState.width, m\_ScissorState.height);

}

else

{

glDisable(GL\_SCISSOR\_TEST);

glScissor(m\_ScissorState.xStart, m\_ScissorState.yStart, m\_ScissorState.width, m\_ScissorState.height);

}

}

}

②绘制

void RenderList::Draw()

{

if (m\_pMainCamera == nullptr)

{

return;

}

//处理domain

for (int i = 0; i < m\_DomainRenderList.size(); i++)

{

RenderDomain render = m\_DomainRenderList[i];

render.vertexBuf.Begin();

{

render.shader.Begin();

glm::mat4 ITMatrix = glm::inverseTranspose(render.modelMatrix);

render.shader.Bind(glm::value\_ptr(render.modelMatrix), glm::value\_ptr(m\_pMainCamera->GetViewMatrix()), glm::value\_ptr(m\_pMainCamera->GetProjectionMatrix())

, glm::value\_ptr(ITMatrix));

{

SceneManager::SetBlendState(render.options.alphaBlend);

SceneManager::SetDepthTestState(render.options.DepthTest);

SceneManager::SetProgramPointSizeState(render.options.Program\_Point\_Size);

glDrawArrays(render.options.DrawType, 0, render.vertexBuf.GetLenth());

}

render.shader.End();

}

render.vertexBuf.End();

}

m\_DomainRenderList.clear();

//处理RenderList

for (int i = 0; i < m\_RendList.size(); i++)

{

RenderAble\* render = m\_RendList[i];

if (render->OnEnable())

{

render->GetVertexBuffer().Begin();

{

render->GetShader().Begin();

glm::mat4 ITMatrix = glm::inverseTranspose(render->GetModelMatrix());

render->GetShader().Bind(glm::value\_ptr(render->GetModelMatrix()), glm::value\_ptr(m\_pMainCamera->GetViewMatrix()), glm::value\_ptr(m\_pMainCamera->GetProjectionMatrix())

, glm::value\_ptr(ITMatrix));

{

SceneManager::SetBlendState(render->GetAlphaBlend());

SceneManager::SetDepthTestState(render->IsDepthTest());

SceneManager::SetProgramPointSizeState(render->IsProgramPointSize());

glDrawArrays(render->GetType(), 0, render->GetVertexBuffer().GetLenth());

}

render->GetShader().End();

}

render->GetVertexBuffer().End();

}

}

m\_RendList.clear();//渲染完了进行清空

}

绘制实际上就是将两个容器中的绘制信息取出来，直接传递到GPU。

引擎会到最后才执行绘制函数，在此之前关于顶点的处理已经全部结束，剩下的都是要直接传递到GPU的数据。

## 4.3 资源管理

资源管理为底层渲染器提供资源。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/解获取2D纹理资源

GLuint ResourceManager::GetPic(const char\* bmpPath,bool isRepeat)

{

string name(bmpPath);

auto it = m\_mTexture.find(name);

if (it != m\_mTexture.end())

{

it->second.CiteCount++;

return it->second.texture;

}

else

{

//去读文件

GLuint texture = CreateTexture2D(bmpPath, isRepeat);

printf("创建图片%s\r\n", bmpPath);

Texture textureStrc;

textureStrc.CiteCount=1;

textureStrc.texture = texture;

m\_mTexture.insert(std::pair<string, Texture>(name, textureStrc));

return texture;

}

}

void ResourceManager::RemovePic(GLuint texture)

{

//引用计数

for (auto it = m\_mTexture.begin(); it != m\_mTexture.end();it++)

{

if (it->second.texture == texture)

{

it->second.CiteCount--;

cout << "卸载图片" << it->first << ",剩余引用数" << it->second.CiteCount << endl;

if (it->second.CiteCount <= 0)

{

glDeleteTextures(1, &(it->second.texture));

m\_mTexture.erase(it);

return;

}

}

}

//引用计数

for (auto it = m\_mProgramTexture.begin(); it != m\_mProgramTexture.end(); it++)

{

if (it->second.texture == texture)

{

it->second.CiteCount--;

cout << "卸载程序纹理" << it->first << ",剩余引用数" << it->second.CiteCount << endl;

if (it->second.CiteCount <= 0)

{

glDeleteTextures(1, &(it->second.texture));

m\_mProgramTexture.erase(it);

return;

}

}

}

}

②获取/解获取TextureCube资源

GLuint ResourceManager::GetTextureCube(const char\* front, const char\* back, const char\* top, const char\* bottom, const char\* left, const char\* right)

{

string name = string(front) + string(back) + string(top) + string(bottom) + string(left) + string(right);

auto it = m\_mTextureCube.find(name);

if (it == m\_mTextureCube.end())

{

//去申请一个资源

GLuint textureId = CreateCubeMap(front, back, top, bottom, left, right);

if (textureId != \_INVALID\_ID\_)

{

Texture texture;

texture.CiteCount = 1;

texture.texture = textureId;

cout << "创建CubeMap:"<<name<<endl;

m\_mTextureCube.insert(std::pair<string, Texture>(name, texture));

}

return textureId;

}

else

{

it->second.CiteCount++;

return it->second.texture;

}

}

void ResourceManager::RemoveTextureCube(GLuint texture)

{

for (auto it = m\_mTextureCube.begin(); it != m\_mTextureCube.end(); it++)

{

if (it->second.texture == texture)

{

it->second.CiteCount--;

cout << "销毁CubeMap:" << it->first<<"剩余引用次数"<<it->second.CiteCount << endl;

if (it->second.CiteCount <= 0&& texture!=\_INVALID\_ID\_)

{

glDeleteTextures(1, &texture);

m\_mTextureCube.erase(it);

return;

}

}

}

}

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创建纹理

GLuint ResourceManager::CreateTexture2D(const char\* fileName, bool isRepeat)

{

unsigned texId = 0;

//1 获取图片格式

FREE\_IMAGE\_FORMAT fifmt = FreeImage\_GetFileType(fileName, 0);

//2 加载图片

FIBITMAP \*dib = FreeImage\_Load(fifmt, fileName, 0);

//3 转化为rgb 24色

dib = FreeImage\_ConvertTo24Bits(dib);

//4 获取数据指针

unsigned char \*pixels = (unsigned char\*)FreeImage\_GetBits(dib);

int width = FreeImage\_GetWidth(dib);

int height = FreeImage\_GetHeight(dib);

glGenTextures(1, &texId);

glBindTexture(GL\_TEXTURE\_2D, texId);

//设置参数

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR); glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

if (!isRepeat)

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

}

else

{

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

}

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_BGR, GL\_UNSIGNED\_BYTE, pixels)

\* 释放内存

\*/

FreeImage\_Unload(dib);

return texId;

}

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。

③创建立方体纹理

GLuint ResourceManager::CreateCubeMap(const char\* front, const char\* back, const char\* top, const char\* bottom, const char\* left, const char\* right)

{

GLuint texture;

glGenTextures(1, &texture);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, texture);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP,GL\_TEXTURE\_WRAP\_S,GL\_CLAMP\_TO\_EDGE)

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP,GL\_TEXTURE\_WRAP\_T,GL\_CLAMP\_TO\_EDGE)

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP,GL\_TEXTURE\_WRAP\_R,GL\_CLAMP\_TO\_EDGE)

int width, height, nFileSize;

unsigned char \*bmpFileContent = LoadFileContent(front, nFileSize);

unsigned char\*pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

bmpFileContent = LoadFileContent(back, nFileSize);

pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Z, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

bmpFileContent = LoadFileContent(left, nFileSize);

pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_X, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

bmpFileContent = LoadFileContent(right, nFileSize);

pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

bmpFileContent = LoadFileContent(top, nFileSize);

pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_Y, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

bmpFileContent = LoadFileContent(bottom, nFileSize);

pixelData = DecodeBMP(bmpFileContent, width, height);

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_NEGATIVE\_Y, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, pixelData);

delete bmpFileContent;

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, \_INVALID\_ID\_);

return texture;

}

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，传输字节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。

从磁盘加载图片文件，这里使用的是引擎自行实现的函数static unsigned char\* LoadFileContent(const char\* path, int& filesize)和unsigned char\* DecodeBMP(unsigned char\* bmpFileData, int& width, int& height)加载字节数据和解码bmp文件，而没有使用外部库。这两个函数实现如下。

unsigned char\* ResourceManager::LoadFileContent(const char\* path, int& filesize)

{

unsigned char\* fileContent = nullptr;

filesize = 0;

FILE\* pFile = fopen(path, "rb");

//按二进制打开文件，只读

if (pFile)

{

fseek(pFile, 0, SEEK\_END);//移动到文件尾

int nLen = ftell(pFile);

if (nLen > 0)

{

rewind(pFile);//移到文件头部

fileContent = new unsigned char[nLen + 1];

fread(fileContent, sizeof(unsigned char), nLen, pFile);

fileContent[nLen] = '\0';

filesize = nLen;

}

fclose(pFile);

}

return fileContent;

}

该函数加载任意类型文件字节数据到内存中，并返回堆区的内存指针。该函数不能被滥用，因为外部需要手动delete该内存。它需要做这几件事情：根据路径按只读方式打开文件->拿到文件长度->new一块unsigned char类型数组，并将文件指针的数据加载到数组中->关闭文件指针，返回字节数组的首地址。

unsigned char\* DecodeBMP(unsigned char\* bmpFileData, int& width, int& height)

{

//如果文件头是0X4D42 说明是BMP文件

if (0x4D42 == \*((unsigned short\*)bmpFileData))

{

int pixelDataOffset = (\*(int\*)(bmpFileData + 10));//像素数据的偏移值

width = (\*(int\*)(bmpFileData + 18));//图片宽

height = (\*(int\*)(bmpFileData + 22));//图片高

unsigned char\* pixelData = bmpFileData + pixelDataOffset;//像素数据

//由BGR->RGB

for (int i = 0; i < width\*height \* 3; i += 3)

{

unsigned char temp = pixelData[i];//b

pixelData[i] = pixelData[i + 2];//b=r

pixelData[i + 2] = temp;//r=b

}

return pixelData;

}

return nullptr;

