河北工业大学城市学院

毕业设计说明书

作　　者：潜森伟　　学　号：208813

系：计算机科学与软件工程

专　　业：软件工程

题　　目：基于 Vulkan 的软渲染引擎的设计与实现

指导者：

评阅者：

2024 年 5 月 27 日

毕业设计（论文）中文摘要

基于 Vulkan 的软渲染引擎的设计与实现

摘要：

计算机图形学飞速发展，新的渲染技术不断出现，这也使得新的渲染引擎不断出现，以满足各行各业对不同渲染技术的要求。本文主要是对引擎架构设计的详细描述以及关键节点的解答。使用到的开发语言为 C++,标准是 C++20，开发工具为 Visual Studio 2022,渲染 API 为 Vulkan。最终能使引擎支持对自定义着色器的正确解析与对多种类型模型的正确渲染。而最终的效果取决于用户对着色器的理解，该引擎提供了最为繁杂的渲染管线的可视化搭建。本设计的课题意义旨在对自身大学 4 年水平的检验与对软件设计整体流程的完整尝试，会有不足之处。主要的研究内容并非深不可测的抽象理论，也没有实际的研究目标与对象，只是为了应用自身所学而即兴开发的渲染引擎，最终目标也仅仅是能够自由地完成渲染内容。

关键词：　计算机图形学　Vulkan　C++

毕业设计（论文）外文摘要

Title Design and Implementation of a Vulkan-Based-Software Rendering Engine

Abstract

Computer graphics is rapidly evolving, with new rendering techniques continually emerging, leading to the continuous development of new rendering engines to meet the varied requirements across industries. This paper primarily provides a detailed description of the engine architecture design and answers key nodes. The development language used is C++, adhering to the C++20 standard, with Visual Studio 2022 as the development tool and Vulkan as the rendering API. The ultimate goal is to enable the engine to correctly parse custom shaders and render multiple types of models accurately. The final rendering effect depends on the user's understanding of shaders, so the engine provides the most intricate visual construction of rendering pipelines. The significance of this design project lies in evaluating one's university-level skills after four years of study and attempting a complete software design process, albeit with potential shortcomings. The main research content does not delve into deep, unfathomable abstract theories, nor does it have specific research goals and objects. Instead, it is simply an impromptu development of a rendering engine to apply one's own learning, with the ultimate goal being the freedom to accomplish rendering tasks.

Keywords： Computer Graphics Vulkan C++

目　次

1 引言

计算机图形学飞速发展，新的渲染技术不断出现，同时，动漫，影视，游戏，广告，科学可视化等领域对渲染软件的要求也越来越高。原有的渲染软件必须不断升级更新才能满足人们的需求。这也使得新的渲染引擎不断出现，以满足各行各业对不同渲染技术的要求。

1.1 编写目的

本文档在概要设计的基础上，进一步的展示了软件结构、数据结构、与算法设计，详细的介绍了系统各个模块的工作联系。预期的读者：对自研渲染有兴趣的C++程序员。

1.2 本设计的主要工作

本次毕业设计的是基于Vulkan的软渲染引擎。主要工作任务实现以下系统功能：多格式模型的读取、渲染管线的搭建、Shader的解析、多种黑盒操作的可视化、正确的渲染。本文重点介绍了渲染引擎开发的详细过程，主要包括需求分析、架构设计、数据结构设计、系统实现、程序文件组织、系统测试与调试。

1.3 开发与测试环境

测试软件名称：ZsEngine(基于本文档描述的渲染引擎)

测试硬件环境：

CPU：Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz

GPU：AMD Radeon HD 8600/8700M

Vulkan版本号：1.2.131

2 可行性研究与需求分析

2.1 可行性研究

课题并非是以商业化为最终目的，在设计之初也并未掺杂任何的创造经济的想法，其最大的作用是用于整合各个时期所学知识，其在社会市场经济与风险上无任何需要研究的方面。C++在理论上可以实现任何高性能程序，配合 Vulkan 能够完成目前所需的所有功能。

2.2 需求分析

本系统开发的首要任务是了解系统最终实现的功能，这对于定义应用程序的功能十分重要，根据渲染引擎特点以及渲染器应具有的基本功能，渲染引擎开发完成后应实现以下目标：

正确地运行在任意测试系统环境下。

支持用户自定义开发并关联引擎相关接口。

读取多种主流格式模型数据。

正确应用用户提供的着色器

提供正确的渲染结果，并于主流商用渲染器效果相差无几。

能够通过简单的设置与编程，快速达到所需的大致效果，为后续视觉开发提供参考。

2.3 关键技术

课题的开发所需要的只有两个最为关键的技术，一个是 C++，另一个便是 Vulkan。

Vulkan 是一种图形和计算 API，由 Khronos Group 开发和维护。它被设计用于高性能图形应用程序，特别是游戏开发。Vulkan 的目标是提供比传统图形 API 更低的驱动开销，并充分利用多核处理器和现代 GPU 的并行计算能力。

与其他图形 API 相比，Vulkan 具有许多优势。首先，它具有低开销的命令缓冲区和多线程渲染，这使得开发者能够更好地控制 GPU 资源的使用和管理。其次，Vulkan 提供了显式的并行计算支持，可以在 GPU 上进行更复杂和高效的计算任务。此外，Vulkan 还引入了异步操作和更好的内存管理机制，以减少 CPU 和 GPU 之间的通信延迟。

Vulkan 的跨平台性也是其重要特点之一。它可以在 Windows、Linux、Android 和其他一些操作系统上运行，并且支持各种硬件配置。这使得开发者可以在不同的平台上构建高性能的图形应用程序，而无需针对每个平台单独编写代码。

Vulkan 还提供了一套庞大的功能和扩展，以满足不同应用的需求。开发者可以根据自己的需求选择需要的功能，并通过扩展来进一步定制 API。这使得 Vulkan 成为一个非常灵活和可扩展的图形和计算解决方案。

总的来说，Vulkan 是一个强大的图形和计算 API，它通过降低驱动开销、利用多核处理器和并行计算能力、提供跨平台支持以及灵活的功能和扩展机制，为开发者提供了构建高性能图形应用程序的强大工具。无论是游戏开发还是其他图形应用开发，Vulkan 都是一个非常值得考虑的选择。

2.4 定义

Shader[2]：着色器（Shader）是用来实现图像渲染的，用来替代固定渲染管线的可编辑程序。

渲染引擎：上原始模型的基础上进行渲染，添加颜色、光照、阴影等内容，最后渲染到屏幕上，呈现最终效果，这就是图形图像渲染引擎的工作。

渲染管线：计算机图形学中从应用程序阶段到最终渲染输出二维图像的一系列处理过程。

描述符集：关联用户的数据资源和着色器

多种黑盒操作：诸如描述符集的绑定相关。

dxc/glslang 编译器： 用于将文本形式的 shader 源文件编译为 spv 二进制文件。

3 系统设计

这个阶段的工作是划分出引擎架构，进行最简单基础的架构分析，但是每个元素仍然处于黑盒子级，具体内容将在详细设计中。概要设计的任务主要是确定系统中模块之间的相互联系。

3.1 模块划分

本系统一共包括 4 个大模块，若干个小模块。小模块将在详细设计中进行描述。大模块分别为 ThirdPartyManager、Core、Resrouce、Editor。如图 3-1 所示。

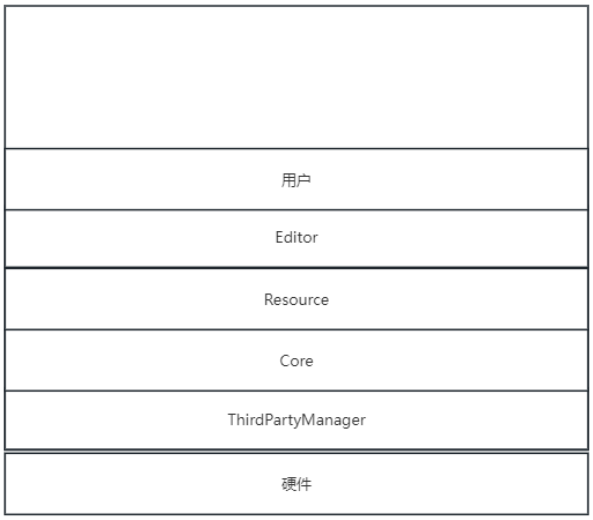

图 3-1 引擎基本架构

3.1.1 第三方库管理(ThirdPartyManager)

ThirdPartyManager 模块主要用于管理引擎用到的第三方支持库，这些库全部支持跨平台[1]与多线程[3]，其中对 4 个主要库进行了一定程度的封装。提供了第三方库的调试信息。

3.1.2 核心层(Core)

Core 模块是除 ThirdPartyManager 外其他所有模块的基础。其中定义了引擎最基础的数据结构。提供了基础的引擎调试信息[4]。

3.1.3 资源层（Resrouce）

Resource 模块主要用于管理引擎资源。对 Vulkan 的若干对象进行了资源转化。其中 Resource 模块中划分了一个 Scene 的子模块，对场景资源进行额外的管理控制，并加速开发期间的编译调试[7]。

3.1.4 上层/编辑器（Editor）

Editor 模块主要用于显示 UI，方便用户与程序进行视觉上的交互。该模块还处理了渲染命令的最终提交。

3.2 系统总体功能设计

通过需求分析，引擎系统主要功能如下：

3.2.1 文件读取

通过“文件视图”拖动文件进行模型的读取；通过“文件视图”拖动文件到着色器中进行纹理的读取。

3.2.2 Shader 编译

不通过 dxc、glslang 等外部编译器[8-9]进行编译。程序内置了 Shaderc 的编译功能，能够将 glsl 源文件编译成 spv。此外，引擎还对此进行反射，获取 shader 内部的成员变量。所用到的是 Google 发布的 Shaderc 源码编译而来[10-12]。

3.2.3 窗口布局

通过 ImGui 实现窗口 Dock。可以通过鼠标对程序内部窗口进行拖拽，从而定制令人舒适的页面布局。并且具有记忆功能，在下次启动引擎时，能够恢复布局。

3.2.4 实时渲染视图

结合上述所有功能，完成最终的渲染效果。任何对 Shader 的操作，都可以实时渲染，其性能在中低端机上也能稳定 60FPS。

3.2.5 材质编辑器

可视化操作整条渲染管线。能够通过连线的方式，对 Shader 进行控制，从而更加方便地观察参数对渲染结果的影响。

3.3 数据结构设计

此小结包含了各个模块的数据结构与类的设计概要，这里给出了最重要最基础的数据结构。

3.3.1 ThirdPartyManager

3.3.1.1 Manager

基类 Manager 对整个第三方库进行管理，继承自 Manager 的各个实体类需要实现三个方法[13-16]。

```
1.  virtual T* init() = 0;
2.  virtual void destroy() = 0;
3.  virtual void windowResize() = 0;
```

init：用于初始化管理类。

destroy：对资源的销毁。

windowResize：只对 SDL,Vulkan,ImGui 这三个与平台窗口相关的管理类有效。

对于整个第三方库，共使用了 5 个子类进行细化管理。每个子类对应一个库。子类的具体结构在详细设计中进行说明。下同。

基类：

Manager<T>：整个第三方管理层的抽象模板类，T 用于方便获取子类。其成员变量 m_isInitialized 用于标识该第三方库是否初始化完成，该基类需要实现 init()与 destroy()两个函数。

