学号\_\_\_\_\_\_20175433\_\_\_\_\_\_ 密级\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**东北大学本科毕业论文**

**基于DirectX11的光栅化渲染器的设计与实现**

学 院 名 称：软件学院

专 业 名 称：数字媒体技术

学 生 姓 名：温田丰

指 导 教 师：马连博 教授

**2021 年 6 月**

基于DirectX 11的光栅化渲染器的设计与实现

|  |  |  |
| --- | --- | --- |
| 作者姓名： | 温田丰 | |
| 校内指导教师： | 马连博 | 教授 |
| 校外指导教师： |  | 工程师 |
| 单位名称： | 软件学院 | |
| 专业名称： | 数字媒体技术 | |

东 北 大 学

2021年6月

**Design and implementation of DirectX 11 based raster renderer**

by Wen Tianfeng

|  |  |  |
| --- | --- | --- |
| Supervisor: | Professor | Ma Lianbo |
| Associate Supervisor: | Engineer |  |

Northeastern University

June 2021

**郑 重 声 明**

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名： 日期：

摘 要

图形渲染是游戏开发技术当中至关重要的一部分，它决定了游戏呈现在用户面前的最终效果。近些年来，游戏行业高速发展，为了提高效率，绝大多数开发者都选择使用游戏引擎进行游戏开发，因为游戏引擎已经对游戏开发所需的大部分技术进行了封装，开发者往往只需要调用相关接口即可实现丰富的功能，而不需要从最底层的代码写起。引擎中的渲染技术也是如此，当今的主流商业游戏引擎如Unreal Engine、Unity等都将渲染技术封装起来，内置丰富的渲染算法并提供简单的接口方便开发者调用，这确实极大地提高了开发效率，但是也让开发者难以了解到渲染技术实现的底层机制，再进一步也就无法进行更多从零开始的创造。

本论文旨在论述如何从底层开始设计并实现一个基于DirectX 11的架构良好的可交互渲染器程序，介绍渲染器实现的相关技术、开发流程、系统设计、系统实现、错误及问题的解决方案以及部分渲染算法在此渲染器程序中的实现方案。

本文的研究成果是设计并实现了一个基于DirectX 11图形接口、以C++为核心编程语言的光栅化渲染器程序，并引入Dear ImGui库实现了高质量的可交互图形用户界面，提供接口来加载外部资源如模型、纹理、着色器等，通过调节材质参数的方式来调整渲染效果，并在程序当中集成了游戏开发中常用的PBR算法。用户可以通过将模型、纹理和自定义着色器资源放置在程序文件目录下，即可通过运行程序来加载并使用这些资源，以此来完成渲染效果的实现和测试。

希望通过本文的阐述，能够提供一条基于DirectX 11从底层开始开发渲染器程序的可行路径，为渲染引擎的设计和实现工作发掘思路。

**关键词：**DirectX 11；渲染器；渲染算法；C++；

**ABSTRACT**

Graphics rendering is a crucial part of game development technology, and it determines the final look of the game in front of the user. Rapid development in recent years, the game industry, in order to improve the efficiency, the vast majority of developers have chose to use game engine in game development. The game engine has already implemented most of the technology, developers only need to call the relevant interface to fulfill a variety of requirements, and do not need to write from the bottom layer of the code, so also the rendering Engine technology. Today's mainstream business game Engine such as Unreal Engine, Unity will encapsulate rendering technology, the built-in all kinds of rendering algorithms and provides a simple interface for developers to call, it greatly improves the development efficiency, but also make it hard for developers to understand the rendering technology from bottom level. You can't go any further and create more from scratch.

The purpose of this paper is to discuss how to design and implement a well-structured interactive renderer program based on DirectX 11 from the bottom level, and introduce the relevant technology, development process, system design, system implementation, solutions to errors and problems of the renderer, as well as the implementation scheme of some rendering algorithms in this renderer program.

The research result of this paper is to design and implement a rasterization renderer program based on DirectX 11 graphical interface with C++ as the core programming language, and introduce Dear ImGui library to realize a high-quality interactive graphical user interface, providing interfaces to load external resources such as models, textures, shaders, etc. The rendering effect is adjusted by adjusting the material parameters, and the PBR algorithms commonly used in game development are integrated in the system. By placing the model, texture, and custom shader resources in the program file directory, user can run the program to load and use these resources to achieve the rendering effect and test it.