}

该函数将BMP图片中的信息读取、解码并返回。先判断是否是bmp真正文件->从文件第10位字节拿到像素数据偏移值->从18号字节拿到图片宽->从22号字节拿到图片高->将像素数据从文件的bgr转码为rgb->返回重新组织过的bmp图片字节数据。

### 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获取和卸载GPU程序

GLuint ResourceManager::GetProgram(const char\* vertexShaderPath, const char\* fragmentShaderPath)

{

string name = string(vertexShaderPath) +","+string(fragmentShaderPath);

auto it = m\_mProgram.find(name);

if (it != m\_mProgram.end())

{

it->second.CiteCount++;

return it->second.program;

}

else

{

int nFileSize = 0;

const char\* vertexShaderCode = (char\*)LoadFileContent(vertexShaderPath, nFileSize);

ASSERT\_PTR\_BOOL(vertexShaderCode);

GLuint vsShader = CompileShader(GL\_VERTEX\_SHADER, vertexShaderCode);

delete vertexShaderCode;

ASSERT\_INT\_BOOL(vsShader);

const char\* fragmentShaderCode = (char\*)LoadFileContent(fragmentShaderPath, nFileSize);

ASSERT\_PTR\_BOOL(fragmentShaderCode);

GLuint fsShader = CompileShader(GL\_FRAGMENT\_SHADER, fragmentShaderCode);

delete fragmentShaderCode;

ASSERT\_INT\_BOOL(fsShader);

GLuint program = CreateProgram(vsShader, fsShader);

printf("创建程序%s,%s\r\n", vertexShaderPath,fragmentShaderPath);

Program programStru;

programStru.program = program;

programStru.CiteCount++;

m\_mProgram.insert(std::pair<string, Program>(name, programStru));

return program;

}

}

void ResourceManager::RemoveProgram(GLuint program)

{

for (auto it = m\_mProgram.begin(); it != m\_mProgram.end();)

{

if (it->second.program == program)

{

it->second.CiteCount--;

cout << "卸载程序" << it->first << ",剩余引用数" << it->second.CiteCount << endl;

if (it->second.CiteCount <= 0)

{

glDeleteProgram(it->second.program);

it = m\_mProgram.erase(it);

return;

}

}

it++;

}

}

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；如果没有编译则从磁盘分别将VertexShader、FragmentShader读到内存并送到GPU编译，若都编译通过就将两个Shader绑定并链接到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。

②编译Shader

GLuint ResourceManager::CompileShader(GLenum shaderType, const char\* shaderCode)

{

//创建一个shader对象

GLuint shader = glCreateShader(shaderType);

//把shader源码放到GPU

glShaderSource(shader, 1, &shaderCode, nullptr);

glCompileShader(shader);

GLint compileResule = GL\_TRUE;

//查看shader的编译状态

glGetShaderiv(shader, GL\_COMPILE\_STATUS, &compileResule);

if (compileResule == GL\_FALSE)

{

char szLog[1024] = { 0 };

GLsizei logLen = 0;

//拿到错误日志

glGetShaderInfoLog(shader, 1024, &logLen, szLog);

printf("Comoile Shaderr fail error log:%s\nshader code:\n%s\n", szLog, shaderCode);

glDeleteShader(shader);

shader = 0;

}

return shader;

}

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。

③链接Shader成GPU程序

GLuint ResourceManager::CreateProgram(GLuint vertexShader, GLuint fragmentShader)

{

GLuint program = glCreateProgram();

//将shader绑定到程序上

glAttachShader(program, vertexShader);

glAttachShader(program, fragmentShader);

glLinkProgram(program);

//解绑定

glDetachShader(program, vertexShader);

glDetachShader(program, fragmentShader);

GLint linkResult;

//检查程序链接是否OK

glGetProgramiv(program, GL\_LINK\_STATUS, &linkResult);

if (linkResult == GL\_FALSE)

{

char szLog[1024] = { 0 };

GLsizei logLen = 0;

//拿到错误日志

glGetProgramInfoLog(program, 1024, &logLen, szLog); //shader，错误日志最多多少字符，做输出拿到实际多少个字符，日志写在哪

printf("Comoile Shaderr fail error log:%s\nshader code:\n%s\n", szLog, program);

glDeleteShader(

program);

program = 0;

}

return program;

}

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模型的获取和卸载

bool ResourceManager::GetModel(const char\* path, VertexBuffer &vbo)

{

string name(path);

auto it = m\_mModel.find(name);

if (it != m\_mModel.end())

{

const auto& obj = it->second;

it->second.CiteCount++;

//根据上面读到的信息，构造VertexBuffer

vbo.Init(obj.Indexes.size());

for (int i = 0; i < obj.Indexes.size(); i++)

{

if (obj.Indexes[i] < obj.VertexList.size())

{

VertexDefine vertex = obj.VertexList.at(obj.Indexes[i]); //取出绘制指令

//位置信息

if (vertex.positionIndex - 1 > obj.positions.size())

{

continue;

}

FloatData data = obj.positions[vertex.positionIndex - 1]; //OBJ文件里面索引信息从1开始

vbo.SetPosition(i, data.v[0], data.v[1], data.v[2]);

//uv

if (vertex.textcoordIndex - 1>obj.texcoords.size())

{

continue;

}

data = obj.texcoords[vertex.textcoordIndex - 1];

vbo.SetTexcoord(i, data.v[0], data.v[1], data.v[2]);

//normal

if (vertex.normalIndex - 1 > obj.normals.size())

{

continue;

}

data = obj.normals[vertex.normalIndex - 1];

vbo.SetNormal(i, data.v[0], data.v[1], data.v[2]);

vbo.SetColor(i, 1, 1, 1, 1);

}

}

return 1;

}

else

{

int nFileSize = 0;

unsigned char\* fileContent = LoadFileContent(path, nFileSize);//读到文件字节流

ASSERT\_PTR\_BOOL(fileContent);

//以上的仅仅是信息，而绘制指令教我们怎么把那些信息组织成一个顶点

//顶点应该包含：位置信息，法线信息，纹理贴图信息

vector<FloatData> positions;

vector<FloatData> texcoords;

vector<FloatData> normals;

vector<VertexDefine> VertexList;

vector<int> Indexes;

std::stringstream ssFileContent((char \*)fileContent);//用字符流去读

char buffer[256];//缓存

while (!ssFileContent.eof())

{

memset(buffer, 0, 256);

ssFileContent.getline(buffer, 256);//读到buffer，同时ss指针后移

//如果行不为空

if (strlen(buffer) > 0)

{

//如果是v开头

if (buffer[0] == 'v')

{

std::stringstream ssBuffer(buffer);//利用stringstream遇到空格就中断流的特性

char temp[256];

//如果是vt

if (buffer[1] == 't')

{

ssBuffer >> temp;//把字母vt流出去，剩下2个数字信息（表示uv）

FloatData data;

ssBuffer >> data.v[0];//u

ssBuffer >> data.v[1];//v

texcoords.push\_back(data);

}

else if (buffer[1] == 'n')

{

ssBuffer >> temp;

FloatData data;

ssBuffer >> data.v[0];//x

ssBuffer >> data.v[1];//y

ssBuffer >> data.v[2];//z

normals.push\_back(data);

}

else

{

ssBuffer >> temp;

FloatData data;

ssBuffer >> data.v[0];//x

ssBuffer >> data.v[1];//y

ssBuffer >> data.v[2];//z

positions.push\_back(data);

}

}

else if (buffer[0] == 'f')

{

std::stringstream ssBuffer(buffer);

char temp[256];

ssBuffer >> temp;

string vertexString;//存储形如1/1/1的信息

//一次解析一行f数据（一个面）

for (unsigned i = 0; i < 3; i++)

{

ssBuffer >> vertexString;

//第一对信息：位置信息

size\_t pos = vertexString.find\_first\_of('/');

string posIndexStr = vertexString.substr(0, pos);

//第二对信息：纹理信息

size\_t pos2 = vertexString.find\_first\_of('/', pos + 1);

string texcoordIndexStr = vertexString.substr(pos + 1, pos2 - 1 - pos);

//第三对信息：法线信息

string normalIndexStr = vertexString.substr(pos2 + 1, normalIndexStr.length() - pos2 - 1);

VertexDefine vertex;

vertex.positionIndex = atoi(posIndexStr.c\_str());

vertex.textcoordIndex = atoi(texcoordIndexStr.c\_str());

vertex.normalIndex = atoi(normalIndexStr.c\_str());

int index = INVALID;

//如果集合里有相同的，则不添加

for (unsigned i = 0; i < VertexList.size(); i++)

{

if (VertexList[i].positionIndex == vertex.positionIndex && VertexList[i].normalIndex == vertex.normalIndex && VertexList[i].textcoordIndex == vertex.textcoordIndex)

{

index = i;//如果有相同的，就认为是

break;

}

}

//如果没有相同的

if (index == INVALID)

{

index = VertexList.size();

VertexList.push\_back(vertex);//如果是第一次出现，则保存该指令。否则仅保存其索引值

}

Indexes.push\_back(index);//保存该指令的索引值

}

}

}

}

if (fileContent != nullptr)

{

delete fileContent;

}

//根据上面读到的信息，构造VertexBuffer

vbo.Init(Indexes.size());

for (int i = 0; i < Indexes.size(); i++)

{

if (Indexes[i] < VertexList.size())

{

VertexDefine vertex = VertexList.at(Indexes[i]); //取出绘制指令

//位置信息

if (vertex.positionIndex - 1 > positions.size())

{

continue;

}

FloatData data = positions[vertex.positionIndex - 1]; //OBJ文件里面索引信息从1开始

vbo.SetPosition(i, data.v[0], data.v[1], data.v[2]);

//uv

if (vertex.textcoordIndex - 1>texcoords.size())

{

continue;

}

data = texcoords[vertex.textcoordIndex - 1];

vbo.SetTexcoord(i, data.v[0], data.v[1], data.v[2]);

//normal

if (vertex.normalIndex - 1 > normals.size())

{

continue;

}

data = normals[vertex.normalIndex - 1];

vbo.SetNormal(i, data.v[0], data.v[1], data.v[2]);