子类：

SDLManager:public Manager<SDLManager>：用于管理 SDL 库。SDL 主要用于与计算机的交互，诸如窗口创建，键鼠响应等。

VkManager：public Manager<VkManager>：用于管理 Vulkan 库。Vulkan 主要用于渲染操作。

ImGuiManager：public Manager<ImGuiManager>：用于管理 ImGui 库。ImGui 主要用于 UI 的绘制。

AssimpManager：public Manager<AssimpManager>：用于管理 Assimp 库。Assimp 主要用于模型数据的读取。

JsoncppManager：public Manager<JsoncppManager>：用于管理 Jsoncpp 库。Jsoncpp 主要用于 Json 的序列化与反序列化。

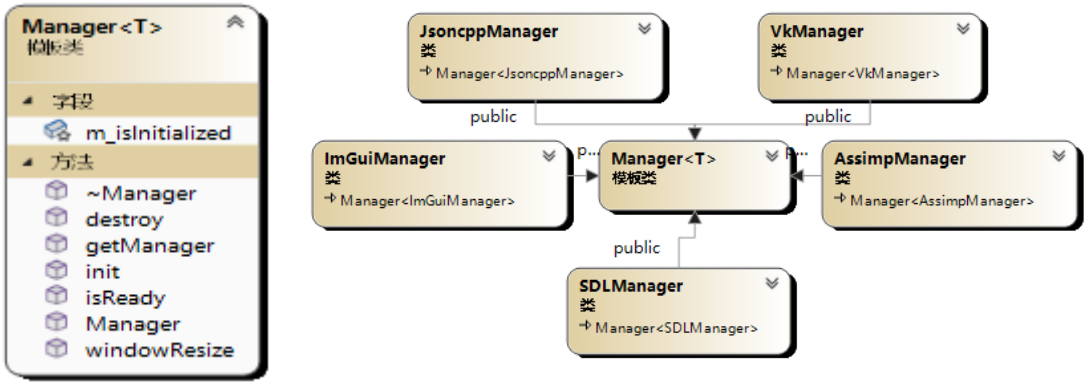

图 3-2 Manager UML 类图

3.3.1.2 vkObject

对 Vulkan 重要对象的封装,vkObject 管理了 Vulkan 对象的整个生命周期，创建与销毁都由其统一控制。该基类包含成员变量 m_label，用于标识实例化对象。凡是包含 m_label 的子类都将纳入 vkObject 的管理。

基类：

vkObject: 对 Vulkan 对象的管理，m_label 不为空的对象都将被纳入 VkManager 统一管理生命周期。

子类：

Image: VkImage 的容器，包含 VkMemory，内存大小与内存偏移。

ImageView：VkImageView 的容器，VkImage 视图。

Sampler: VkSampler 容器，包含了 VkSampler 的创建。

FrameBuffer: VkFrameBuffer 的容器，其中包含多个 VkFrameBuffer，用于实现多重缓冲。

Fence/Semaphore: Vulkan 同步对象的容器，包含创建与信号初始化。

CommandBuffer: VkCommandBuffer 容器，其中包含一个 Vk 对象，包含了 vk 命令的记录起始。

Queue: 整合了与队列有关的内容,包含了 VkQueue,VkFence,VkSemaphore，VkCommandBuffer。用于最终队列提交。

Subpass: VkSubpassDescription 的容器，实现了该结构体成员的 Set/Get 方法。其主要作用是承载 VkAttachmentReference 结构体。

RenderPass: RenderPass 相关的载体，不包含创建功能。其包含了 VkRenderPass,VkFramebuffer,VkCommandbuffer 三个对象，还包含了多个 vkBeginRenderPass 等函数所需结构体。提供了简易的 begin/end 渲染流程的函数。

Buffer: Vulkan 数据缓冲区。包含了 VkBuffer，VkMemory, Size，Offset，Uasge,Memory Property。拥有创建与销毁的功能。

VertexBuffer: Buffer 的特例，专门用于存储顶点数据的 Buffer。其中还包含了临时缓冲对象，用于转化缓冲区的 Memory property。

UniformBuffer: Buffer 的特例，专门用于存储 Uniform 的数据。

IndexBuffer：Buffer 的特例，专门用于存储索引缓冲。

PipelineLayout：渲染管线布局，包含创建功能。

Pipeline： 渲染管线，其包含了渲染管线的所有需要的结构体，并提供了 Set/Get 方法。

Texture ： 引擎设计之初遗留的对象，是对 VkImage,VkImageView,VkMemory 的整合，不包含创建销毁功能，只提供了一个容器。

TextureEXT： Texture 的子类，加入了创建与销毁的功能，还支持 VkImage 的 Blit，Copy 与 Translate Layout。而且支持 Array 和 Cube 两个类型。

DescriptorSet：VkDescriptorSet 的整合，引擎设计之初遗留，之后被 Slot 与 SlotData 替代。

Slot ： 整合了整条渲染管线 Descriptor 相关的内容，包含了 Set 的初始化、设置、更新、创建、销毁等功能。

SlotData： VkDescriptorBinding 的具象化，被包含在 Slot 中，包含了数据的映射绑定与更新。

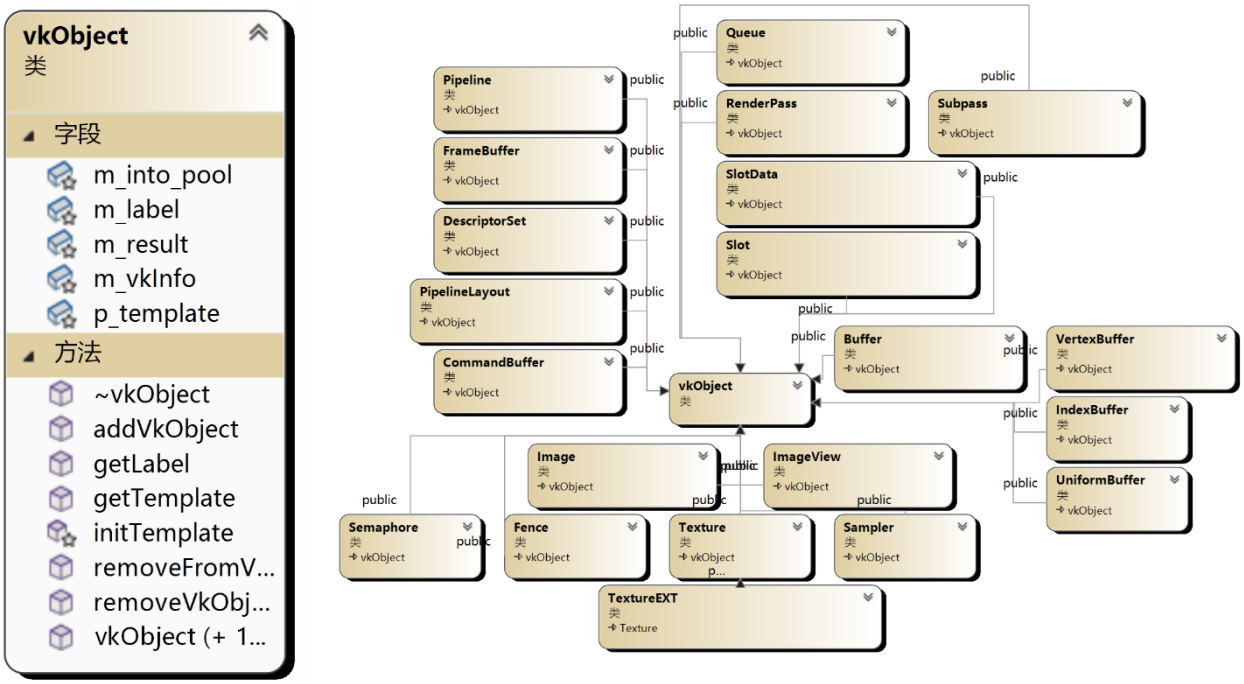

图 3-3 vkObject UML 类图

3.3.2 Core

该模块包含的内容较少，主要是用于分层，方便后续其它层在其之上进行搭建，并于 ThirdPartyManager 进行关联。该模块中最重要的就是 ZClass,所有继承自该类的子类都将纳入引擎的管理。其子类必须实现 m_uuid 的生成。其中 p_id_generate 即为 UUID 生成器，引擎给出了一个基础的，每秒最多可生成 1048576 个不同 UUID 的生成器，如果不满足需要，可以通过继承 IDGenerate，重写 generateUUID()方法来实现更多数量的 UUID 生成器。

基类

ZClass: 包含 3 个必须的成员,m_initialized 用于标识是否出初始化完成，m_uuid 用于标识 UUID，p_id_generate 是 UUID 生成器。

子类

其子类全部是实现在 Resource 模块中。

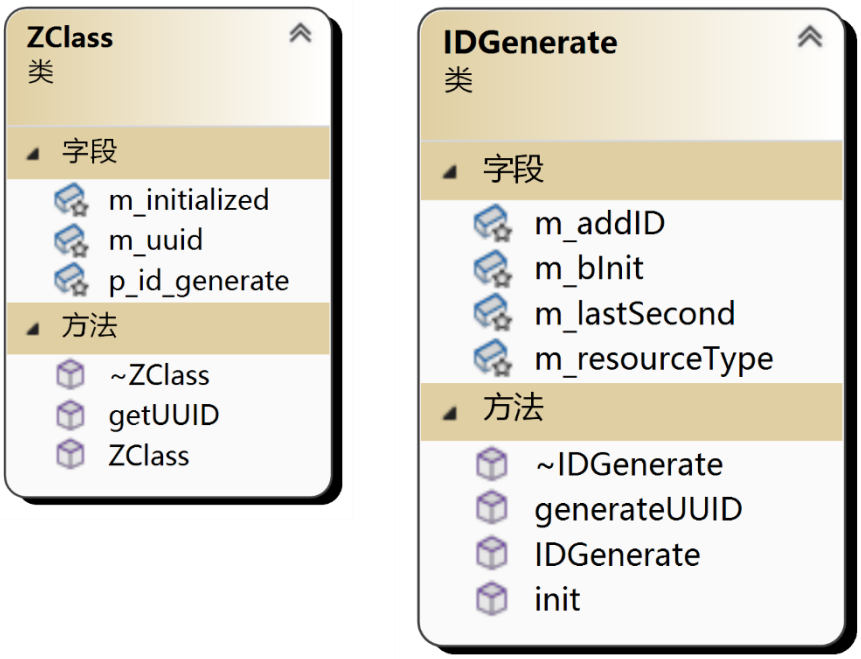

图 3-4 ZClass、IDGenerate UML 类图

3.3.3 Resource

3.3.3.1 ZResource

继承自 ZClass，是整个引擎资源的基类。所有引擎资源类都要继承并实现如下两个方法：

```
virtual Json::Value serializeToJSON() = 0;
virtual void deserializeToObj(Json::Value& root) = 0;
```

serializeToJson()：将类成员及其数据转化为 Json 序列。

deserializeToObj(Json::Value& root)：将 Json 序列转化为实体类。Root 为 Json 序列根节点。

基类：

ZResource：ZClass 的子类，加入了 m_json_title 用于标记 Json 数据节点名称。m_res_name 用于标识资源名称。p_property 属性结构体。p_template 是新加入功能 Material Editor 所需要，实现该系列方法，即可将类显示在 Material Editor 中。