It is hoped that through the elaboration of this paper, it can provide a feasible path to develop renderer program from the bottom level based on DirectX 11, and explore ideas for the design and implementation of rendering engine.

**Key words:** DirectX 11; Renderer; Rendering Algorithm; C++

目 录

[摘 要 II](#_Toc19304)

[ABSTRACT III](#_Toc22538)

[第1章　绪 论 1](#_Toc23936)

[1.1 课题研究背景与意义 1](#_Toc24502)

[1.2 国内外研究现状 1](#_Toc2550)

[1.3 论文研究内容 2](#_Toc32336)

[1.4 论文的组织结构 2](#_Toc32336)

[1.5 本章小节](#_Toc32336) 3

[第2章　相关技术 5](#_Toc28980)

[2.1 图形渲染管线 6](#_Toc32544)

[2.2 DirectX 11 7](#_Toc27733)

[2.3 Dear ImGui 用户界面 7](#_Toc21972)

[2.4 Assimp 模型资源加载 7](#_Toc12351)

[2.5 基于物理的渲染 8](#_Toc31185)

[2.6 本章小节](#_Toc12351) 10

[第3章　需求分析](#_Toc28980) 11

[3.1 功能需求 1](#_Toc23433)1

[3.2 用户需求 1](#_Toc16266)2

[3.3 性能需求 1](#_Toc15084)3

[3.4 本章小节 1](#_Toc15084)3

[第4章 系统设计 1](#_Toc26403)5

[4.1 总体架构设计](#_Toc24586) 15

[4.1.1 基本逻辑 1](#_Toc2540)5

[4.1.2 顶层模块概述 1](#_Toc2540)6

[4.2 实体-组件架构设计 1](#_Toc24586)6

[4.2.1 实体-组件模式概述 1](#_Toc2540)6

[4.2.2 实体 1](#_Toc2540)7

[4.2.3 组件 1](#_Toc2540)7

[4.3 图形渲染模块设计 1](#_Toc20993)7

[4.3.1 渲染窗口 1](#_Toc2540)8

[4.3.2 图形渲染管理器 1](#_Toc2540)8

[4.4 场景管理模块设计 2](#_Toc17031)0

[4.5 输入控制模块设计 2](#_Toc17031)0

[4.6 资源结构设计 2](#_Toc17031)1

[4.6.1 顶点缓冲和索引缓冲设计](#_Toc2540) 21

[4.6.2 网格设计](#_Toc2540) 22

[4.6.3 常量缓冲设计](#_Toc2540) 22

[4.6.4 纹理设计](#_Toc2540) 22

[4.6.5 采样器状态设计](#_Toc2540) 23

[4.6.6 结构缓冲设计](#_Toc2540) 23

[4.6.7 着色器设计](#_Toc2540) 23

[4.6.8 着色器资源参数设计](#_Toc2540) 24

[4.6.9 材质设计](#_Toc2540) 24

[4.7 实体渲染系统设计 2](#_Toc17031)4

[4.7.1 实体初始化阶段](#_Toc2540) 25

[4.7.2 实体渲染阶段](#_Toc2540) 25

[4.8 用户界面模块设计 2](#_Toc17031)6

[4.9 本章小节 2](#_Toc20993)7

[第5章 系统实现](#_Toc26403) 29

[5.1 总体架构实现 2](#_Toc24586)9

[5.1.1 程序入口](#_Toc2540) 29

[5.1.2 初始化](#_Toc2540) 30

[5.1.3 运行](#_Toc2540) 30

[5.2 实体-组件架构实现](#_Toc20993) 31

[5.2.1 实体实现](#_Toc2540) 31

[5.2.2 组件实现](#_Toc2540) 34

[5.3 图形渲染模块实现](#_Toc17031) 38

[5.3.1 渲染窗口实现](#_Toc2540) 38