}

}

printf("创建模型%s\r\n", path);

//存储到map

ObjModel obj;

obj.Indexes.assign(Indexes.begin(), Indexes.end());

obj.normals.assign(normals.begin(), normals.end());

obj.texcoords.assign(texcoords.begin(), texcoords.end());

obj.VertexList.assign(VertexList.begin(), VertexList.end());

obj.positions.assign(positions.begin(), positions.end());

obj.CiteCount = 1;

m\_mModel.insert(std::pair<string,ObjModel>(name, obj));

return 1;

}

}

void ResourceManager::RemoveModel(string path)

{

auto it = m\_mModel.find(path);

if (it != m\_mModel.end())

{

it->second.CiteCount--;

cout << "卸载模型" << it->first << ",剩余引用数" << it->second.CiteCount<<endl;

if (it->second.CiteCount <= 0)

{

m\_mModel.erase(it);

}

}

}

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，下面重点讲解OBJ格式模型数据的解释方式。

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。至于为什么不同享内存，这是因为模型数据可能在引擎中会被人为改变，所以模型数据不应该共享内存，而是要多分配一块内存来避免数据紊乱、错误。

## 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（这里表示GPU程序整体，因为只有单个shader没有任何意义）、可绘制对象、受光的绘制对象等。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。

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

bool VertexBuffer::Init(const int& Lenth)

{

if (!m\_IsInit)

{

this->m\_Vertex = new Vertex[Lenth];

m\_Lenth = Lenth;

m\_Vbo = ResourceManager::CreateBufferObject(GL\_ARRAY\_BUFFER, sizeof(Vertex)\*Lenth, GL\_STATIC\_DRAW, nullptr);

ASSERT\_INT\_BOOL(m\_Vbo);

m\_IsInit = 1;

return 1;

}

return 0;

}

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I nit函数被执行之后内存就开辟好了，可以往里面写数据了。

②设置数据

void VertexBuffer::SetNormal(const int& index, const float& x, const float& y, const float& z, const float& w)

{

if (index >= this->m\_Lenth || !m\_IsInit)

{

return;

}

m\_Vertex[index].SetNormal(x, y, z, w);

}

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。

这里以法线为例讲解顶点数据的设置，其余的纹理坐标、位置信息、顶点颜色和法线设置方式完全相同。

③绘制前准备

void VertexBuffer::Begin()

{

if (m\_Vbo == \_INVALID\_ID\_ || !m\_IsInit)

{

return;

}

glBindBuffer(GL\_ARRAY\_BUFFER, m\_Vbo);

glBufferSubData(GL\_ARRAY\_BUFFER, 0, sizeof(Vertex)\*m\_Lenth, this->m\_Vertex);

}

void VertexBuffer::End()

{

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

}

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。在绘制结束后设置当前Buffer为无效Buffer，防止后续有错误修改影响到该缓存。

### 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。

成员变量回顾：

bool m\_IsInit = 0;

GLuint m\_Program;

//常规选项

GLint m\_ModelMatrixLocation, m\_ViewMatrixLocation, m\_ProjactionMatrixLocation,m\_ITModelMatrixLocation;

GLint m\_PositionLocation, m\_ColorLocation, m\_TexcoordLocation, m\_NormalLocation;

//其它选项

std::map<string, UniformTexture> m\_mUniformTextures;

std::map<string, UniformTexture> m\_mUniformCubeMap;

std::map<string, UniformVec4> m\_mUniformVec4;

std::map<string, UniformVec3> m\_mUniformVec3;

std::map<string, UniformFloat> m\_mUniformFloats;

std::map<string, UniformMatrix> m\_mUniformMatrixs;

string m\_ModelMatrixNameInShader = "ModelMatrix";

string m\_ViewMatrixNameInShader = "ViewMatrix";

string m\_ProjectionMatrixNameInShader = "ProjectionMatrix";

string m\_ITModelMatrixNameInShader = "IT\_ModelMatrix";

string m\_PositionName = "position";

string m\_ColorName = "color";

string m\_NormalName = "normal";

string m\_TexcoordName = "texcoord";

①Shader初始化

bool Shader::Init(const char\* vertShaderPath, const char\* fragmentShaderPath)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

m\_Program = ResourceManager::GetProgram(vertShaderPath, fragmentShaderPath);

ASSERT\_INT\_BOOL(m\_Program);

//获取各Location

m\_ModelMatrixLocation = glGetUniformLocation(m\_Program, m\_ModelMatrixNameInShader.c\_str());

m\_ViewMatrixLocation = glGetUniformLocation(m\_Program, m\_ViewMatrixNameInShader.c\_str());

m\_ProjactionMatrixLocation = glGetUniformLocation(m\_Program, m\_ProjectionMatrixNameInShader.c\_str());

m\_ITModelMatrixLocation = glGetUniformLocation(m\_Program, m\_ITModelMatrixNameInShader.c\_str());

m\_PositionLocation = glGetAttribLocation(m\_Program, m\_PositionName.c\_str());

m\_ColorLocation = glGetAttribLocation(m\_Program, m\_ColorName.c\_str());

m\_NormalLocation = glGetAttribLocation(m\_Program, m\_NormalName.c\_str());

m\_TexcoordLocation = glGetAttribLocation(m\_Program, m\_TexcoordName.c\_str());

return 1;

}

return 0;

}

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。

例如设置模型矩阵：

void Shader::SetModelMatrixNameInShader(string modelMatrix)

{

m\_ModelMatrixNameInShader = modelMatrix;

INIT\_TEST\_VOID

m\_ModelMatrixLocation = glGetUniformLocation(m\_Program, m\_ModelMatrixNameInShader.c\_str());

}

该函数设置ModelMatrix的名称，并且会根据该名称去加载名称对应的变量。

例如设置位置信息：

void Shader::SetPositionName(string name)

{

m\_PositionName = name;

INIT\_TEST\_VOID

m\_PositionLocation = glGetAttribLocation(m\_Program, m\_PositionName.c\_str());

}

②Shader其它变量设置

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。

设置2D纹理类型：

void Shader::SetTexture2D(const char\* imagePath, bool isRepeat, const char\* nameInShader)

{

INIT\_TEST\_VOID

ASSERT(imagePath);

auto it = m\_mUniformTextures.find(nameInShader);

//如果it不存在于map中，则新建去shader中获取

if (it == m\_mUniformTextures.end())

{

UniformTexture texture;

texture.location = glGetUniformLocation(m\_Program, nameInShader);

if (texture.location != \_INVALID\_LOCATION\_)

{

//去读图片

texture.texture = ResourceManager::GetPic(imagePath, isRepeat);

if (texture.texture != \_INVALID\_ID\_)

{

m\_mUniformTextures.insert(std::pair<string, UniformTexture>(nameInShader, texture));

}

}

}

//如果存在，则认为想要替换图片

else

{

GLuint texture = ResourceManager::GetPic(imagePath);

if (texture!= \_INVALID\_ID\_)

{

ResourceManager::RemovePic(it->second.texture);

it->second.texture = texture;

}

}

}

设置vec类型（vec2，vec3和vec4类似）

void Shader::SetVec4(const char\* nameInShader, float x, float y, float z, float w)

{

INIT\_TEST\_VOID

auto it = m\_mUniformVec4.find(nameInShader);

//如果it不存在于map中，则新建去shader中获取

if (it == m\_mUniformVec4.end())

{

UniformVec4 vec4;

vec4.location = glGetUniformLocation(m\_Program, nameInShader);

if (vec4.location != \_INVALID\_LOCATION\_)

{

vec4.v[0] = x;

vec4.v[1] = y;

vec4.v[2] = z;

vec4.v[3] = w;

m\_mUniformVec4.insert(std::pair<string, UniformVec4>(nameInShader,vec4));

}

}

else

{

it->second.v[0] = x;

it->second.v[1] = y;

it->second.v[2] = z;

it->second.v[3] = w;

}

}

设置float类型

void Shader::SetFloat(const char\* nameInShader, float value)

{

INIT\_TEST\_VOID

auto it = m\_mUniformFloats.find(nameInShader);

//如果it不存在于map中，则新建去shader中获取

if (it == m\_mUniformFloats.end())

{

UniformFloat vec4;

vec4.location = glGetUniformLocation(m\_Program, nameInShader);

if (vec4.location != \_INVALID\_LOCATION\_)

{

vec4.value = value;

m\_mUniformFloats.insert(std::pair<string, UniformFloat>(nameInShader, vec4));

}

}

else

{

it->second.value = value;

}

}

设置Matrix类型（mat2、mat3和mat4类似）

void Shader::SetMatrix(const char\* nameInShader, const glm::mat4& matrix)

{

INIT\_TEST\_VOID

auto it = m\_mUniformMatrixs.find(nameInShader);

//如果it不存在于map中，则新建去shader中获取

if (it == m\_mUniformMatrixs.end())

{

UniformMatrix mat;

mat.location = glGetUniformLocation(m\_Program, nameInShader);

if (mat.location != \_INVALID\_LOCATION\_)

{

mat.value = matrix;

m\_mUniformMatrixs.insert(std::pair<string, UniformMatrix>(nameInShader, mat));

}

}

else

{

it->second.value = matrix;

}

}

设置CubeMap类型

void Shader::SetCueMap(const char\* front, const char\* back, const char\* top, const char\* bottom, const char\* left, const char\* right, const char\* nameInShader)

{

INIT\_TEST\_VOID

string name(nameInShader);

auto it = m\_mUniformCubeMap.find(name);

if (it == m\_mUniformCubeMap.end())

{

//获取Location

UniformTexture texture;

texture.location = glGetUniformLocation(m\_Program, nameInShader);

if (texture.location != \_INVALID\_LOCATION\_)

{

texture.texture = ResourceManager::GetTextureCube(front, back, top, bottom, left, right);

if (texture.texture != \_INVALID\_ID\_)

{

m\_mUniformCubeMap.insert(std::pair<string, UniformTexture>(name, texture));

}

}

}

else

{

//如果有

ResourceManager::RemoveTextureCube(it->second.texture);

it->second.texture = ResourceManager::GetTextureCube(front, back, top, bottom, left, right);