子类：

RCamera：摄像机。包含了键鼠响应，渲染纹理。

RTexture：对图像文件的整合，包含了文件的路径、大小等文件信息，不包含文件数据。

RShader：对 GLSL 源码的第一次处理加工。

Shader：对 RShader 的第一次加工，将 RShader 加工完的 GLSL 数据再次加工方便使用。引擎最核心的自动识别 GLSL 数据并创建管线的功能由其实现。

RMaterial：材质。Shader 只是 GLSL 的数据，RMaterial 则是将 GLSL 与实际的数据绑定在一起，包含了管线的绑定，DescriptorSet 的更新等内容，Slot 在这 Shader 阶段进行创建，SlotData 则是该阶段进行创建。也是引擎最核心的功能。

RRenderPass：包含了 RenderPass 的创建，将 ThirdPartyManager 中的 RenderPass 管理移至 Resource 中。

RModel：模型类，顾名思义。包含了 RMaterial 的自动创建。包含顶点数据与材质数据。

RShape：基于 RModel，标准形状基类。

RPlane：基于 RShape，平面模型。

RMesh：RModel 的子成员，包含了顶点索引信息与材质索引。

RLight：光源的实现类，其中包含了 RShape 用于在场景中显示。

RVertex：顶点数据的整合，其中包含了一个顶点所有的顶点数据。能够根据 Shader 的 VertexInputAttribute 自动生成符合内存偏移的顶点数据结构。

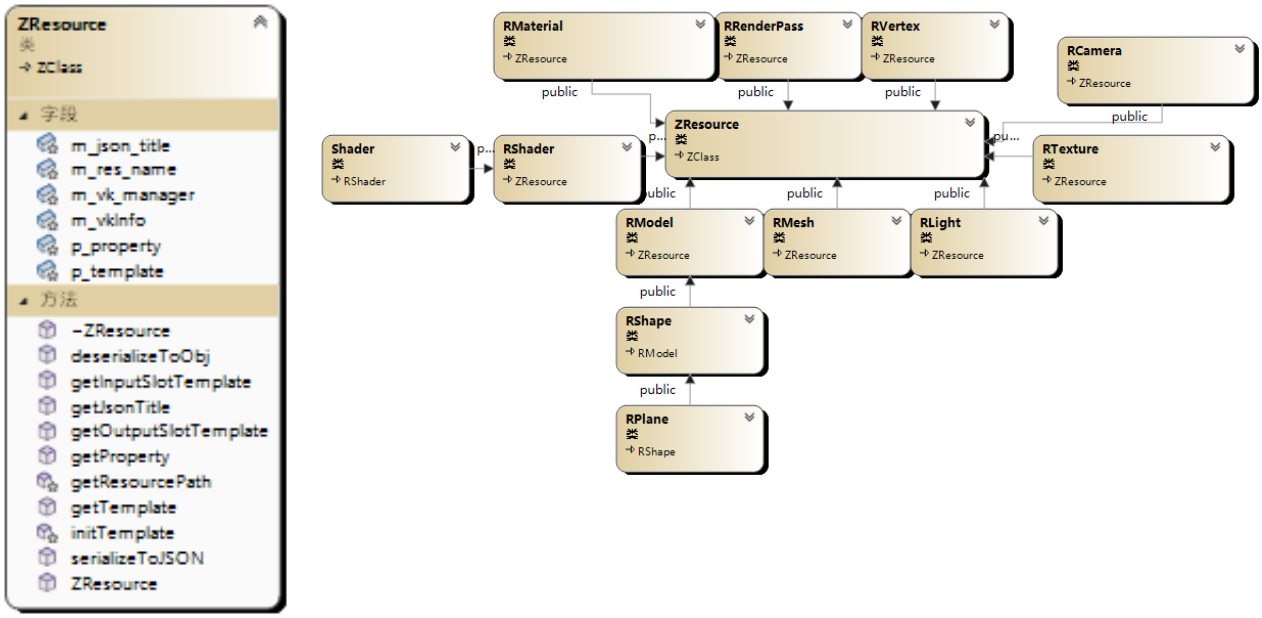

图 3-5 ZResource UML 类图

3.3.3.2 Scene

Resource 模块下的子模块。场景的具象化。

3.3.3.3 SceneObject

继承自 ZResource，使其包含在引擎资产中。继承自该基类，代表该子类为场景中所需物件，而非后台的数据处理等过程。

基类

SceneObject：所有可以在场景中使用或者渲染的对象的基类。子类继承该类，便会被加入到可渲染队列。

子类：

Scene：场景类，包含了场景中所有需要使用到的物体，渲染则是以场景为单位。

Light：场景灯光，继承自 RLight，拥有 RLight 的所有功能。

Camera：场景摄像机，继承自 RCamera，拥有 RCamera 的所有功能。

SubCamera: 场景副摄像机，继承自 Camera，新增了渲染纹理目标的成员，可以指定渲染位置与渲染目标纹理。

Material: 场景材质，继承自 RMaterial，拥有 RMaterial 的所有功能。

Texture: 场景材质，继承自 RTexture。其中仅仅是 TextureEXT 与文件信息的整合。

Model: 场景模型，继承自 RModel。增加了材质合批与网格合批，用于优化性能。

Shape: 场景形状，继承自 Model，抽象类。

Cube、Plane、Sphere：继承自 Shape，具体的形状模型。

Mesh: 场景网格，继承自 RMesh。

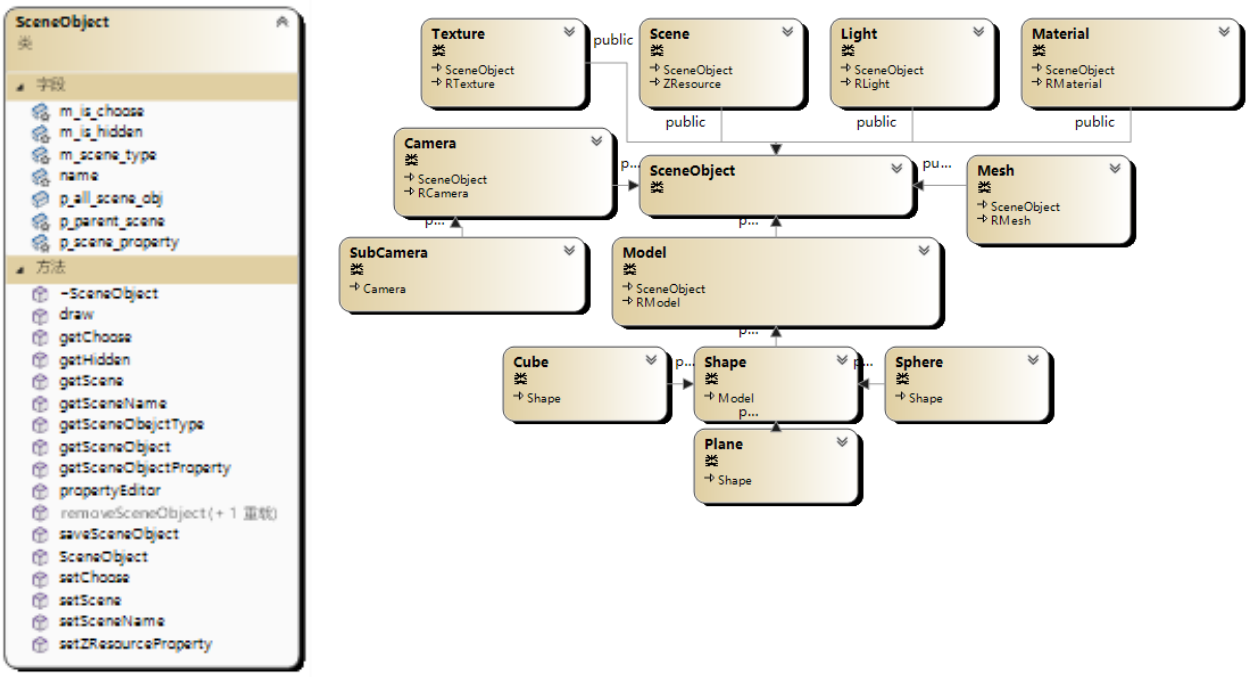

图 3-6 SceneObject UML 类图

3.3.4 Editor

用于用户视觉交互的界面系统。

基类：

BaseEditor: 可视化编辑器基类，抽象类，继承自 ZClass，包含了引擎的运行路径，设置了编辑器的运行路径。拥有 5 个 onXXX()函数需要由子类实现，其分别代表了编辑器每次渲染时的各个阶段。