[5.3.2 图形渲染管理器实现](#_Toc2540) 39

[5.4 场景管理模块实现 4](#_Toc17031)3

[5.4.1 场景管理器初始化](#_Toc2540) 43

[5.4.2 场景管理器更新](#_Toc2540) 43

[5.5 输入控制模块实现 4](#_Toc17031)4

[5.5.1 输入控制模块总体实现](#_Toc2540) 44

[5.5.2 鼠标输入控制模块实现](#_Toc2540) 45

[5.5.3 键盘输入控制模块实现](#_Toc2540) 46

[5.6 资源结构实现 4](#_Toc17031)7

[4.6.1 顶点缓冲和索引缓冲实现](#_Toc2540) 47

[4.6.2 网格实现](#_Toc2540) 48

[4.6.3 常量缓冲实现](#_Toc2540) 48

[4.6.4 纹理实现](#_Toc2540) 51

[4.6.5 采样器状态实现](#_Toc2540) 52

[4.6.6 结构缓冲实现](#_Toc2540) 53

[4.6.7 着色器实现](#_Toc2540) 54

[4.6.8 着色器参数实现](#_Toc2540) 55

[4.6.9 材质实现](#_Toc2540) 56

[5.7 实体渲染系统实现 6](#_Toc17031)0

[5.8 用户界面模块实现 6](#_Toc17031)1

[5.8.1 用户界面总体架构实现](#_Toc2540) 61

[5.8.2 场景对象结构面板实现](#_Toc2540) 63

[5.8.3 细节信息面板实现](#_Toc2540) 64

5.8.4 窗口工具栏实现64

[5.9 渲染算法实现 6](#_Toc17031)5

[5.9.1 前向渲染实现](#_Toc2540) 65

[5.9.2 基于物理的渲染实现](#_Toc2540) 70

[5.10 本章小节](#_Toc17031) 71

[第6章 系统测试](#_Toc26403) 73

[6.1 系统环境](#_Toc24586) 73

6.1.1 系统调试阶段测试效果73

6.1.2 系统发布阶段测试方案73

[6.2 系统界面](#_Toc20993) 74

[6.3 PBR效果测试](#_Toc20993) 74

[6.4 本章小节](#_Toc20993) 74

[第7章 总结和展望](#_Toc26403) 77

[7.1 工作总结](#_Toc20993) 77

[7.2 未来展望](#_Toc20993) 77

[参考文献](#_Toc18532) 78

[致 谢 7](#_Toc28784)9

第1章　绪 论

## 1.1 课题研究背景与意义

图形渲染一直是游戏、动画、和影视等行业不变的主题，因为这些领域的产品能够给用户最直观的反馈就是画面表现。怎样利用设备来渲染出理想的效果始终是开发人员研究的主要方向。如今，随着软件、硬件技术的不断发展，计算机算力高速增长，渲染算法的不断更迭，游戏画面渲染的研究已经十分成熟，计算机GPU（Graphics Processing Unit，图形处理器）等硬件的发展，使开发者可以使用更加丰富的手段来高效实现渲染效果。

图形渲染发展至今，虽然已经出现了光线追踪这样新型的渲染方案，但是光栅化渲染仍然是画面渲染阵营的主力军，作为从图形渲染诞生时就存在的渲染方案，光栅化已经几乎可以满足任何的渲染需求，而且目前许多渲染算法也是基于光栅化渲染管线创造的。所以，对于基于光栅化的渲染器的实现进行探究仍有重要意义。

渲染器是游戏引擎当中的核心组成部分。渲染器模块是否拥有合理的资源整合及利用方案，完善、高效、可编程可定制的渲染管线，是否内置优秀的渲染算法很大程度决定了一个游戏引擎能够实现的渲染效果。

独立的渲染器程序则只关注和图形渲染开发相关的技术实现，所以相比完整的游戏引擎，渲染器可以提供渲染相关的更完善、更丰富的功能，也能够在开发过程中探索更先进、更高级的技术实现。

## 1.2 国内外现状

经过二十多年的发展，长时间的技术积淀，国外在渲染器开发方面的技术已经非常成熟，而且技术的更新迭代也十分迅速。市面上主流的渲染器如Vray、Corona、Octane和Arnold等，也都是由国外技术公司开发，功能都十分强大且各具特色。

鼎鼎大名的Vray渲染器是最受业界欢迎的渲染引擎。基于Vray 内核开发的软件包括3ds Max、Maya、Sketchup、Rhino、CINEMA 4D等，其为不同领域的优秀3D建模软件提供了高质量的图像和动画渲染功能。除此之外，VRay也可以提供单独的渲染程序，便于用户实现自定义的渲染需求。