}

}

在第三章——引擎架构的时候，为Shader类设计了若干个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。具体思路是：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。

③将内存数据传递到显存

void Shader::Bind(const float \*M, const float\* V, const float \*P, const float \*IT )

{

INIT\_TEST\_VOID

if (m\_ModelMatrixLocation != \_INVALID\_LOCATION\_ && M != nullptr)

{

glUniformMatrix4fv(this->m\_ModelMatrixLocation, 1, GL\_FALSE, M);

}

if (m\_ViewMatrixLocation != \_INVALID\_LOCATION\_ && V != nullptr)

{

glUniformMatrix4fv(this->m\_ViewMatrixLocation, 1, GL\_FALSE, V);

}

if (m\_ProjactionMatrixLocation != \_INVALID\_LOCATION\_ && P != nullptr)

{

glUniformMatrix4fv(this->m\_ProjactionMatrixLocation, 1, GL\_FALSE, P);

}

if (m\_ITModelMatrixLocation != \_INVALID\_LOCATION\_ && IT != nullptr)

{

glUniformMatrix4fv(this->m\_ITModelMatrixLocation, 1, GL\_FALSE, IT);

}

//map不为空，则开始贴纹理

if (m\_mUniformTextures.size() > 0)

{

for (auto it = m\_mUniformTextures.begin(); it != m\_mUniformTextures.end(); it++)

{

glUniform1i(it->second.location, 0);

glBindTexture(GL\_TEXTURE\_2D, it->second.texture);

}

}

if (m\_mUniformCubeMap.size() > 0)

{

for (auto it = m\_mUniformCubeMap.begin(); it != m\_mUniformCubeMap.end(); it++)

{

glUniform1i(it->second.location, 0);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, it->second.texture);

}

}

if (m\_mUniformVec4.size() > 0)

{

for (auto it = m\_mUniformVec4.begin(); it != m\_mUniformVec4.end(); it++)

{

glUniform4fv(it->second.location, 1, it->second.v);

}

}

if (m\_mUniformVec3.size() > 0)

{

for (auto it = m\_mUniformVec3.begin(); it != m\_mUniformVec3.end(); it++)

{

glUniform3fv(it->second.location, 1, it->second.v);

}

}

if (m\_mUniformFloats.size() > 0)

{

for (auto it = m\_mUniformFloats.begin(); it != m\_mUniformFloats.end(); it++)

{

glUniform1f(it->second.location,it->second.value);

}

}

if (m\_mUniformMatrixs.size() > 0)

{

for (auto it = m\_mUniformMatrixs.begin(); it != m\_mUniformMatrixs.end(); it++)

{

glUniformMatrix4fv(it->second.location, 1, GL\_FALSE, glm::value\_ptr(it->second.value));

}

}

//告诉GPU,怎么样去读取vbo的内存块

//启用该插槽

if (m\_PositionLocation != \_INVALID\_LOCATION\_)

{

glEnableVertexAttribArray(this->m\_PositionLocation);

glVertexAttribPointer(m\_PositionLocation, 4, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), 0);//插槽的位置，插槽中数据有几个分量(x,y,z,w)，每个分量是什么类型，是否归一化，两个点之间的距离，设置的信息从vbo的啥地方开始取值

}

if (m\_ColorLocation != \_INVALID\_LOCATION\_)

{

glEnableVertexAttribArray(this->m\_ColorLocation);

glVertexAttribPointer(m\_ColorLocation, 4, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)(sizeof(float) \* 4));

}

if (m\_TexcoordLocation != \_INVALID\_LOCATION\_)

{

glEnableVertexAttribArray(this->m\_TexcoordLocation);

glVertexAttribPointer(this->m\_TexcoordLocation, 4, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)(sizeof(float) \* 8));

}

if (m\_NormalLocation != \_INVALID\_LOCATION\_)

{

glEnableVertexAttribArray(this->m\_NormalLocation);

glVertexAttribPointer(this->m\_NormalLocation, 4, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)(sizeof(float) \* 12));

}

}

该函数的功能是将内存中的数据传递到GPU使得Shader嫩够被正确运行，被更新的数据包括：attribute顶点数据，uniform变量。其中uniform变量包括最通用的MVP矩阵和IT矩阵，以及开发者自行设置的变量。

### 4.4.3 RenderAble

RendAble是所有可绘制对象的基类，它抽象出所有可绘制对象共同的特点。

成员变量回顾：

class RenderAble:public Node

bool m\_IsDraw = 1;

RenderOption m\_Options;//渲染选项

VertexBuffer m\_VertexBuf;//VBO

Shader m\_Shader;//Shader

glm::mat4 m\_ModelMatrix;

①清理资源

void RenderAble::Destory()

{

INIT\_TEST\_VOID;

m\_IsInit = 0;

m\_Shader.Destory();

m\_VertexBuf.Destory();

}

清理资源函数的定义是个虚函数，它可以被开发者重写，使得能够清理自定义的游戏物体的资源，默认的实现需要去做Shader、VertexBuffer的清理工作。

②位置变换

void RenderAble::SetPosition(float x, float y, float z)

{

Node::SetPosition(x, y, z);

UpdateTransform();

}

void RenderAble::SetPosition(const vec3& pos)

{

Node::SetPosition(pos.x, pos.y, pos.z);

UpdateTransform();

}

void RenderAble::SetRotate(float x, float y, float z)

{

Node::SetRotate(x, y, z);

UpdateTransform();

}

void RenderAble::SetRotate(const vec3& rot)

{

Node::SetRotate(rot.x, rot.y, rot.z);

UpdateTransform();

}

void RenderAble::SetScale(float x, float y, float z)

{

Node::SetScale(x, y, z);

UpdateTransform();

}

void RenderAble::SetScale(const vec3& scale)

{

Node::SetScale(scale.x, scale.y, scale.z);

UpdateTransform();

}

void RenderAble::UpdateTransform()

{

m\_ModelMatrix = glm::translate(m\_Transform.m\_Position);

m\_ModelMatrix = glm::rotate(m\_ModelMatrix, m\_Transform.m\_Rotate.x, vec3(1, 0, 0));

m\_ModelMatrix = glm::rotate(m\_ModelMatrix, m\_Transform.m\_Rotate.y, vec3(0, 1, 0));

m\_ModelMatrix = glm::rotate(m\_ModelMatrix, m\_Transform.m\_Rotate.z, vec3(0, 0, 1));

m\_ModelMatrix \*= glm::scale(m\_Transform.m\_Scale);

}

需要绘制的对象与普通对象的不同在于，需要绘制的对象需要持有一个矩阵以描述自己在世界坐标系的位置，也就是说所有位置、旋转、缩放变换对于RenderAble对象而言不仅仅是改变自己的位置信息的值，更重要的是改变自己的ModelMatrix。RenderAble类重写了Node类的位置变换函数，在其中添加了矩阵的变换。

③RenderOption

渲染选项是让RenderList读的，也就是描述这样一个问题：渲染这个游戏物体，需要哪些全局状态的支持。比如绘制粒子系统需要让程序指定点的大小，绘制GameObject需要设置图元是三角形等等。

struct RenderOption

{

DRAW\_TYPE DrawType = DRAW\_TRIANGLES;//所需的图元

bool DepthTest = 0;//深度测试信息

AlphaBlendInfo alphaBlend;//alpha混合信息

ProgramPointSize Program\_Point\_Size;//点大小信息

...//其它绘制选项可随时补充

};

### 4.4.4 RenderAble\_Light

需要受到光照的游戏物体会继承RenderAble\_Light类，该类需要描述材质和灯光信息，使得Shader能够正常渲染物体的受光效果。RenderAble\_Light定义了Shader中与光照相关的变量命名规范，按规范命名的Shader光照将是正常的。

成员变量回顾：

Material m\_Material;

①描述物体材质

void RenderAble\_Light::SetAmbientMaterial(float r, float g, float b, float a)

{

INIT\_TEST\_VOID

m\_Material.SetAmbientMaterial(vec4(r, g, b, a));

m\_Shader.SetVec4("U\_AmbientMaterial", r, g, b, a);

}

void RenderAble\_Light::SetDiffuseMaterial(float r, float g, float b, float a)

{

INIT\_TEST\_VOID

m\_Material.SetDiffuseMaterial(vec4(r, g, b, a));

m\_Shader.SetVec4("U\_DiffuseMaterial", r, g, b, a);

}

void RenderAble\_Light::SetSpecularMaterial(float r, float g, float b, float a)

{

INIT\_TEST\_VOID

m\_Material.SetSepcularMaterial(vec4(r, g, b, a));

m\_Shader.SetVec4("U\_SpecularMaterial", r, g, b, a);

}

物体材质的描述包括内存中的变量和Shader中的变量，这里规定了Shader中的材质变量名，使用该命名的Shader能够被正确调用。

②描述光源

void RenderAble\_Light::SetLight\_1(const Light& light1)

{

INIT\_TEST\_VOID

switch (light1.GetType())