子类：

ZEditor: BaseEditor 的简化，实现了 onXXX()的方法，让其后子类仅需重写需要的方法即可。

Editor: 继承自 ZEditor,主要作用是当做渲染的容器,渲染是一个死循环，该循环被放在 Editor 中。

FileEditor: 文件编辑器，可视化文件操作。

SceneEditor: 场景编辑器，用于显示最终的渲染效果。

MaterialEditor: 引擎新加入功能，可视化的材质编辑器。

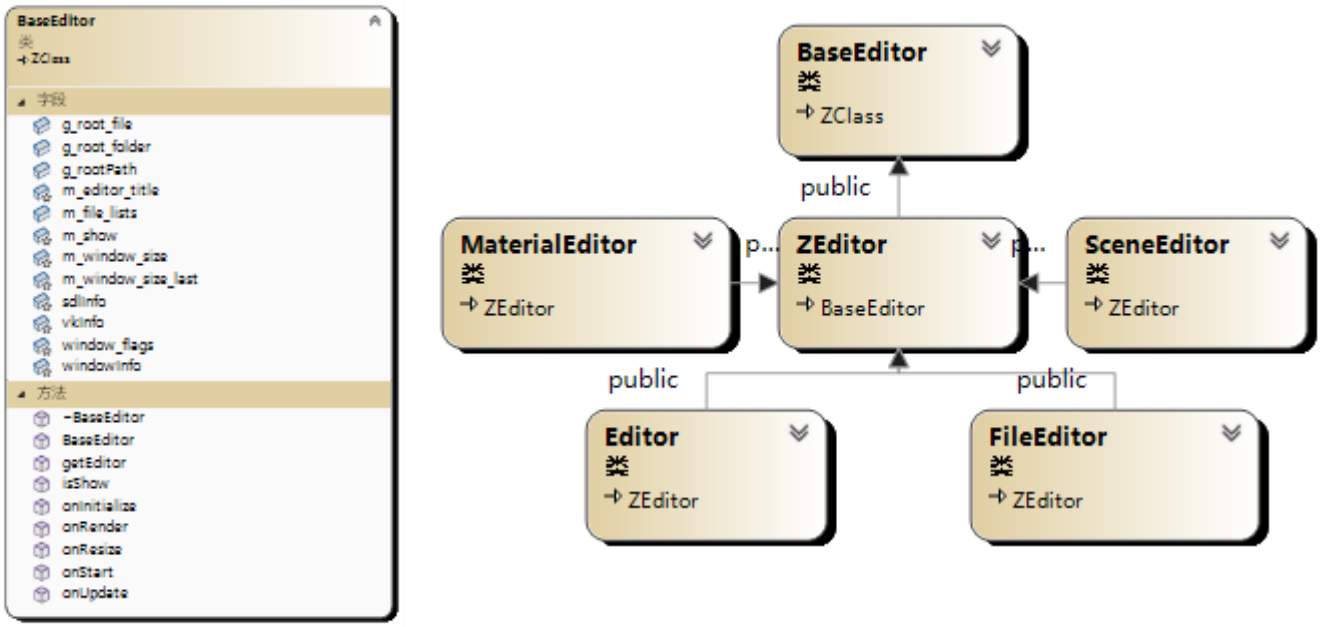

图 3-7 Editor UML 类图

4 系统实现

此章节，我们将有上到下，由浅入深，由上层到底层，详细的讲述引擎工作原理以及各个功能的详细实现，读者能够根据此说明对引擎源码进行修改。以下是 ZsEngine 的 main.cpp 内容：

```
1.  #include <ThirdPartyManager/third_party_manager_global.h>
2.  #include <Editor/editor_include.h>
3.  int main(int argc, char* argv[]) {
4.
```

```
5.      ThirdParty::init();
6.
7.      Editor::loadEditors();
8.      Editor::editor->onRender();
9.
10.     Core::destroy();
11.     ThirdParty::destroy();
12.     return 0;
13. }
```

4.1 初始化

程序最开始先对第三方库进行了初始化，其中包含了 SDL、Vulkan 和 ImGui_SDL_Vulkan 的初始化。其具体的初始化都在 Manager<T>::init 函数中各自有具体的实现。之后所有的工作都需要使用 ThirdPartyManager 模块中的内容，这里进行显示初始化。

之后调用 Editor 模块的 loadEditors 接口，用与加载交互界面。

```
1.  void loadEditors()
2.  {
3.      // RenderPass 预制
4.      Core::Resource::preload();
5.      editor = new Editor;
6.      editor_root = new ZEditor(true); {
7.          editor_root->setOnRenderFunc(OnRenderFunc::editor_root_func);
8.      }
9.      editor_scene = new SceneEditor;
10.     file_editor = new FileEditor;
11.     material_editor = new MaterialEditor;
12. }
```

4.1.1 ZEditor

图 3-7 包含了 Editor 模块的类图，其中 BaseEditor 继承 ZClass 并实现了初始化的函数，用于将 Editor 包括在引擎中，BaseEditor 中还包含了许多 ThirdPartyManager 层的管理对象，方便界面开发。BaseEditor UML 见图 3-7。

ZEditor 继承 BaseEditor，新增了一个非常关键的成员 p_onRender_func，该成员指向一个函数，ZEditor 重载了 onRender() 函数，其中执行了 p_onRender_func 指向的函数，这样就可分离该模块，让用户在外部对界面进行编写，只需要将编写的界面函数作为参数传递到 setOnRenderFunc 中，即可完成对界面的自定义。

4.1.2 Editor

Ediotr 其作用则是整个引擎的容器，重写了父类 onRender 方法:

```
1.  void Editor::Editor::onRender()
2.  {
3.      onStart();
4.
5.      bool renderReady = true;
6.      while (!windowInfo.m_window_isClosed)
7.      {
8.          // 渲染前准备
9.          onRenderStart();
10.
11.         // 事件处理，返回 False 则不继续向下执行
12.         if (!onEventProcess(renderReady))
13.         {
14.             vkInfo.newFrame();
15.             continue;
16.         }
17.
18.         if (renderReady) {
19.             // 获取交换链图像
20.             ThirdParty::vk_manager.getCurrentSwapchainImageIndex();
21.
22.             // 渲染 IMGUI
23.             onImGuiRender();
24.
25.             // 渲染呈现
26.             onFrameRenderAndPresent();
27.         }
28.     }
29.     save();
30. }
```

引擎需要一个不断循环的容器，而 Editor 就是这个容器。其中最为关键的部分就是 onImGuiRender():

```
1.  void Editor::Editor::onImGuiRender()
2.  {
3.      // Graphics render[ImGui]
4.      ThirdParty::imgui_manager.onRenderStart();
5.      for (const auto& editor : Core::Core_Pool::p_all_editor) {
6.          ZEditor* p_editor = static_cast<ZEditor*>(editor.second);
7.          if (p_editor->isShow() && p_editor != this)
8.          {
```

```
            p_editor->onResize();
            p_editor->onInitialize();
            p_editor->onStart();
            p_editor->onUpdate();
            p_editor->onRender();
        }
    }
    ThirdParty::imgui_manager.onRender();
    ThirdParty::imgui_manager.onRenderEnd();
}
```

这里将所有已知 ZEditor 的子类进行预渲染。并将渲染命令存储。然后执行 onFrameRenderAndPresent，提交所有的命令。其中包括了 Scene 产生的其他渲染命令，诸如模型等渲染命令。

4.1.3 SceneEditor

场景编辑器，主要是对渲染场景进行可视化，最终的渲染效果由其呈现。该类拥有一个成员 p_scene，用于与 Resource 的 Scene 子模块进行交互，其指向 Scene 根场景。

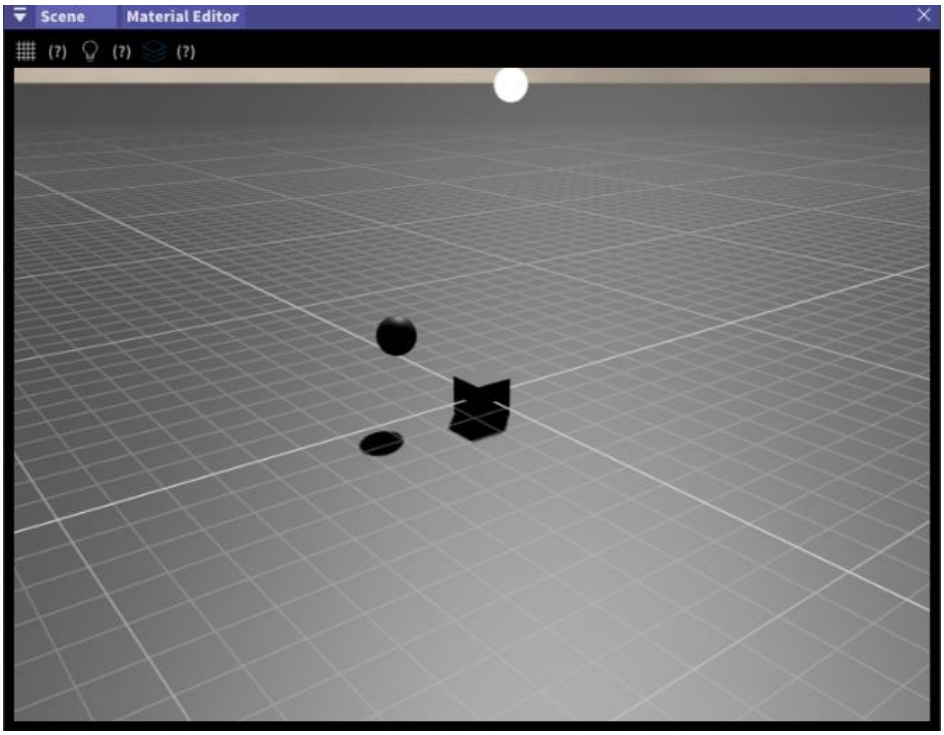

图 4-1 SceneEditor 显示效果

4.1.4 FileEditor

文件资源管理器，用于可视化文件操作。并于其他窗口进行 IO 上的联系。

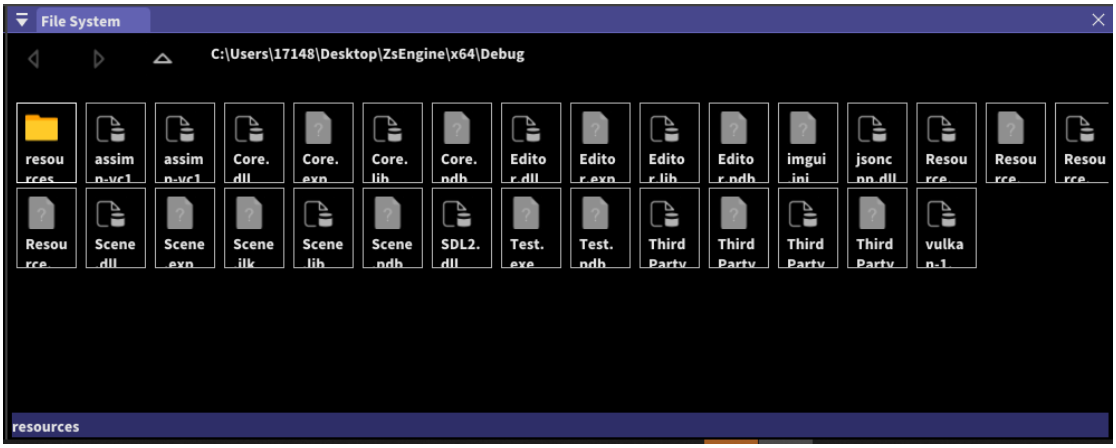

图 4-2 FileEditor 显示效果

4.1.5 MaterialEditor

材质管理器，用于可视化材质操作。

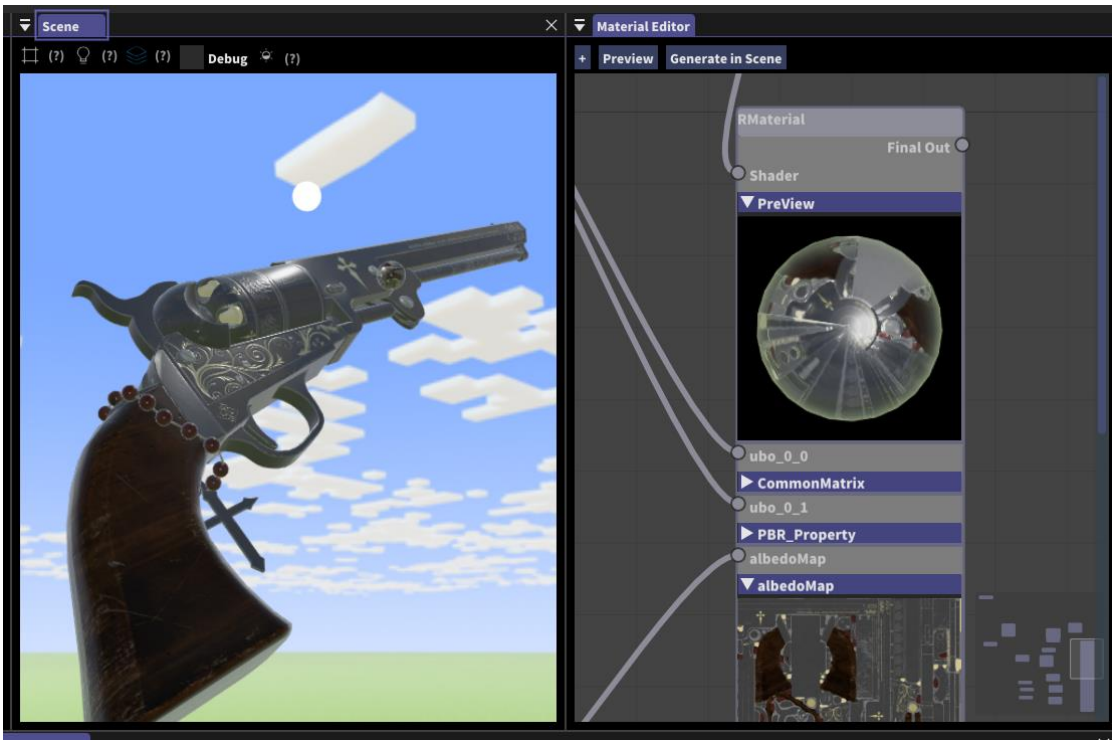

图 4-3 MaterialEditor 显示效果

4.1.6 自定义窗口

引擎包含了预设的自定义窗口，诸如 Scene Collection、Material Collection、Property Editor，这些窗口的注入为上述 4.1.1 的描述。

