

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？这就要说到光栅化的问题。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）

其中θ表示着色点到光源的射线和光源方向法线的夹角，如果夹角大于外锥则根本照射不到。ɑ表示外锥角度，Φ表示内锥角度。p表示指数因子。

（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

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被运行，因为在这里进行裁剪可以避免将所有几何体进行可能是不必要的矩阵变换，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

## 2.4 构建一个3D场景

# 引擎设计与架构

## 3.1需要使用的外部库

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。

同时应该将全局常用的宏定义写在ggl.h文件中。

#define INVALID (-1)

#define \_INVALID\_ID\_ (0)

#define \_INVALID\_LOCATION\_ (-1)

#define ASSERT\_PTR\_BOOL(p) if(p==nullptr){return 0;}

#define ASSERT\_INT\_BOOL(i) if(i==\_INVALID\_ID\_){return 0;}

#define ASSERT\_INT\_VOID(i) if(i==\_INVALID\_ID\_){return ;}

#define ARRLEN(arr) sizeof(arr)/sizeof(arr[0])

#define BEGIN VBO\_NAME.Begin();{SHADER\_NAME.Begin();{SHADER\_NAME.Bind(glm::value\_ptr(MODELMATRIX\_NAME), glm::value\_ptr(VIEWMATRIX\_NAME), glm::value\_ptr(PROJECTIONMATRIX\_NAME));

#define END }SHADER\_NAME.End();}VBO\_NAME.End();

#define DRAW { glDrawArrays(GL\_TRIANGLES,0,VBO\_NAME.GetLenth());}

#define DRAW\_MODEL BEGIN DRAW END

#define VBO\_NAME m\_VertexBuf

#define SHADER\_NAME m\_Shader

#define MODELMATRIX\_NAME m\_ModelMatrix

#define VIEWMATRIX\_NAME viewMatrix

#define PROJECTIONMATRIX\_NAME ProjectionMatrix

包括一些类型安全的断言、常用绘制指令、常用判断、常量等。

## 3.2平台无关层

平台无关层的功能相当于main函数，注册并弹出一个OpenGL渲染窗口。它需要利用OpenGL跨平台特性来实现，那么我们定义一个class EngineBehavior在文件GameDefine/Global/Winapp.h，在类中进行窗口的初始化，接口设计如下：

class EngineBehavior

{

public:

void Awake(const HINSTANCE& hInstance);

virtual void Start();

virtual void Update();

virtual void OnDestory();

virtual void Event(UINT message, WPARAM wParam, LPARAM lParam);

private:

void SystemStart();

void SystemUpdate();

void SystemDestory();

int SystemEvent(UINT message, WPARAM wParam, LPARAM lParam);

};

其中void Awake(const HINSTANCE& hInstance)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void 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void 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int SystemEvent(UINT message, WPARAM wParam, LPARAM lParam)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void 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 virtual void Start();virtual void Update();virtual void OnDestory();virtual void Event(UINT message, WPARAM wParam, LPARAM lParam);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。

## 3.3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。

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；底层渲染器是引擎的核心模块，它装载了渲染引擎所有核心逻辑和算法，负责进行单场景的渲染管理；多场景管理模块管理所有场景，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，它向平台无关层隐藏了场景细节，向底层渲染器隐藏了操作系统细节。

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。

多场景管理

多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，能够加载、更新、退出一个场景。 底层渲染只管渲染一个场景需要做的事情，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

底层渲染器

只管渲染一个场景需要做的事情，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。多场景管理模块的更新应该是非常少的，仅限于加强对当前场景的管理支持而提供新的接口，对其它模块没有任何影响。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。

## 3.4资源管理

资源管理模块接口让底层渲染器和开发者调用，而不关心底层渲染器用来做什么和怎么用，并提供接口让取资源的函数能够在结束使用时手动析构资源。需要管理的主要就是CPU的内存以及GPU的显存，被管理的资源包括图片资源、模型资源、GPU程序的读取、转码加载，以及后续对已加载资源的管理等。下面讲述具体设计。

### 3.4.1 GPU程序的内存模型

struct Program

{

GLuint program = \_INVALID\_ID\_;

int CiteCount = 0;

public:

bool operator<(const Program& other)

};

### 3.4.2 2D纹理的内存模型

struct Texture

{

GLuint texture=\_INVALID\_ID\_;

int CiteCount = 0;

public:

bool operator<(const Texture& other)

};

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。

### 3.4.3 obj文件的内存模型

struct FloatData

{

float v[3];

};

struct VertexDefine

{

int32\_t positionIndex;//位置信息的Index

int32\_t textcoordIndex;//纹理贴图信息的Index

int32\_t normalIndex;//法线信息的Index

};

struct ObjModel

{

vector<FloatData> positions, texcoords, normals;//记录真实数据

vector<VertexDefine> VertexList;//记录绘制指令

vector<int32\_t> Indexes;//表示所有绘制指令的索引位置。

};

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。

### 3.4.4 资源管理类需要的容器

目前需要管理的只有3种资源，后续版本更新后将不断增加。

//texture管理

static std::map<string, Texture> m\_mTexture;

//GPU程序管理

static std::map<string, Program> m\_mProgram;

//ObjModel管理

static std::map<string, ObjModel> m\_mModel;

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。

### 3.4.5 资源管理类接口定义

纹理资源

static GLuint GetPic(const char\* bmpPath);

static void RemovePic(GLuint texture);

Shader

static GLuint GetProgram(const char\* vertexShaderPath, const char\* fragmentShaderPath);

static void RemoveProgram(GLuint program);

模型资源

static bool GetModel(const char\* path,VertexBuffer& vbo);

static void RemoveModel(const char\* path);

任意类型文件

static unsigned char\* LoadFileContent(const char\* path, int& filesize);

目前提供了对纹理，模型，程序，显存，的加载和卸载，以及最通用的加载一个文件并返回文件的内容，使其它模块可以很方便地获取到外部资源而不用担心内存泄露。当然，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。

## 3.5底层渲染（先不写）

底层渲染模块是最为复杂的，包括但不限于VertexBuffer、ElementBuffer、FrameBuffer、Shader程序、视景体这些组件以及持有这些组件的逻辑对象摄像机、模型、天空盒、地面、粒子系统等。单场景的管理也将在这里执行。Scene类实现了场景基本的逻辑，是所有场景的基类，开发者需要继承Scene类，并重写一系列虚函数，就可以完成游戏场景的渲染。

负责单场景的渲染和管理工作。实现了定义场景内所有渲染对象、更新所有对象信息、裁剪、向GPU传递需要渲染的顶点数据等功能。

## 3.6多场景管理

多场景管理模块管理所有场景，并执行当前场景的加载、运行、结束等逻辑。开发者可以很方便地通过类的全局方法加载场景、切换场景，而不用担心内存或是显存泄露。

负责管理多个场景，执行对当前场景的加载、运行、卸载功能。

# 编码实现

# 架构与实现的优劣分析

# 第六章 引擎后期实现方向