{

case 0:

break;

case 1://方向光

m\_Shader.SetVec4("U\_Light1\_Dir", vec4(light1.GetRotate(), 0));

m\_Shader.SetVec4("U\_Light1\_Ambient", light1.GetAmbientColor());

m\_Shader.SetVec4("U\_Light1\_Diffuse", light1.GetDiffuseColor());

m\_Shader.SetVec4("U\_Light1\_Specular", light1.GetSpecularColor());

m\_Shader.SetVec4("U\_Light1\_Opt", 1, 0, 0, 32);

break;

case 2://点光源

m\_Shader.SetVec4("U\_Light1\_Pos", vec4(light1.GetPosition(), 1));

m\_Shader.SetVec4("U\_Light1\_Ambient", light1.GetAmbientColor());

m\_Shader.SetVec4("U\_Light1\_Diffuse", light1.GetDiffuseColor());

m\_Shader.SetVec4("U\_Light1\_Specular", light1.GetSpecularColor());

m\_Shader.SetFloat("U\_Light1\_Constant", light1.GetConstAttenuation());

m\_Shader.SetFloat("U\_Light1\_Linear", light1.GetLinearAttenuation());

m\_Shader.SetFloat("U\_Light1\_Quadric", light1.GetLinearAttenuation());

m\_Shader.SetVec4("U\_Light1\_Opt", 2, 0, 0, 32);

break;

case 3://聚光灯

m\_Shader.SetVec4("U\_Light1\_Pos", vec4(light1.GetPosition(), 1));

m\_Shader.SetVec4("U\_Light1\_Dir", vec4(light1.GetRotate(), 1));

m\_Shader.SetVec4("U\_Light1\_Ambient", light1.GetAmbientColor());

m\_Shader.SetVec4("U\_Light1\_Diffuse", light1.GetDiffuseColor());

m\_Shader.SetVec4("U\_Light1\_Specular", light1.GetSpecularColor());

m\_Shader.SetFloat("U\_Light1\_Constant", light1.GetConstAttenuation());

m\_Shader.SetFloat("U\_Light1\_Linear", light1.GetLinearAttenuation());

m\_Shader.SetFloat("U\_Light1\_Quadric", light1.GetLinearAttenuation());

m\_Shader.SetFloat("U\_Light1\_CutOff", light1.GetCutoff());

m\_Shader.SetFloat("U\_Light1\_Exponent", light1.GetExponent());

m\_Shader.SetVec4("U\_Light1\_Opt", 3, 0, 0, 32);

default:

break;

}

}

这里以描述光源1举例，首先获取光源类型，因为特定类型的光源只需要向GPU传递某些属性。然后再根据光源类型获取对应光的属性，并传到GPU。

## 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 光照Shader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，引擎实现了下述的对方向光、点光源、聚光灯颜色计算的算法。

所需参数：

uniform vec4 U\_AmbientMaterial;

uniform vec4 U\_DiffuseMaterial;

uniform vec4 U\_SpecularMaterial;

uniform vec4 U\_Light1\_Pos;

uniform vec4 U\_Light1\_Dir;

uniform vec4 U\_Light1\_Ambient;

uniform vec4 U\_Light1\_Diffuse;

uniform vec4 U\_Light1\_Specular;

uniform vec4 U\_Light1\_Opt;

uniform float U\_Light1\_Constant;

uniform float U\_Light1\_Linear;

uniform float U\_Light1\_Quadric;

uniform float U\_Light1\_CutOff;

uniform float U\_Light1\_Exponent;

uniform vec4 U\_CameraPos;

①方向光颜色计算函数

vec4 GetDirectionLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightDir,vec4 lightOpt)

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec4 ambientColor=lightAmbient\*U\_AmbientMaterial;

vec3 L=lightDir.xyz;

L=normalize(L);

vec3 N=normalize(V\_Normal.xyz);

float diffuseIntensity=max(0.0,dot(L,N));

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*diffuseIntensity;

vec4 specularColor=vec4(0.0,0.0,0.0,1.0);

if(diffuseIntensity!=0.0)

{

vec3 reflectDirection=normalize(reflect(-L,N));

vec3 viewDirection=normalize(U\_CameraPos.xyz-V\_WorldPos.xyz);

specularColor=lightSpecular\*U\_SpecularMaterial\*pow(max(0.0,dot(viewDirection,reflectDirection)),lightOpt.w);

}

color=ambientColor+diffuseColor+specularColor;

return color;

}

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。

②点光源颜色计算函数

vec4 GetPointLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightPos,vec4 lightOpt,float constant,float linear,float quadric)

{

vec4 ambientColor=lightAmbient\*U\_AmbientMaterial;

vec3 L=lightPos.xyz-V\_WorldPos.xyz;

float distance=length(L);

float attenuation=1.0/(constant+linear\*distance+distance\*quadric\*quadric);//衰减系数

L=normalize(L);

vec3 N=normalize(V\_Normal.xyz);

float diffuseIntensity=max(0.0,dot(L,N));

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*diffuseIntensity\*attenuation\*2.0;

vec4 specularColor=vec4(0.0,0.0,0.0,0.0);

if(diffuseIntensity!=0.0)

{

vec3 reflectDirection=normalize(reflect(-L,N));

vec3 viewDirection=normalize(U\_CameraPos.xyz-V\_WorldPos.xyz);

specularColor=lightSpecular\*U\_SpecularMaterial\*pow(max(0.0,dot(viewDirection,reflectDirection)),lightOpt.w);

}

vec4 color=ambientColor+diffuseColor+specularColor;

return color;

}

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。

③聚光灯颜色计算函数

vec4 GetSpotLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightPos,vec4 lightDir,vec4 lightOpt,float constant,float linear,float quadric,float exponent,float cutOff)

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec3 L=lightPos.xyz-V\_WorldPos.xyz;//光指向人的向量

float distance=length(L);

float attenuation=1.0/(constant+linear\*distance+distance\*quadric\*quadric);//衰减系数

L=normalize(L);

vec3 N=normalize(V\_Normal.xyz);

float diffuseIntensity=max(0.0,dot(L,N));//反射光角度的余弦

//计算衰减

if(diffuseIntensity>0.0)

{

float cos\_ObjLight=max(0.0,dot(-L,normalize(lightDir.xyz)));//人和光的夹角余弦

float cos\_CutOff=cos(cutOff\*3.14/180.0);//最大可见光的余弦

float cos\_Exponent=cos(exponent\*3.14/180.0);//不衰减光的余弦

//如果在不变域

if(cos\_ObjLight>cos\_Exponent)

{

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*attenuation;

vec3 reflectDirection=normalize(reflect(-L,N));

vec3 viewDirection=normalize(U\_CameraPos.xyz-V\_WorldPos.xyz);

vec4 specularColor=lightSpecular\*U\_SpecularMaterial\*pow(max(0.0,dot(viewDirection,reflectDirection)),lightOpt.w);

color=lightAmbient\*U\_AmbientMaterial+diffuseColor+specularColor;

}

//如果角度小于衰减范围，说明在衰减域

else if(cos\_ObjLight>cos\_CutOff)

{

diffuseIntensity=pow(cos\_ObjLight,U\_Light1\_Opt.w);

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*attenuation\*diffuseIntensity\*2.0;

vec3 reflectDirection=normalize(reflect(-L,N));

vec3 viewDirection=normalize(U\_CameraPos.xyz-V\_WorldPos.xyz);

vec4 specularColor=lightSpecular\*U\_SpecularMaterial\*pow(max(0.0,dot(viewDirection,reflectDirection)),lightOpt.w);

color=lightAmbient\*U\_AmbientMaterial+diffuseColor+specularColor;

}

//如果不可见，不对color做任何处理

}

return color;

}

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。

### 4.5.2 给物体表面贴上纹理

①贴2D纹理

uniform sampler2D U\_Texture\_1;

vec4 GetTexture1()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec4 textureColor=texture2D(U\_Texture\_1,V\_Texcoord.xy);

if(textureColor!=vec4(0.0,0.0,0.0,1.0))

{

color=textureColor;

}

return color;

}

一般一个物体只需要一张纹理（可以理解为一个人只有一层表面皮肤，而不是两层皮肤颜色混合），这里实现了对一张纹理的贴图，贴纹理使用了GLSL内置的函数texture2D（sampler2D ,vec2），该函数将sampler2D 类型纹理正确地贴在物体表面。

②贴立方体纹理

uniform samplerCube U\_Texture\_CubeMap;

vec4 GetTexture1()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec4 textureColor=texture(U\_Texture\_CubeMap,normalize(V\_Texcoord));

if(textureColor!=vec4(0.0,0.0,0.0,1.0))

{

color=textureColor;

}

return color;

}

可以将立方体纹理想想为将一个立方体盒子从地面开始展开，平铺。

贴立方体纹理使用了GLSL内置函数texture（samplerCube,vec3），该函数将一个CubeMap正确地贴在立方体的内表面。

### 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。

①地面初始化

bool Ground::Init()

{

if (!m\_IsInit)

{

m\_IsInit = 1;

m\_VertexBuf.Init(4);

//起始坐标

int zStart = 100;

int xStart = -100;

int length = 100;

m\_VertexBuf.SetPosition(0, xStart, height, zStart);

m\_VertexBuf.SetPosition(1, xStart + length, height, zStart);

m\_VertexBuf.SetPosition(2, xStart, height, zStart - length);

m\_VertexBuf.SetPosition(3, xStart + length, height, zStart - length);

m\_VertexBuf.SetNormal(0, 0, 1, 0);

m\_VertexBuf.SetNormal(1, 0, 1, 0);

m\_VertexBuf.SetNormal(2, 0, 1, 0);

m\_VertexBuf.SetNormal(3, 0, 1, 0);

m\_VertexBuf.SetColor(0, 0.4f, 0.4f, 0.4f);

m\_VertexBuf.SetColor(1, 0.4f, 0.4f, 0.4f);

m\_VertexBuf.SetColor(2, 0.4f, 0.4f, 0.4f);

m\_VertexBuf.SetColor(3, 0.4f, 0.4f, 0.4f);

m\_Shader.Init("res/ground.vert", "res/ground.frag");

SetAmbientMaterial(0.1f, 0.1f, 0.1f, 1);

SetDiffuseMaterial(0.6f, 0.6f, 0.6f, 1);

SetSpecularMaterial(0, 0, 0, 0);

return 1;

}

return 0;

}

bool Ground::Init(const char\* picName)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

m\_VertexBuf.Init(4);

//起始坐标

int size = 100;

int rept = 100;

m\_VertexBuf.SetPosition(0, -size, -0.5f, size);

m\_VertexBuf.SetTexcoord(0, 0, 0);

m\_VertexBuf.SetPosition(1, size, -0.5f, size);

m\_VertexBuf.SetTexcoord(1, rept, 0);

m\_VertexBuf.SetPosition(2, -size, height, -size);

m\_VertexBuf.SetTexcoord(2, rept, rept);

m\_VertexBuf.SetPosition(3, size, height, -size);

m\_VertexBuf.SetTexcoord(3, 0, rept);

m\_VertexBuf.SetNormal(0, 0, 1, 0);

m\_VertexBuf.SetNormal(1, 0, 1, 0);

m\_VertexBuf.SetNormal(2, 0, 1, 0);

m\_VertexBuf.SetNormal(3, 0, 1, 0);

m\_VertexBuf.SetColor(0, 1, 1, 1, 1);

m\_VertexBuf.SetColor(1, 1, 1, 1, 1);

m\_VertexBuf.SetColor(2, 1, 1, 1, 1);

m\_VertexBuf.SetColor(3, 1, 1, 1, 1);

m\_Shader.Init("res/ground.vert", "res/ground.frag");