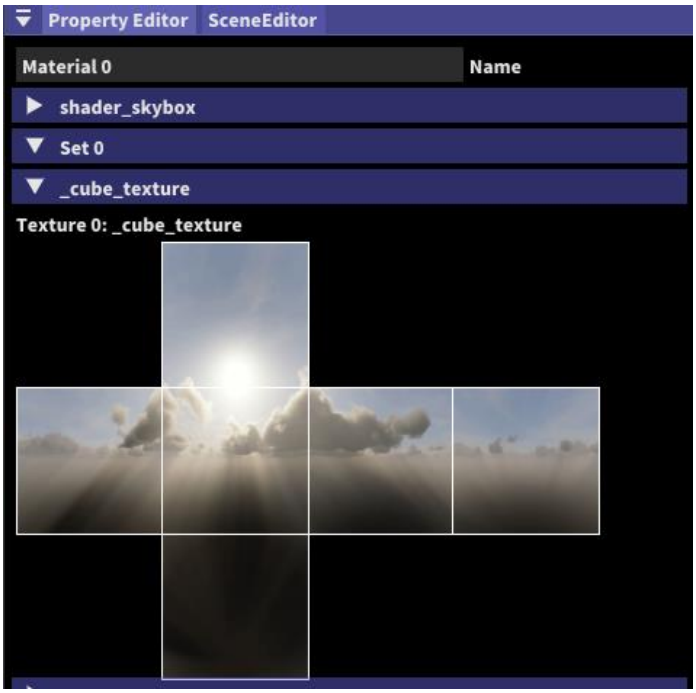

图 4-4 自定义窗口 显示效果

4.2 渲染

初始化完成后，Editor 模块导出了一个 Editor 类型的对象 editor，调用 Editor::onRender()方法，即调用容器的渲染，开始主循环。其中绝大部分的渲染内容都在 SceneEditor 的 p_scene 成员中。

editor->onRender() 中执行了 SceneEditor 的 onRender，SceneEditor 的 onRender 则执行了 p_scene 中各个成员的渲染规则。接下来则开始说明整套渲染时如何实现的。从 Scene 开始。

4.2.1 Scene

Scene 是 Resource 的子模块。其对场景中需要显示的内容进行了二次封装。该模块目前只进行了几种控件的处理：Light、Camera、Material、Texture、Model、Mesh、Scene。这些类全部继承自 SceneObject，Scene 类则是包含了除 Scene 外的所有子类，并增加了 onRender 的对外接口。

Scene 类是一个“大杂烩”，其包含了所有的需要进行渲染物体，这里以一次内置光照模型渲染一个立方体与一个用户载入模型与两个光源为例。

当 SceneEditor 第一次执行到 p_scene->onRender 时，如果根场景未被创建。则会默认创建根场景，其数据结构为双头链表，然后在根场景的末尾创建普通场景 0。一个普通场景只会包含一个默认的 Camera 类（摄像机），并将其置为 Main。当然，在这之前，引擎必须进行最为基本的初始化，即在用户没有进行任何的设置，其也能够正常运行。进行的操作大致为：创建渲染管线，创建基础的采样器，预加载创建及编译预先设置的基本着色器（诸如内置光照模型、天空盒、网格、深度等……），创建主摄像的副摄像（用于渲染深度），基础全局光，这些操作全部都在初始化第一个普通场景时执行。等待所有基本初始化操作完成后，显示给用户 UI 界面。用户按下添加立方体，引擎会为其创建一个 Cube(立方体)实例并赋予其基本的材质用于默认显示，其中实例包含了立方体的顶点相关数据，此时的数据处于原始状态。

Scene 拥有所有摄像以及 SubCamera(副摄像)的管理，渲染则是以 Camera 为单位进行。SubCamera 主要用于渲染 GBuffer，Camera 用于渲染主图。在上述 Cube 创建完成后，执行了第一次 Draw 函数。此时的函数是以 SubCamera 为视角，因为阴影需要在光源视角下进行渲染，且在主渲染之前。

Cube 是 Shape 的子类，Shape 是 Model 的子类。其中调用父类函数 setRenderCamera 来设置渲染的视角，在调用 setShader 设置临时的 Shader 来渲染阴影所需的 GBuffer。这里需要提及一下引擎 Model，Shader，Material 之间的联系：

Shader 是对 GLSL 的具象化，Material 则是 Shader 的具象化。一个 Model 需要包含一个或者多个 Mesh，一个 Mesh 则包含一个 Material，一个 Material 则包含一个 Shader，用于控制渲染。

回到刚才，设置临时的 Shader 这一过程，由于 Vulkan 任何事情都交由开发者，所以 Shader 的切换需要做大量的工作。首先识别 Shader 的 VertexInputAttribute，来获取一开始说到的原始状态的顶点数据，将其转化为指定结构体存储到 VkBuffer，这便真正渲染可以使用的顶点数据。然后检索 Model 的 Material 缓存，检查是否创建过该 Shader 类型的材质，如果不存在则创建新的 Material，之所以加入 Material 缓存机制，是因为 Material 涉及到本地图像的读取，占用显存还十分耗时。引擎的一大特点之一即是 Material 的

设计，接下来将详细介绍核心 Material 的设计实现：

4.2.1.1 Material

模型是通过执行 Shader 呈现出不同的效果，而不同的模型具有不同的材质属性需求，这就导致以 Shader 为单位对模型进行渲染时不可行的，即每次循环，I = Shader。Material 则解决了此问题，让模型渲染是以模型为单位，渲染时绑定 Material 即可。

Shader 是对 GLSL 的具象化，其中包含了 GLSL 的所有需要的属性以及接下来渲染过程中需要提供的属性，一组 GLSL 文件对应一个 Shader，Material 只需要解析 Shader 中包含的 GLSL 属性，创建并提供其所需要的属性即可。这里引擎引入一个两个新的控件 Slot 与 SlotData。这两个控件是 VkDescriptorSet 与 VkDescriptorBinding 的具象化，用于向渲染管线提供所需数据，Slot 中包含了多个 SlotData。在 Shader 解析 GLSL 之初，就会生成相应的原初 Slot，用于加速 Material 的生成。Model 的 setShader 函数实则是调用 Material 的 setShader 函数，其会对 Shader 的 Slot 进行解析，并创建相应的 Slot 复制体，此时 Slot 内部的 SlotData 是空槽。只有在 Material 执行 bind 方法时，其内部包含了 lambda 形式的数据绑定，这时会对 SlotData 进行绑定，并在渲染前进行提交更新。而在提交更新前需要绑定 Pipeline(渲染管线)，渲染管线的创建需要 RenderPass 与 PipelineLayout,PipelineLayout 则是需要基于 GLSL 中的 DescriptorSetLayout 创建，故这些操作都是在 Shader 解析完 GLSL 之后发生的，Material 只需要绑定 p_shader 提供的管线即可。

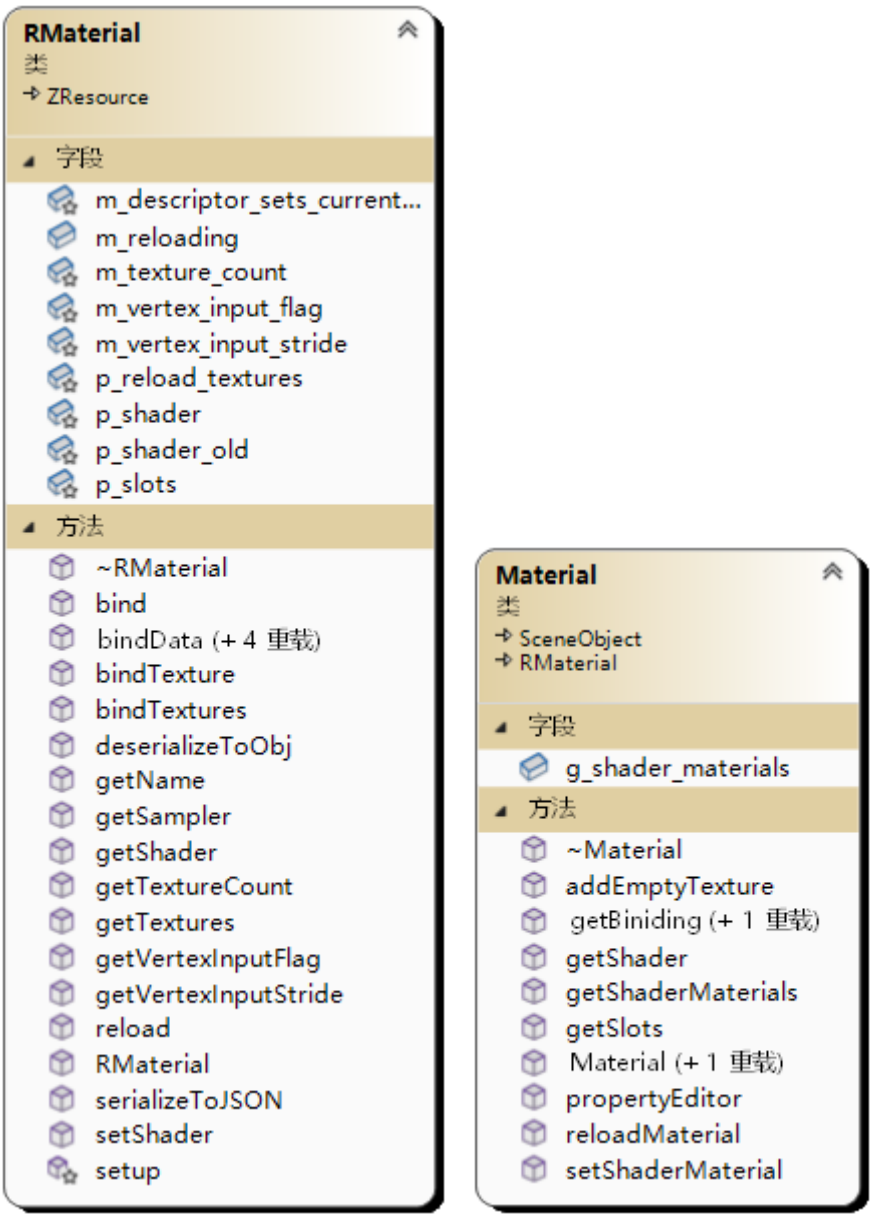

图 4-5 RMaterial 与 Material UML 类图

4.2.1.2 数据绑定方式

上述还讲到了一个非常关键的问题，也是所有引擎自由度受限的原因之一，即 Material 与数据的绑定。本引擎是使用 lambda 来解决数据绑定问题。这里先给出大致的绑定代码，以基础光照模型为例：

```
1. // SHADER_DEFAULT
2. {
3.     auto p_shader = getShader(SHADER_DEFAULT);
4.     Shader_Data_Binding_Func _func = SHADER_DATA_BINDING_FUNC_LAMBDA {
5.         auto scene = (Core::Resource::Scene*)_scene;
6.         Core::Resource::Camera* _cam = scene->getMainCamera();
7.
```

```
8.          // 绑定数据 ubo- Common
9.          if (_mat && _cam) {
10.             Core::Resource::Material* mat = ((Core::Resource::Material*)_mat);
11.             Core::Resource::Mesh* mesh =
12.                 _mesh != nullptr ? (Core::Resource::Mesh*)_mesh : ((Core::Resour
ce::Model*)_scene_obj)->getMesh(_mesh_index);
13.             const auto& _cam_data = _cam->getProperty();
14.             auto _mesh_data = mesh->getProperty();
15.             // LIGHT
16.             //-----------------------------------------------------
17.             size_t _light_index = 0;
18.             if (((Core::Resource::Scene*)((Core::Resource::Model*)_scene_obj)->g
etScene())->getShow_light()) {
19.                 for (const auto& _light : scene->getLights()) {
20.                     Core::Resource::Light* light = ((Core::Resource::Light*)_lig
ht.second);
21.                     Core::Resource::Camera* _sub_cam = (Core::Resource::Camera*)
light->getRenderCamera();
22.                     const auto& _light_data = light->getProperty();
23.                     if (!light->getHidden()) {
24.                         //-------------------------------------------------------
-
25.                         mat->bindData(0, 2, &_light_data->m_vec3_pos, _light_ind
ex)
26.                             ->bindData(0, 2, &_light_data->m_front, _light_index
)
27.                             ->bindData(0, 2, &_light_data->m_color, _light_index
)
28.                             ->bindData(0, 2, &_light_data->m_strength, _light_in
dex)
29.                             ->bindData(0, 2, &_light_data->m_type, _light_index)

30.                             ->bindData(0, 2, &_light_data->m_attenuation, _light
_index)
31.                             ->bindData(0, 2, &_light_data->m_radians, _light_ind
ex)
32.                             ->bindData(0, 2, &_light_data->m_cut_off, _light_ind
ex)
33.                             ->bindData(0, 2, &_light_data->m_cut_off_outer, _lig
ht_index)
34.                             ->bindData(0, 2, &_sub_cam->getProperty()->m_mat4_vi
ew_proj, _light_index);
35.                         mat->bindTexture(2, 0, _sub_cam->getCurrentTextureEXT(DE
PTH_ATTACHMENT), _light_index);
```

```
                        _light_index++;
                    }
                }
            }
            // COMMON
            //------------------------------------------------------
            mat->bindData(0, 0, &_cam_data->m_mat4_view)
                ->bindData(0, 0, &_cam_data->m_mat4_proj)
                ->bindData(0, 0, &_light_index)
                //------------------------------------------------------
                ->bindData(0, 1, &_cam_data->m_vec3_pos)
                ->bindData(0, 1, &_cam_data->m_zNear)
                ->bindData(0, 1, &_cam_data->m_zFar);
            // MODEL
            //------------------------------------------------------
            auto _tex_count = std::min(_mesh_data->m_texture_count, (int)mat->ge
tTextureCount());
            mat->bindData(1, 0, &_mesh_data->m_model)
                ->bindData(1, 0, &_mesh_data->m_model_inverse)
                ->bindData(1, 0, &_mesh_data->m_vec3_color)
                ->bindData(1, 0, &_tex_count)
                ->bindData(1, 0, &_mesh_data->m_shinness)
                ->bindData(1, 0, &_mesh_data->m_has_shadow);
        }
        };
    addDrawFunc(p_shader, _func);
}
```

基础光照模型包含了 3 个 Set，6 个 Binding，想要实现自定义的数据链接，只需要定义 Shader_Data_Binding_Func 类型的函数即可，该类型实际是：

```
// 数据绑定函数
// Core::Resource::Scene*
// Core::Resource::Material*
// Core::Resource::Model*
// mesh Index
// Core::Resource::Mesh _merge_no_texture
// Core::Resource::Camera
using Shader_Data_Binding_Func = void(*)(void*, void*, void*, int, void*, void*
);
```

提供了 5 个参数供用户使用，分别为 Scene、Material、Model、Index、Mesh，这 5 个参数满足了几乎所有的需求，Scene 包含了场景中所有的信息，Material 则是用于执行对 Slot 的数据填充，Model 包含了模型中的所有数据，Index 和

Mesh 则是混合使用，模型渲染是以模型为单位，而内部则是以 Mesh 为单位，Index 则是用于控制当前设置的 Mesh，而 Mesh 参数则是当整个模型为白模的情况下，引擎会合并所有的 Mesh 节点，使其变为一个 Mesh，使用一个 Material，大幅度增加白模检视性能，此时 Mesh 则会使用白模 Mesh。

上述的代码可以看到包含了大量的 bindData，该函数即是 Resource 与 Vulkan 数据交互的桥梁，接下来看一下 Slot 与 SlotData 的数据结构即明了。

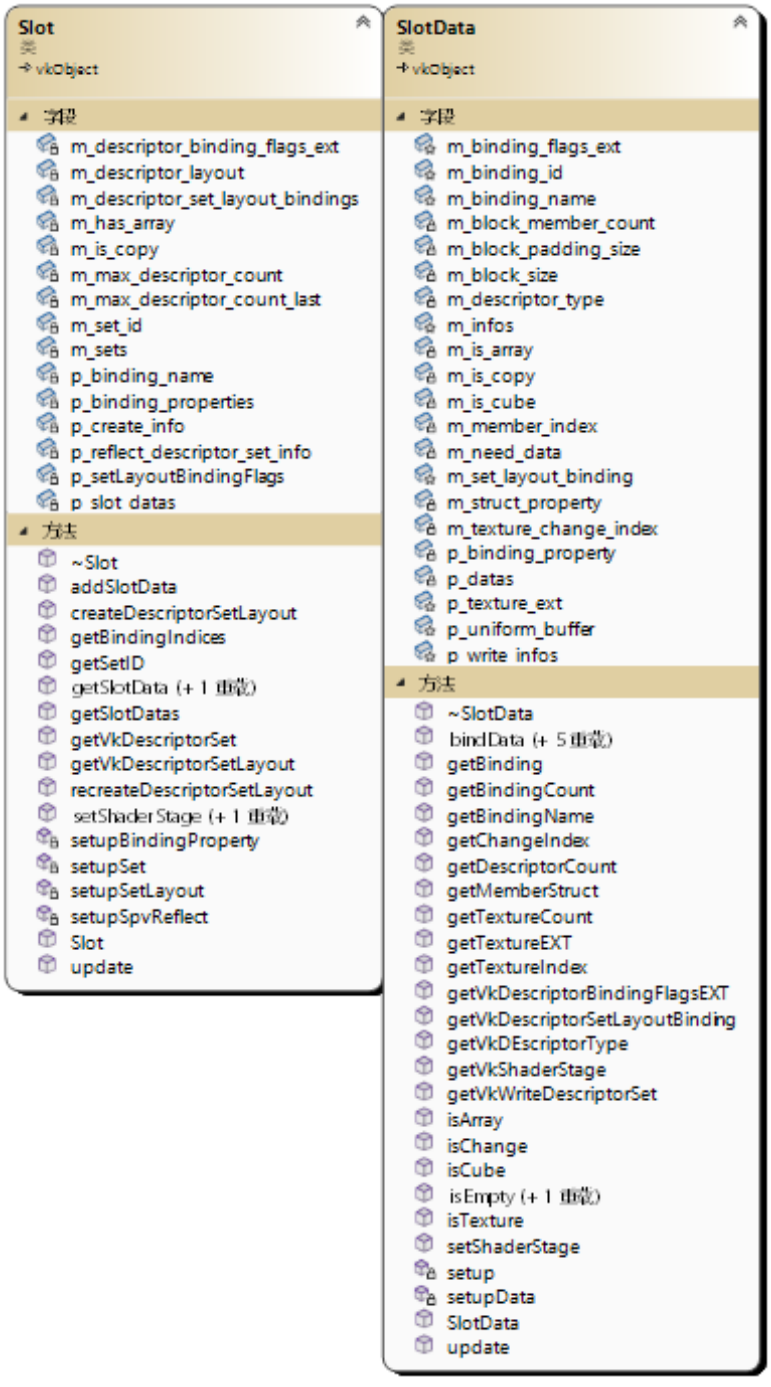

图 4-6 Slot 与 SlotData UML 类图

以下是 Material::bindData 的定义：

```
1.  RMaterial* bindData(uint32_t _set_id, uint32_t _binding_id, void* data, uint32_
t index = 0);
2.  RMaterial* bindData(uint32_t _set_id, uint32_t _binding_id, VkSampler _sampler,
 uint32_t index = 0);
3.  RMaterial* bindData(uint32_t _set_id, uint32_t _binding_id, std::string _member
_name, void* data, uint32_t _index = 0);
4.  RMaterial* bindData(uint32_t _set_id, const std::string& _binding_name, uint32_
t _member_id, void* data, uint32_t _index = 0);
5.  RMaterial* bindData(uint32_t _set_id, uint32_t _binding_id, uint32_t _member_id
, void* _data, uint32_t _index = 0);
6.  RMaterial* bindTexture(uint32_t _set_id, uint32_t _binding_id, ThirdParty::Text
ureEXT* _tex, uint32_t index = 0);
```

这些重写都必须包含_set_id,_binding_id，前者代表 SetID，后者代表 BindingID。Shader 中 DescriptorSet 对应一个 Slot，Slot 中包含多个 SlotData 即 DescriptorSetBinding。这里只是通过 id 简单地索引到需要填充数据的 SlotData。函数声明末尾都包含了提供默认值的形参，这是由于引擎支持 Bindless 特性，即无需在 glsl 文件中指定 UBO 数组长度，而是由程序执行时所用到的数量来动态分配 UBO。这需要显卡支持 Bindless，目前市面上的所有支持 Vulkan 的显卡都支持该特性，在 Vulkan 1.2 中上升为 Core 特性。