SetTexture2D(picName);

SetAmbientMaterial(1, 1, 1);

SetDiffuseMaterial(0.2f, 0.2f, 0.2f);

SetSpecularMaterial(0, 0, 0);

m\_Options.DepthTest = 1;

m\_Options.DrawType = DRAW\_TRIANGLES\_STRIP;

return 1;

}

return 0;

}

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。

②地面VertexShader

VertexShader

attribute vec4 position;

attribute vec4 color;

attribute vec4 normal;

attribute vec4 texcoord;

uniform mat4 ModelMatrix;

uniform mat4 ViewMatrix;

uniform mat4 ProjectionMatrix;

uniform mat4 IT\_ModelMatrix;

varying vec4 V\_Color;

varying vec2 V\_Texcoord;

varying vec3 V\_WorldPos;

varying vec3 V\_Normal;

void main()

{

V\_Color=color;

V\_Normal=(IT\_ModelMatrix\*normal).xyz;

V\_Texcoord=texcoord.xy;

V\_WorldPos=(ModelMatrix\*position).xyz;

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*position;

}

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。V\_WorldPos表示物体在世界坐标系的位置，将物体的模型矩阵乘上pos信息就可以拿到物体在世界坐标系的坐标。gl\_Position是GLSL内置变量，用于描述顶点位置，99%的游戏物体只需要让gl\_Position等于MVP矩阵乘上物体Pos即可。

③地面FragmentShader

（为避免重复省略了与光照、贴图相关参数和功能函数）

#ifdef GL\_ES

precision mediump float

#endif

varying vec4 V\_Color;

varying vec2 V\_Texcoord;

varying vec3 V\_WorldPos;

varying vec3 V\_Normal;

//光源1

vec4 GetLight1Color()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

//direction

if(U\_Light1\_Opt.x==1.0)

{

color=GetDirectionLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Pos,U\_Light1\_Opt);

}

//point

else if(U\_Light1\_Opt.x==2.0)

{

color=GetPointLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Pos,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric);

}

//spot

else if(U\_Light1\_Opt.x==3.0)

{

color=GetSpotLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Pos,U\_Light1\_Dir,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric,U\_Light1\_Exponent,U\_Light1\_CutOff);

}

return color;

}

void main()

{

vec4 lightColor=GetLight1Color();

vec4 textureColor=texture2D(U\_Texture\_1,V\_Texcoord.xy);

gl\_FragColor = textureColor\*V\_Color\*lightColor;

}

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4.5.2讲述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。gl\_FragColor是GLSL的内置变量，用于输出这个像素点的颜色。

### 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。

成员变量回顾：

class GameObject :public RenderAble\_Light

string m\_ModelName;

①初始化/析构模型数据

bool GameObject::Init(const char\* path, const char\* vertexShader, const char\* fragmentShader)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

if (ResourceManager::GetModel(path, m\_VertexBuf))

{

m\_ModelName = path;

m\_Shader.Init(vertexShader, fragmentShader);

SetAmbientMaterial(vec4(0.1f, 0.1f, 0.1f, 1.0f));

SetDiffuseMaterial(vec4(0.6f, 0.6f, 0.6f, 1.0f));

SetSpecularMaterial(vec4(1, 1, 1, 1.0f));

m\_Options.DepthTest = 1;

m\_Options.DrawType = DRAW\_TRIANGLES;

return 1;

}

}

return 0;

}

void GameObject::Destory()

{

INIT\_TEST\_VOID

RenderAble::Destory();

ResourceManager::RemoveModel(m\_ModelName);

m\_IsInit = 0;

}

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。

③Phong光照算法-逐顶点光照

/..省略对3种类型光颜色的抽象算法，下面给出顶点着色器的其余部分的代码../

#ifdef GL\_ES

precision mediump float;

#endif

attribute vec4 position;

attribute vec4 texcoord;

attribute vec4 normal;

uniform mat4 ModelMatrix;

uniform mat4 ViewMatrix;

uniform mat4 ProjectionMatrix;

uniform mat4 IT\_ModelMatrix;

uniform vec4 U\_AmbientMaterial;

uniform vec4 U\_DiffuseMaterial;

uniform vec4 U\_SpecularMaterial;

uniform vec4 U\_CameraPos;

uniform vec4 U\_Light1\_Pos;

uniform vec4 U\_Light1\_Dir;

uniform vec4 U\_Light1\_Ambient;

uniform vec4 U\_Light1\_Diffuse;

uniform vec4 U\_Light1\_Specular;

uniform vec4 U\_Light1\_Opt;

uniform float U\_Light1\_Constant;

uniform float U\_Light1\_Linear;

uniform float U\_Light1\_Quadric;

uniform float U\_Light1\_CutOff;

uniform float U\_Light1\_Exponent;

varying vec4 V\_Color;

varying vec4 V\_Texcoord;

vec4 GetLight1Color()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

//direction

if(U\_Light1\_Opt.x==1.0)

{

color=GetDirectionLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Dir,U\_Light1\_Opt);

}

//point

else if(U\_Light1\_Opt.x==2.0)

{

color=GetPointLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Pos,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric);

}

//spot

else if(U\_Light1\_Opt.x==3.0)

{

color=GetSpotLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Pos,U\_Light1\_Dir,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric,U\_Light1\_Exponent,U\_Light1\_CutOff);

}

return color;

}

void main()

{

V\_Texcoord=texcoord;

V\_Color=GetLight1Color();

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*position;

}

可以看到逐顶点光照的顶点着色器和逐像素光照的片元着色器代码类似，唯一区别只在于计算的位置不同，将光照的计算放在顶点着色器能大幅提高运行效率。

④Blin光照算法-逐顶点光照

//..省略逐顶点光照的和③相同的部分代码，下面给出顶点着色器的光照计算部分的代码../

//获取任意方向光的光强

vec4 GetDirectionLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightDir,vec4 lightOpt)

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec4 ambientColor=lightAmbient\*U\_AmbientMaterial;

vec3 L=normalize((IT\_ModelMatrix\*lightDir).xyz);

vec3 N=normalize(normal.xyz);

float diffuseIntensity=max(0.0,dot(L,N));

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*diffuseIntensity;

vec4 specularColor=vec4(0.0,0.0,0.0,1.0);

if(diffuseIntensity!=0.0)

{

vec3 worldPos=(ModelMatrix\*position).xyz;

vec3 viewDirection=normalize(U\_CameraPos.xyz-worldPos.xyz);//视线方向

vec3 halfDir=normalize(L+viewDirection);//用view和入射光线的半角模拟反射方向

specularColor=U\_Light1\_Specular\*U\_SpecularMaterial\*pow(max(0.0,dot(N,halfDir)),lightOpt.w);

}

color=ambientColor+diffuseColor+specularColor;

return color;

}

//获取任意点光源颜色

vec4 GetPointLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightPos,vec4 lightOpt,float constant,float linear,float quadric)

{

vec4 ambientColor=lightAmbient\*U\_AmbientMaterial;

vec3 L=(IT\_ModelMatrix\*lightPos).xyz;

float distance=length(L);

L=normalize(L);

vec3 N=normalize(normal.xyz);

float attenuation=1.0/(constant+linear\*distance+distance\*quadric\*quadric);//衰减系数

float diffuseIntensity=max(0.0,dot(L,N));

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*diffuseIntensity\*attenuation\*2.0;

vec4 specularColor=vec4(0.0,0.0,0.0,0.0);

if(diffuseIntensity!=0.0)

{

vec3 worldPos=(ModelMatrix\*position).xyz;

vec3 viewDirection=normalize(U\_CameraPos.xyz-worldPos.xyz);//视线方向

vec3 halfDir=normalize(L+viewDirection);//用view和入射光线的半角模拟反射方向

specularColor=U\_Light1\_Specular\*U\_SpecularMaterial\*pow(max(0.0,dot(N,halfDir)),lightOpt.w);

}

vec4 color=ambientColor+diffuseColor+specularColor;

return color;

}

//获取任意聚光灯颜色

vec4 GetSpotLightColor(vec4 lightAmbient,vec4 lightDiffuse,vec4 lightSpecular,vec4 lightPos,vec4 lightDir,vec4 lightOpt,float constant,float linear,float quadric,float exponent,float cutOff)

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec3 L=(IT\_ModelMatrix\*lightDir).xyz;

float distance=length(L);

float attenuation=1.0/(constant+linear\*distance+distance\*quadric\*quadric);//衰减系数

L=normalize(L);

vec3 N=normalize(normal.xyz);

float diffuseIntensity=max(0.0,dot(L,N));//反射光角度的余弦

//计算衰减

if(diffuseIntensity>0.0)

{

float cos\_ObjLight=max(0.0,dot(-L,normalize(lightDir.xyz)));//人和光的夹角余弦

float cos\_CutOff=cos(cutOff\*3.14/180.0);//最大可见光的余弦

float cos\_Exponent=cos(exponent\*3.14/180.0);//不衰减光的余弦

//如果在不变域

if(cos\_ObjLight>cos\_Exponent)

{

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*attenuation;

vec3 worldPos=(ModelMatrix\*position).xyz;

vec3 viewDirection=normalize(U\_CameraPos.xyz-worldPos.xyz);//视线方向

vec3 halfDir=normalize(L+viewDirection);//用view和入射光线的半角模拟反射方向

vec4 specularColor=U\_Light1\_Specular\*U\_SpecularMaterial\*pow(max(0.0,dot(N,halfDir)),lightOpt.w);

color=lightAmbient\*U\_AmbientMaterial+diffuseColor+specularColor;

}

//如果角度小于衰减范围，说明在衰减域

else if(cos\_ObjLight>cos\_CutOff)

{

diffuseIntensity=pow(cos\_ObjLight,lightOpt.w);

vec4 diffuseColor=lightDiffuse\*U\_DiffuseMaterial\*attenuation\*diffuseIntensity\*2.0;

vec3 worldPos=(ModelMatrix\*position).xyz;

vec3 viewDirection=normalize(U\_CameraPos.xyz-worldPos.xyz);//视线方向

vec3 halfDir=normalize(L+viewDirection);//用view和入射光线的半角模拟反射方向

vec4 specularColor=U\_Light1\_Specular\*U\_SpecularMaterial\*pow(max(0.0,dot(N,halfDir)),lightOpt.w);

color=lightAmbient\*U\_AmbientMaterial+diffuseColor+specularColor;

}

//如果不可见，不对color做任何处理

}

return color;

}

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。

⑤Blin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。

### 4.5.5 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;//目标视点

glm::vec3 m\_Up;//发射出去的方向向量

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

Camera\_1st::Camera\_1st() :m\_Position(4, 4, 4), m\_ViewCenter(0, 0, -1), m\_Up(0, 1, 0), m\_ViewportWidget(WINDOW\_WIDTH), m\_ViewportHeight(WINDOW\_HEIGHT)

{

m\_ViewMatrix = glm::lookAt(this->m\_Position, this->m\_ViewCenter, this->m\_Up);

this->m\_ProjectionMatrix = glm::perspective(this->m\_Radius, this->m\_ViewportWidget / this->m\_ViewportHeight, this->m\_Near, this->m\_Far);

}

摄像机的构造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，初始化之后的摄像机能看见东西，但是不支持移动，下面给出摄像机移动的支持。

②摄像机逻辑更新——上下左右移动

void Camera\_1st::Update()

{

float frameTime = Time::DeltaTime();

//通过前方的方向向量移动，方向为视点-当前位置

vec3 forwardDirection = glm::normalize(m\_ViewCenter - m\_Position);

//右边：前方和向上的叉乘

vec3 rightDirection = glm::normalize(glm::cross(forwardDirection, m\_Up));

//上方：头顶方向 UP

//摄像机向右移动，移动的是一个向量

if (m\_IsMoveToRight)

{

vec3 delta = rightDirection\*frameTime\*m\_MoveSpeed;

m\_Position += delta;

m\_ViewCenter += delta;

}

if (m\_IsMoveToLeft)

{

vec3 delta = rightDirection\*frameTime\*m\_MoveSpeed;

m\_Position -= delta;

m\_ViewCenter -= delta;

}

if (m\_IsMoveToTop)

{

vec3 delta = m\_Up\*frameTime\*m\_MoveSpeed;

m\_Position+= delta;

m\_ViewCenter+= delta;

}

if (m\_IsMoveToBottom)

{

vec3 delta = m\_Up\*frameTime\*m\_MoveSpeed;

m\_Position -= delta;

m\_ViewCenter -= delta;

}

this->m\_ViewMatrix = glm::lookAt(m\_Position, m\_ViewCenter, m\_Up);

}

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。

③视口远近移动

void Camera\_1st::MoveToFront()

{

vec3 forwardDirection = glm::normalize(m\_ViewCenter - m\_Position);

vec3 delta = forwardDirection\*0.1f\*m\_MoveSpeed;

m\_Position += delta;

m\_ViewCenter += delta;

}

void Camera\_1st::MoveToBack()

{

vec3 forwardDirection = glm::normalize(m\_ViewCenter - m\_Position);

vec3 delta = forwardDirection\*0.1f\*m\_MoveSpeed;

m\_Position -= delta;

m\_ViewCenter -= delta;

}

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。

④摄像机旋转

void Camera\_1st::Pitch(float angle)

{

vec3 viewDirection = m\_ViewCenter - m\_Position;

viewDirection = glm::normalize(viewDirection);

vec3 rightDirection = glm::cross(viewDirection, m\_Up);

rightDirection=glm::normalize(rightDirection);

RotateView(angle, rightDirection.x, rightDirection.y, rightDirection.z);

}

void Camera\_1st::Yaw(float angle)

{

RotateView(angle, m\_Up.x, m\_Up.y, m\_Up.z);

}

Pitch方法绕x轴正方向（右手边）旋转，Yaw方法绕y轴正方向（头顶）旋转，区别只在于传入的参数不同。下面会介绍的RotateView函数绕任意轴旋转angle角。

故Pitch函数需要计算出摄像机的x轴正方向的方向向量，传入RotateView函数，计算方法是：x轴正方向的方向向量与y轴和视线方向所在平面垂直，故将y轴与视线做叉乘得到右手边方向。

void Camera\_1st::RotateView(float angle, float x, float y, float z) {

vec3 viewDirection = m\_ViewCenter - m\_Position;

vec3 newDirection(0.0f, 0.0f, 0.0f);

float C = cosf(angle);

float S = sinf(angle);

vec3 tempX(C + x\*x\*(1 - C), x\*y\*(1 - C) - z\*S, x\*z\*(1 - C) + y\*S);

newDirection.x = glm::dot(tempX, viewDirection);

vec3 tempY(x\*y\*(1 - C) + z\*S, C + y\*y\*(1 - C), y\*z\*(1 - C) - x\*S);

newDirection.y = glm::dot(tempY, viewDirection);

vec3 tempZ(x\*z\*(1 - C) - y\*S, y\*z\*(1 - C) + x\*S, C + z\*z\*(1 - C));

newDirection.z = glm::dot(tempZ, viewDirection);

m\_ViewCenter = m\_Position + newDirection;

}

RotateView函数实际上就是讲所给的轴代入公式进行计算，然后将计算出来的新的方向向量赋值给视点位置即可。

### 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，就是一个盒子然后直接把贴图贴在盒子内表面，这样的话需要向GPU提交6次顶点数据。

6面离散形式天空盒成员变量：

VertexBuffer m\_VertexBuf[6];//6个面,顺序前后上下左右

Shader m\_Shader[6];

glm::mat4 m\_ModelMatrix;

bool m\_IsInit = 0;

①初始化

bool SkyBox::Init(const char\* forwardPath, const char\* backPath, const char\* topPath, const char\* bottomPath, const char\* leftPath, const char\* rightPath)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

this->SetForward(forwardPath);

this->SetBack(backPath);

this->SetTop(topPath);

this->SetBottom(bottomPath);

this->SetLeft(leftPath);

this->SetRight(rightPath);

}

return 1;

}

void SkyBox::SetForward(const char\* picPath)

{

if (m\_IsInit)

{

int forwardIndex = 0;

this->m\_VertexBuf[forwardIndex].Init(4);

this->m\_VertexBuf[forwardIndex].SetPosition(0, -0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[forwardIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[forwardIndex].SetPosition(1, 0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[forwardIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[forwardIndex].SetPosition(2, 0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[forwardIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[forwardIndex].SetPosition(3, -0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[forwardIndex].SetTexcoord(3, 0, 1);

m\_Shader[forwardIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[forwardIndex].SetTexture2D(picPath,0);

}

}

void SkyBox::SetBack(const char\* picPath)

{

if (m\_IsInit)

{

int backIndex = 1;

this->m\_VertexBuf[backIndex].Init(4);

this->m\_VertexBuf[backIndex].SetPosition(0, 0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[backIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[backIndex].SetPosition(1, -0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[backIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[backIndex].SetPosition(2, -0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[backIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[backIndex].SetPosition(3, 0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[backIndex].SetTexcoord(3, 0, 1);

m\_Shader[backIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[backIndex].SetTexture2D(picPath,0);

}

}

void SkyBox::SetLeft(const char\* picPath)

{

if (m\_IsInit)

{

int leftIndex = 4;

this->m\_VertexBuf[leftIndex].Init(4);

this->m\_VertexBuf[leftIndex].SetPosition(0, -0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[leftIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[leftIndex].SetPosition(1, -0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[leftIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[leftIndex].SetPosition(2, -0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[leftIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[leftIndex].SetPosition(3, -0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[leftIndex].SetTexcoord(3, 0, 1);

m\_Shader[leftIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[leftIndex].SetTexture2D(picPath,0);

}

}

void SkyBox::SetRight(const char\* picPath)

{

if (m\_IsInit)

{

int rightIndex = 5;

this->m\_VertexBuf[rightIndex].Init(4);

this->m\_VertexBuf[rightIndex].SetPosition(0, 0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[rightIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[rightIndex].SetPosition(1, 0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[rightIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[rightIndex].SetPosition(2, 0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[rightIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[rightIndex].SetPosition(3, 0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[rightIndex].SetTexcoord(3, 0, 1);

m\_Shader[rightIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[rightIndex].SetTexture2D(picPath,0);

}

}

void SkyBox::SetTop(const char\* picPath)

{

if (m\_IsInit)

{

int bottomIndex = 2;

this->m\_VertexBuf[bottomIndex].Init(4);

this->m\_VertexBuf[bottomIndex].SetPosition(0, -0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[bottomIndex].SetPosition(1, 0.5f, 0.5f, -0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[bottomIndex].SetPosition(2, 0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[bottomIndex].SetPosition(3, -0.5f, 0.5f, 0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(3, 0, 1);

m\_Shader[bottomIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[bottomIndex].SetTexture2D(picPath,0);

}

}

void SkyBox::SetBottom(const char\* picPath)

{

if (m\_IsInit)

{

int bottomIndex = 3;

this->m\_VertexBuf[bottomIndex].Init(4);

this->m\_VertexBuf[bottomIndex].SetPosition(0, -0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(0, 0, 0);

this->m\_VertexBuf[bottomIndex].SetPosition(1, 0.5f, -0.5f, 0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(1, 1, 0);

this->m\_VertexBuf[bottomIndex].SetPosition(2, 0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(2, 1, 1);

this->m\_VertexBuf[bottomIndex].SetPosition(3, -0.5f, -0.5f, -0.5f);

this->m\_VertexBuf[bottomIndex].SetTexcoord(3, 0, 1);

m\_Shader[bottomIndex].Init("res/skybox.vert", "res/skybox.frag");

m\_Shader[bottomIndex].SetTexture2D(picPath,0);

}

}

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个面那么每次绘制天空盒就会提交6此DrawCall。

②逻辑更新

void SkyBox::Update(const vec3& cameraPos)