在 bindTexture 内部中，找到了需要设定的 SlotData 后，会执行其 bindData 函数，来 Slot 与 SlotData 位于 ThirdPartyManager 层，用于与 Vulkan 进行交互。这里需要提及 Vulkan 的 Shader 数据绑定的一些机制。Shader 在 Vulkan 的表现为 ShaderModule ，ShaderModul 绑定到 Pipeline 并创建 Pipeline 后，是无法修改的，ShaderModule 可以销毁，它作为了一个属性被 Pipeline 所包含，同时 Pipeline 还包含了 DescriptorSetLayout。这两个属性实际上是不会互相关联的，Shader 只是作为一段程序，DescriptorSetLayout 只是一段内存布局的描述，用于分配检验 DescriptorSet。Shader 则是根据需要的数据向管线提供的内存地址寻找，此时数据偏移就显得格外重要。这也是大部分引擎难以自定义数据绑定的原因之一，而本引擎通过反射 Shader，则不存在数据偏移问题，只需要用户将正确的数据通过正确的 SetId 与 BindingId 绑定即可。

```
1.  ThirdParty::SlotData* ThirdParty::SlotData::bindData(void* _data, uint32_t _ind
ex /*= 0*/)
2.  {
3.      // 检查是否创建 Member 内存空间
```

```
4.      if ((int)p_datas.size() <= (int)_index) {
5.          void* _ptr = malloc(m_block_size);  // 申请内存空间
6.          p_datas.push_back(_ptr);
7.      }
8.
9.      auto _member_size = 0;
10.     auto _member_offset = 0;
11.
12.     // 获取 member 的 size 与 offset
13.     {
14.         auto _member_ite = m_struct_property.second.begin();
15.         for (size_t i = 0; i < m_member_index; i++) _member_ite++;
16.         auto _member_pair = (*_member_ite).second;
17.         _member_size = _member_pair.first;
18.         _member_offset = _member_pair.second;
19.         m_member_index++;
20.         // 如果执行完一轮，则重置当前 member index
21.         if (m_member_index > m_block_member_count - 1)
22.             m_member_index = 0;
23.     }
24.
25.     char* src = (char*)p_datas[_index] + _member_offset;
26.     memcpy(src, _data, _member_size);
27.
28.     return this;
29. }
```

这是 bindData 的 UniformBuffer 绑定实现，传统的 UBO 绑定时预先设置 Struct，其中的成员映射需要与 Shader 中一致，而这里则是让成员与成员映射，无需提前定义一个 Struct，这样大大增加了数据自由度。而上述代码为了用户的使用体验良好，无需再 bindData 时设置繁琐的数据偏移，只需要按顺序绑定即可，内部的偏移计算由引擎完成。对于纹理的绑定与 UBO 有些许不同，SlotData 在初始化时，即使 Shader 中提供了纹理类型的 Uniform，但是仍然只有用户设定后才会真正意义上的添加纹理数据，这也得益于 bindless，否则对于检查严苛的 vulkan 会不停地提示 Shader 需要使用纹理而未提供纹理数据。

4.2.1.3 数据更新与渲染绘制

承接 4.2.1 后文，执行 setShader 后，引擎内部对 Material 进行了一系列的设置，设置完成后则会调用 Draw 函数，该函数非常重要，是 DrawCall 的发起，但实际实现非常常规简单。

```
for (const auto& [_mat_index, _meshes] : p_merge_meshes) {
    auto _mat = p_materials_copy[_mat_index];
    if (_mat && p_parent_scene) {
        // 此处的 material 必须是对应一个 mesh,
        auto _mesh = _meshes[0];
        _mesh->update(p_property);
        _func(p_parent_scene, _mat, this, 0, _mesh, p_render_camera);
        _mat->bind(_cmd);
        _mesh->draw(_cmd);
    }
}
```

这是每个 Draw 函数中都存在且最核心的部分，对 Mesh 进行循环遍历，每个 Mesh 都包含一个 Material Index，这样可以让 Model 管理所有的 Material，而不用将 Material 分布到每个 Mesh 中，而且也有利于 Material 的去重与复用。其中_func 则是 4.2.1.2 中提到的 Shader_Data_Binding_Func 数据绑定函数，该函数是用户必须提供的。Material::bind 则是内部对 Slot 进行循环遍历，执行 Slot::Update。Slot::Update 内部则是对 SlotData 进行循环遍历执行 SlotData::update。这样时间复杂度理论上来到了惊人的 $O(N^3)$，但实际上，Slot::Update 的执行时间可以忽略不计，其中只是单纯的将需要用到的句柄取出存储，且 Vulkan DescriptorSet 与 DescriptorSetBinding 都有上限 65535，一般的着色器 Set 与 Binding 数量相乘也难以超过 10000。

4.2.2 Camera

上述的流程皆是以 Camera 为单位，Camera 中包含了渲染的最终结果。

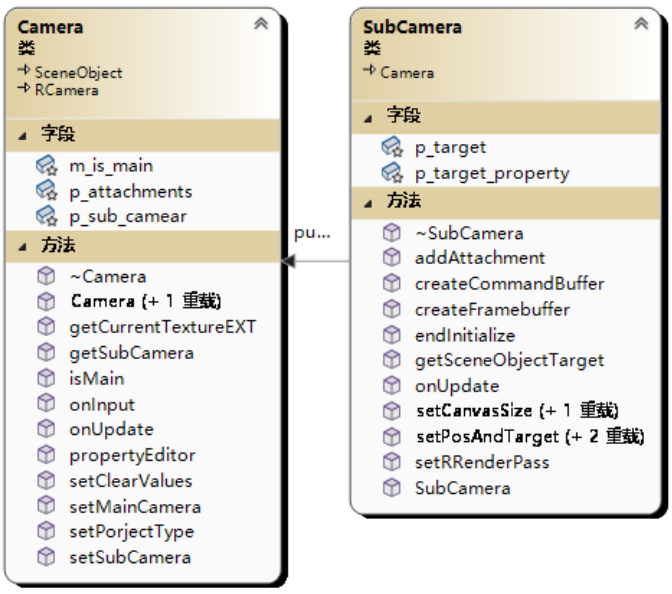

图 4-7 Camera 与 SubCamera 的 UML 类图

Vulkan 渲染是从 VkBeginRenderPass 开始，该函数需要提供 RenderPass 与 RenderPassBeginInfo，前者已经在 Scene 初始化时创建，后者则交由 Camera 管理。该结构体包含 Framebuffer，是渲染的最终导向，其中包含了 VkImage 与 VkImageView，前者是图像的内存区域，后者为内存数据视图。Camera 还包含了键鼠的控制。当一次渲染完成后，SceneEditor 则会通过 p_scene 中的 MainCamera 显示最终的画面。引擎使用的是双重缓存，至此一次完整流程的渲染结束。

4.2.3 Model

此时用户通过 UI 界面，添加了一个由路径读取的 Model，第一个收到命令的则是 Scene，会调用 ThirdPartyManager 中的 AssimpManager 对文件进行二进制的读取，此时程序处于中断状态。当完成读取后恢复，这时的 Model 包含了原始数据，此时引擎会在后台进行数据的解析，分离出引擎需要用到的顶点数据，材质等详细信息，然后以材质索引创建所有材质，然后进行材质去重；此时同步进行对 Mesh 的合并，将相同材质 ID 的 Mesh 进行合并，并将合并后的 Index 数据创建为 Vulkan 能使用的 Buffer。当材质去重完成以及 Mesh 合并完成后，设置 Model 的标志位，使其加入引擎能够渲染的资产队列中。然后该模型会和上述立方体一样经历一整套渲染流程。

4.2.4 Light

灯光是最终渲染效果中最决定性的因素，此时用户通过 UI 界面添加了一个新的点光源，第一个收到命令的仍然是 Scene，Scene 会向自身成员中添加光源。其过程与添加一个立方体相同，增加了作为光源的属性，并标识为光源。

4.3 最终呈现及销毁

上述操作完成后，即 editor->onRender 中所有 ImGui 操作执行完毕，此时会执行队列提交。4.2.2 中的 Camera 会包含一组 CommandBuffer 用于记录命令，在此时，所有的 CommandBuffer 都会被提交，其执行顺序交由硬件控制。当用户按下退出按钮时，则先调用 Core 模块的 desctroy，用于销毁引擎所产生的资源，再调用 ThirdPartyManager 的 destroy，用于销毁第三方库产生的资源。

为了方便调试追踪内存申请情况，杜绝内存泄漏问题，再 Debug 模式下，提供了带有调试信息的 new /delete 操作符，再最终程序退出时会显示内存泄漏情况。

4.4 引擎内置窗口

引擎内置了一套用于连接属性与 UI 的方法。每个 ZResource 都包含了 propertyEditor 方法用于关联 Editor 模块。Editor 内部能够调用该函数用于数据可视化。诸如 SceneCollection, PropertyEditor 窗口。这里重点解释 Material Editor。

4.4.1 Property Editor

该窗口会显示当前选中的 SceneCollection 、Material Collection 窗口中的内容，能够进行可视化的数据操作。

4.4.2 Material Editor

该窗口是作为新的特性发布，截至撰稿前可能有未测试到的会引发程序奔溃的恶性 BUG，望周知。

该功能主要由 4 个部分组成 Template、Node、Degelate、Link。

Template：是类似 Slot 的插槽性质的存在，其中包括 SlotTemplate，每个 SlotTemplate 代表着输入输出，其可携带一个 void* 类型的数据地址。

Node：代表每个 Material Editor 中的节点，其中包含了一个 Template

Delegate: 整体功能实现的核心接口。需用户自行继承实现接口功能，引擎提供了一套基本的 MaterialDelegate 的实现。

Link: Material Editor 连线存储，SlotTemplate 之间的连接被记录到此。

该部分具体实现难点主要是视觉界面的开发，要通过 ImGui 提供的自绘来对 UI 进行二次开发，当整个材质图连线完成后，点击生成按钮即可生成上述 4.2 所需要的全部内容。

4.5 引擎内置基本着色器

这些着色器是引擎最基本的默认着色方式，用户可以根据提供的着色器编写方式与数据绑定方式进行自己的着色器开发。其中的示例包含了用到的所有方法。小标题即为 Shader 文件名称，编译的结果统一存放在 resource/spirv/[Optional Shader Name]/[Shader Stage].spv 下。

4.5.1 Shader_2d_mesh

用于在 Scene 中显示参照网格，SceneEditor 视图中可关闭。这里说明一下大致的实现算法，其中涉及到部分基础的计算机图形学知识。

该着色器不需要输入顶点数据，使用内部固定数据，绘制一组三角形即一个正方形，包含了全屏幕，目前主流的网格算法都是基于屏幕空间坐标系进行的。我们需要的是在“地平面”上绘制无限网格，故要得到“地平线”t。如下图所示：