{

if (m\_IsInit)

{

this->m\_ModelMatrix = translate(cameraPos);

}

}

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。

③绘制

void SkyBox::Draw()

{

if (m\_IsInit)

{

for (int i = 0; i < ARRLEN(m\_VertexBuf); i++)

{

RenderDomain domain(m\_VertexBuf[i], m\_Shader[i], m\_ModelMatrix);

domain.options.DrawType = DRAW\_QUADS;

domain.options.DepthTest = 0;

SceneManager::DrawGameObject(domain);

}

}

}

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。

④Shader

顶点Shader:

attribute vec4 position;

attribute vec4 texcoord;

uniform mat4 ModelMatrix;

uniform mat4 ViewMatrix;

uniform mat4 ProjectionMatrix;

varying vec4 V\_Texcoord;

void main()

{

V\_Texcoord=texcoord;

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*position;

}

片元Shader:

#ifdef GL\_ES

precision mediump float;

#endif

uniform sampler2D U\_Texture\_1;

varying vec4 V\_Texcoord;

void main()

{

gl\_FragColor=texture2D(U\_Texture\_1,V\_Texcoord.xy);

}

6面离散的天空盒Shader较为简单，只需要进行两个操作：计算位置->贴图，即可完成将纹理贴到1个面上。执行6次Shader即可将纹理贴到6个面上。

⑤销毁

void SkyBox::Destory()

{

if (m\_IsInit)

{

for (int i = 0; i < ARRLEN(m\_VertexBuf); i++)

{

m\_VertexBuf[i].Destory();

m\_Shader[i].Destory();

}

m\_IsInit = 0;

}

}

由于该6面体天空盒有6个面，那么资源的析构也需要进行6次。

### 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。

立方体天空盒的成员变量：

class SkyBoxC:public RenderAble

string m\_ModelName;

①初始化

bool SkyBoxC::Init(const char\* forwardPath, const char\* backPath, const char\* topPath, const char\* bottomPath, const char\* leftPath, const char\* rightPath,

const char\* vertexShader, const char\* fragShader)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

if (ResourceManager::GetModel("res/Cube.obj", m\_VertexBuf))

{

m\_ModelName = "res/Cube.obj";

m\_Shader.Init(vertexShader, fragShader);

m\_Shader.SetCueMap(forwardPath, backPath, topPath, bottomPath, leftPath, rightPath);

m\_Options.DepthTest = 0;

m\_Options.DrawType = DRAW\_TRIANGLES;

return 1;

}

}

return 0;

}

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。

②Shader

顶点Shader

#ifdef GL\_ES

precision mediump float;

#endif

attribute vec4 position;

uniform mat4 ModelMatrix;

uniform mat4 ViewMatrix;

uniform mat4 ProjectionMatrix;

varying vec3 V\_Texcoord;

void main()

{

V\_Texcoord=-position.xyz;

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*position;

}

片元Shader

#ifdef GL\_ES

precision mediump float;

#endif

varying vec3 V\_Texcoord;

uniform samplerCube U\_Texture\_CubeMap;

void main()

{

gl\_FragColor=texture(U\_Texture\_CubeMap,normalize(V\_Texcoord));

}

注意顶点着色器部分，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。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。

成员变量定义：

class FogObj:public GameObject

//非线性雾需求数据

float m\_Mul = 1;//雾随距离的衰减速度

float m\_Pow = 0.8f

①初始化

bool FogObj::Init(const char\* path, const char\* vertexShader, const char\* fragmentShader)

{

if (GameObject::Init(path, vertexShader, fragmentShader))

{

SetFloat("U\_FogMul", m\_Mul);

SetFloat("U\_FogPow", m\_Pow);

return 1;

}

return 0;

}

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、幂次数赋值即可。

②Shader

顶点Shader：

//省略光照类型颜色的抽象算法

vec4 GetLight1Color()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

//direction

if(U\_Light1\_Opt.x==1.0)

{

color=GetDirectionLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Dir,U\_Light1\_Opt);

}

//point

else if(U\_Light1\_Opt.x==2.0)

{

color=GetPointLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Pos,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric);

}

//spot

else if(U\_Light1\_Opt.x==3.0)

{

color=GetSpotLightColor(U\_Light1\_Ambient,U\_Light1\_Diffuse,U\_Light1\_Specular,U\_Light1\_Pos,U\_Light1\_Dir,U\_Light1\_Opt,U\_Light1\_Constant,U\_Light1\_Linear,U\_Light1\_Quadric,U\_Light1\_Exponent,U\_Light1\_CutOff);

}

return color;

}

void main()

{

V\_EyeSpacePos=ViewMatrix\*ModelMatrix\*position;

V\_Texcoord=texcoord;

V\_Color=GetLight1Color();

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*position;

}

顶点着色器中包含光照颜色计算，因为雾化的游戏物体其受光会被雾削弱，对光照精细度的要求没有那么高，故将计算放在顶点着色器。

片元着色器

#ifdef GL\_ES

precision mediump float;

#endif

uniform sampler2D U\_Texture\_1;

uniform float U\_FogFar;

uniform float U\_FogNear;

uniform float U\_FogMul;

uniform float U\_FogPow;

varying vec4 V\_Color;

varying vec4 V\_EyeSpacePos;//顶点在摄像机坐标系的位置

varying vec4 V\_Texcoord;

//非线性雾Ex

float GetCalculateExpExFog(float distance)

{

float fogAlpha=exp(-pow(distance\*U\_FogMul,U\_FogPow));

fogAlpha=clamp(fogAlpha,0.2,1.0);

return fogAlpha;

}

//贴图1

vec4 GetTexture1()

{

vec4 color=vec4(1.0,1.0,1.0,1.0);

vec4 textureColor=texture2D(U\_Texture\_1,V\_Texcoord.xy);

if(textureColor.xyz!=vec3(0.0,0.0,0.0))

{

color=textureColor;

}

return color;

}

void main()

{

float fogAlpha=GetCalculateExpExFog(abs(V\_EyeSpacePos.z/V\_EyeSpacePos.w));

vec4 color=V\_Color\*GetTexture1();

gl\_FragColor=mix(vec4(0.7,0.7,0.7,1.0),color,fogAlpha);

}

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。函数float GetCalculateExpExFog(float distance)计算出雾的浓度（0.2~1.0），计算思路是这样的：将距离与衰减因子做乘积，然后与幂次数做幂运算，将所得结果平方。最后再讲所得值限制在0.2~1.0这个区间，就可以实现物体由远及近的雾化效果。

### 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。下面给出实现实例：

①旋涡状粒子

属性定义：

class SurroundParticle:public ParticleSystem

float angle = 0.0f;

float m\_PointSize = 20;

int m\_MaxPar;//最大粒子数

虚函数重写：

void SurroundParticle::Init(const vec3& position, const int& maxNum, const char\* picPath)

{

if (!m\_IsInit)

{

m\_IsInit = 1;

SetPosition(position);

m\_MaxPar = maxNum;

m\_VertexBuf.Init(m\_MaxPar);//初始化若干个顶点

for (int i = 0; i < m\_MaxPar; i++)

{

m\_VertexBuf.SetColor(i, 0.1f, 0.4f, 0.6f);

m\_VertexBuf.SetPosition(i, 2 \* cosf(float(i)\*8.0f\*3.14f / 180.0f), 0, 2 \* sinf(float(i)\*8.0f\*3.14f / 180.0f));

}

m\_Shader.Init("res/SurroundParticle.vert", "res/SurroundParticle.frag");

m\_Shader.SetTexture2D(picPath,1);

SetPointSize(m\_PointSize);

}

}

void SurroundParticle::Update()

{

INIT\_TEST\_VOID

float deltime = Time::DeltaTime();

angle += deltime\*10.0f;

SetRotate(0, angle, 0);

for (int i = 0; i < VBO\_NAME.GetLenth(); i++)

{

Vertex& vertex = VBO\_NAME.mutable\_vertex()[i];

vertex.SetNormal(vertex.GetNormal().x, 0.1f\*i, vertex.GetNormal().z, vertex.GetNormal().w);//这里法线是用来代替单个粒子的位置偏移使用

if (i > 90)

{

break;

}

}

}

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。

②Shader

顶点Shader

attribute vec4 position;

attribute vec4 color;

attribute vec4 normal;

uniform mat4 ModelMatrix;

uniform mat4 ViewMatrix;

uniform mat4 ProjectionMatrix;

uniform float U\_PointSize;

varying vec4 V\_Color;

void main()

{

V\_Color=color;

gl\_PointSize=U\_PointSize;

vec4 pos=vec4(position.x+normal.x,position.y+normal.y,position.z+normal.z,1.0);

gl\_Position=ProjectionMatrix\*ViewMatrix\*ModelMatrix\*pos;

}

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。

片元着色器和其它模块类似，故省略。

# 总结与改进

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。其中平台无关层需要实现的功能边界较小，在此处不做讨论。

## 5.1 资源管理模块

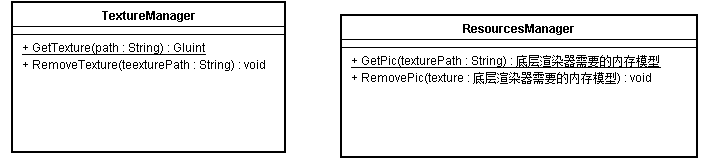
资源管理模块通过引用计数的方式，使用STL的map、vector容器管理资源。对外提供Get/Remove方式获取和卸载资源，同时会增加资源管理模块内的引用计数值。下面对部分有待改善的资源的管理方式加以讨论。

### 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。

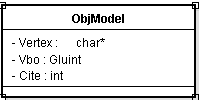
对于Shader管理、模型管理同样，这样的设计使得各类资源的管理分开，而对外只提供ResourcesManager中必要的接口，实现了一个高内聚、低耦合且易于维护的资源管理器总体架构。

### 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。

### 5.2.2 RenderAble\_Light

RenderAble\_Light类是所有受光对象的基类，

## 5.3 场景管理模块

# 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。其中平台无关层需要实现的功能边界较小，在此处不做讨论。

# 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.