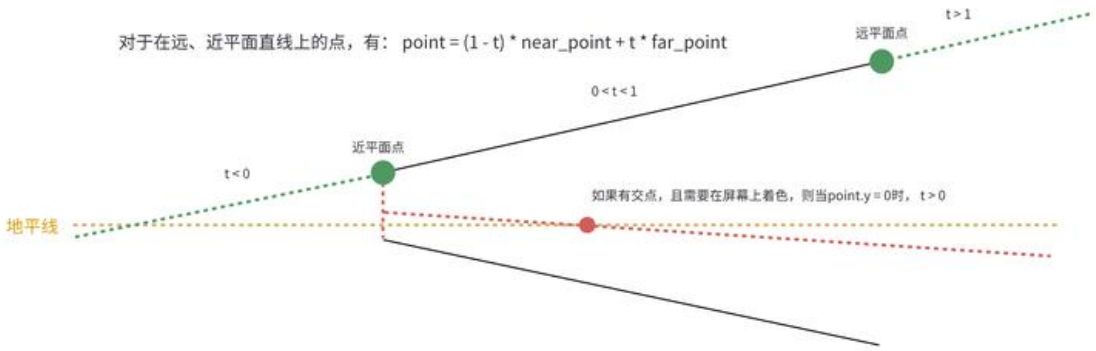

图 4-8 shader_mesh 基本公式

如图所示，我们可以根据远近平面建立射线，对于每一个像素而言，射线与地平面的交点处，t 值大于 0 才需要显示（t 小于 0 的交点在近平面左侧，相当于在视角范围的“后方”，则不显示）

所以，我们有如下两个公式：

$$y=nearPoint.y+t\times(farPoint.y-nearPoint.y)$$

$$t=-nearPoint.yfarPoint.y-nearPoint.y$$

实际上，我们要的就是在裁切空间下，四个角的远近点世界坐标。在顶点着色器中求得四个角的世界坐标，将其传入片元着色器用于计算地平面 t。之后只需要判断 t 与 0 即可判断是地平线还是地平线外。此时能够正确绘制出地平线区域而非网格。网格则是通过 uv 进行计算透明度，在网格刻度之上的，其 uv 小数部分会小于一个阈值，大于这个阈值的片段直接丢弃即可。最终效果如下：

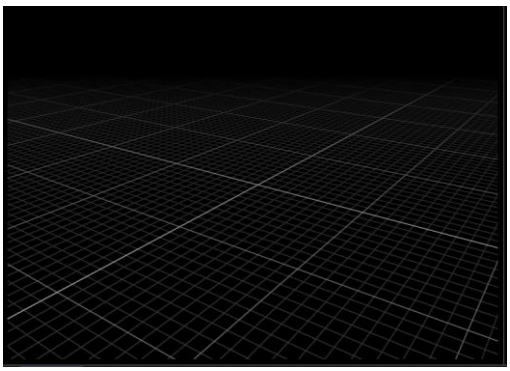

图 4-9 Shader_2d_mesh 显示效果

4.5.2 Shader_debug

用于调试 GBufer,目前是用于绘制深度图，在透视投影下，深度图是非线性的，需要将其转化为线性深度用于显示。在正交投影下，则直接显示。该着色器的主要作用是将 VK_FOMAT_R 转化为 VK_FORMAT_R8G8B8A8_UNORMA 类型。最终效果如下：

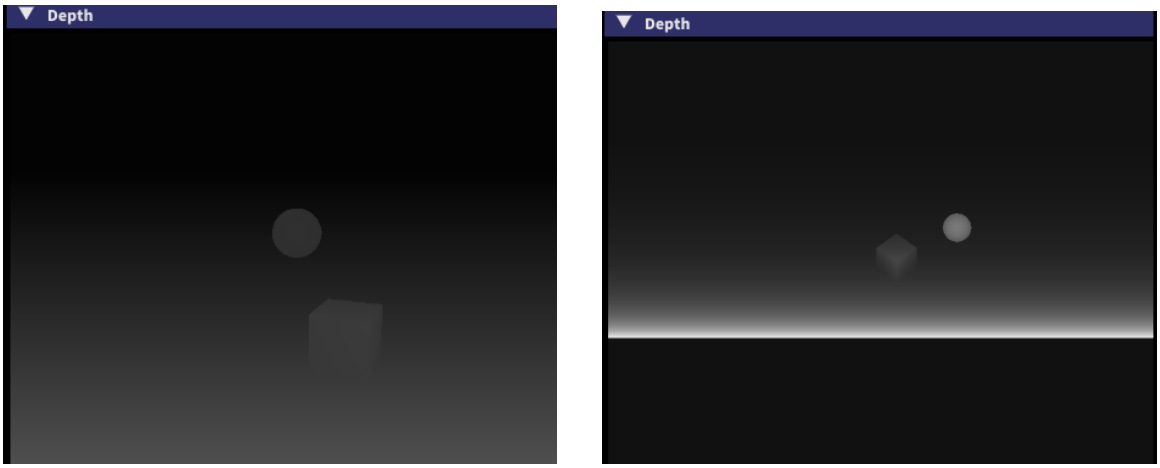

图 4-10 透视投影深度图（左）正交投影深度（右）

4.5.3 Shader_skybox

天空盒，支持 HDRI 和立方体贴图。

图 4-11 立方体贴图（1） HDRI（2） Property Editor 内容截图

4.5.4 基础光照模型（Blinn-phong）

支持阴影的 Blinn-phong 光照模型，支持最多 8 光源。

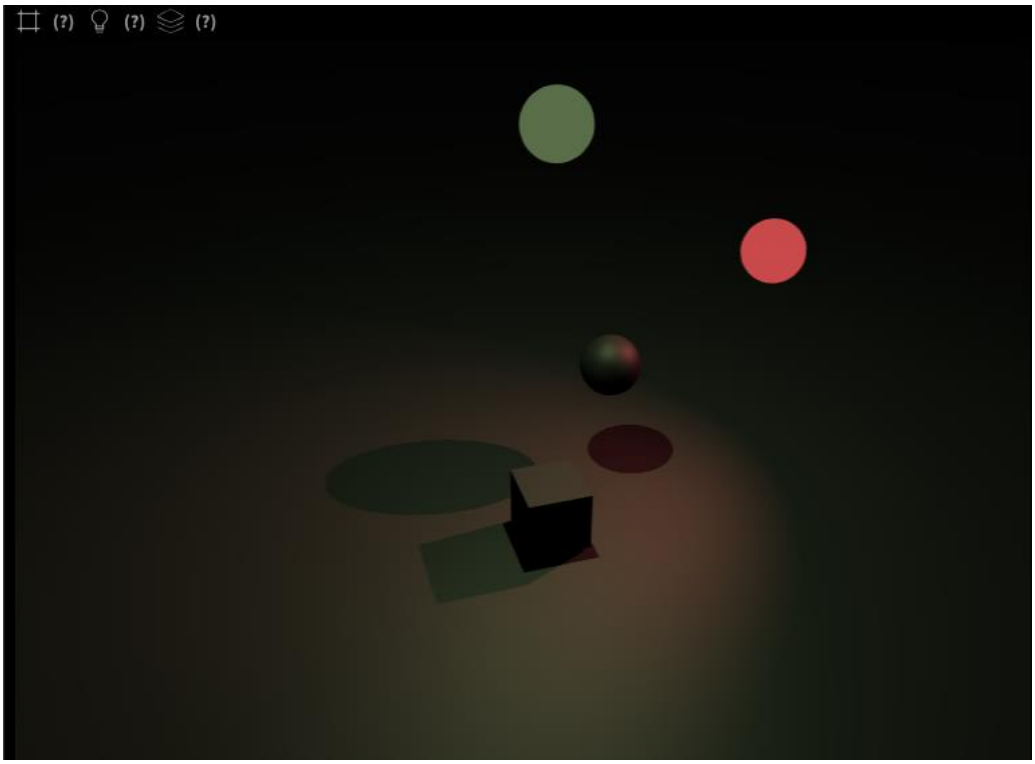

图 4-12 点光源（绿）与聚光灯（红）渲染效果

结　论

该引擎使用 C++和 Vulkan 开发，有着优秀的性能，能够在硬件较差的设备上流畅运行。目前处于起步阶段，具备了 Shader 的解析，模型的读取，渲染管线的自动搭建等特性，能够无需了解 Vulkan 繁琐的开发步骤，开盒即食，使用引擎提供的 API 进行高度自由的视觉创作。但是还不支持模型顶点修改，建模，烘焙贴图等修改保存功能，还需要继续精进打磨。Blender 的轻量级渲染引擎 eevee 可以作为一个非常好的参考，日后继续开发，一点点充盈内容。

参 考 文 献

[1] 李丽薇。浅析 Visual C++编程技巧[J]. 黑龙江科技信息，2017,（16）：190. [2017-10-10].
[2] 蒋庆磊，聂永涛，吴慧君。基于 C++的图像检测系统的设计[J]. 现代农村科技，2017,（06）：84-85. [2017-10-10].
[3] 徐洪智，张彬连，钟键。《C++程序设计》课程实验教学改革与探索[J]. 现代计算机（专业版），2017,（15）：50-53. [2017-10-10].
[4] 李峰，刘洞波。 基于翻转课堂的 C++课程教学模式探究[J]. 黑龙江教育（高教研究与评估），2017,（06）：13-14. [2017-10-10].
[5] 郜非。 基于实践的 C++互动教学模式的建立[J]. 高教学刊，2017,（08）：127-128. [2017-10-10].
[6] 徐杰恒，朱希成。 基于 Visual C++图像边缘检测的实现[J]. 工业控制计算机，2017,30（06）：17-18+20. [2017-10-10].
[7] 郑莉，董渊，张瑞丰. C++语言程序设计(第 3 版). 北京：清华大学出版社，2003
[8] 吴乃陵，况迎辉，李海文. C++程序设计. 北京：高等教育出版社，2003
[9] 谭浩强. C++程序设计. 北京：清华大学出版社，2004
[10]钱能. C++程序设计. 北京：清华大学出版社，1999
[11]和克智. C++程序设计(第 2 版). 西安：西安交通大学出版社，1999
[12] [美]H.M.Deitel，P.j.Deitel. C++程序设计教程(第 4 版). 施平安译.北京：清华大学出版社，2004
[13]陈维兴，林小茶. C++面向对象程序设计. 北京：中国铁道出版社，2004
[14]陈卫卫. C/C++程序设计教程. 北京：希望电子出版社，2002
[15]马希荣，王洪权，姜丽芬等. C++语言程序设计(二级). 北京：电子工业出版社，2005
[16]罗建军，朱丹军，顾刚等. C++程序设计教程. 北京：高等教育出版社，2004

致　　谢

首先，感谢李玉海老师在这次毕业论文中对我耐心而专业的指导，詹静老师在论文写作流程和方法方面对我的教导让我受益匪浅，从而顺利完成本次毕业论文。我认为一篇论文不能代表我在软件工程的水平，更不应该止步于此，而是要学习李玉海老师不断学习的精神和勤奋求真的治学态度，在软件工程领域开拓进取，学以致用，成为该领域的人才，不负李玉海老师的谆谆教诲。

其次，感谢本班学习委员和给我帮助的所有同学，班长和学习委员为我解答了很多我不熟悉的难题，很多同学也为我送来相关资料和论文写作技巧，让我体会到同学之间互帮互助、团结奋进的真挚友情，这种温暖、团结、进取的学习气氛，让我感动，让我一生铭记。

最后感谢我的家人，是他们多年来对我学业的支持才让我走到这一步，才使我得以顺利完成学业。