Corona渲染器始于2009年，是布拉格捷克技术大学的一个独立学术项目，他们三个人就是corona渲染器的三巨头，第一个就是参与项目的学生，后来他的导师和他一个朋友，三人就开创了一家公司。经过多年的沉淀，Corona渲染器已经发展成为一个全新的商业项目。Corona渲染器是一款高性能无偏差，照片级物理渲染器，因为它的材质属性跟现实世界中物理现象基本保持一致。所以它的超写实效果有着极强的优势。Corona渲染器在灯光上是基于现实效果，所以非常柔和，不像Vray 渲染器，灯光比较生硬，更适合于商用效果[1]。

国内在渲染技术相关的发展起步要比国外晚很多，所以发展速度也要慢一些。但国内最近已经有团队实现了技术实践的飞跃，如D5渲染器，它是由国内的软件开发团队（南京维伍网络科技有限公司）开发的高质量图像、动画渲染器。D5在建筑设计方面有较为出色的表现，已经成为目前该领域中最受欢迎的渲染软件之一。最近，D5渲染器更新到了1.6版本，开发团队与NVIDIA方面相合作，将RTX支持引入到了软件中。D5渲染器作为一款较新的渲染器，从去年公测至今已经获得了大量专业用户的青睐[2]。

## 1.3 论文研究内容

本文的重要研究内容有以下几个方面：

1. 探索开发基于DirectX 11的光栅化渲染器所需的相关技术，涉及图形渲染管线知识、DirectX应用程序接口、资源处理库和图形界面库的引用等。
2. 对于预期完成后的系统程序进行需求分析，在各个层面讨论系统应该具有怎样的功能，能够满足怎样的需求。
3. 针对系统实现所需的相关技术配置开发环境。研究DirectXTK、Assimp、Dear ImGui等开发库如何正确的引入系统，Visual Studio如何调试、发布程序等。
4. 对系统架构进行设计，研究实体-组件模式的实现方案。
5. 研究HLSL着色器的编写，前向渲染、基于物理的渲染在系统当中的实现方案。

## 1.4 论文的组织结构

第1章，绪论。介绍基于DirectX 11的光栅化渲染器开发的研究背景和意义、图形渲染技术和渲染器开发在国内外的发展现状以及本论文的组织结构。

第2章，相关技术。介绍系统开发所需的相关技术，包括图形渲染管线的实现、DirectX 11应用程序接口、Dear ImGui和Assimp开发库以及系统应用的渲染算法。

第3章，需求分析。根据系统实现目标进行需求分析，从功能需求、用户需求和性能需求三个方面进行讨论。

第4章，系统设计。从系统的整体到内部的各个模块的进行结构设计，将系统设计为由图形渲染模块，场景管理模块和输入模块组成，并对定义的资源数据结构进行具体设计。

第5章，系统实现。依据系统设计实现了对应的各个模块及其内部构成。

第6章，系统测试。对系统的各个操作功能、运行效率和图形渲染结果进行介绍。

第7章，总结与展望。对工作做了简要的总结，并对后续工作提出了设想。

## 1.5 本章小节

本章节为绪论，首先介绍了基于DirectX 11的光栅化渲染器的设计与实现课题的研究背景和意义，阐述了渲染器在当今的游戏、动画等行业的重要性。然后对渲染技术和渲染器在国内外的发展情况进行介绍，最后对论文的主要工作和组织结构进行了说明。

第2章　相关技术

## 2.1 图形渲染管线

图形渲染管线是实时渲染的核心组件。渲染管线的功能是通过给定虚拟相机、三维场景物体和光源等场景要素来渲染一副二维的图像。渲染管线是实时渲染的基础工具，它的基本构造包括四个阶段：应用层传入数据、几何图形处理、光栅化和像素的处理，每个阶段由可能由几个子阶段构成。应用层是在CPU上执行从程序，它将渲染图形所需的数据传入管线当中，然后几何阶段对传入的数据进行变换、投影等几何处理，决定要绘制的内容，光栅化阶段则根据图形输入获取其在屏幕空间覆盖的像素，最后一个阶段对得到的像素进行处理，输出目标颜色[3]。

Direct3D渲染管线的结构如图2.1所示，图中左侧实线框表示该阶段是可编程的，虚线框表示该阶段仅是可配置的。

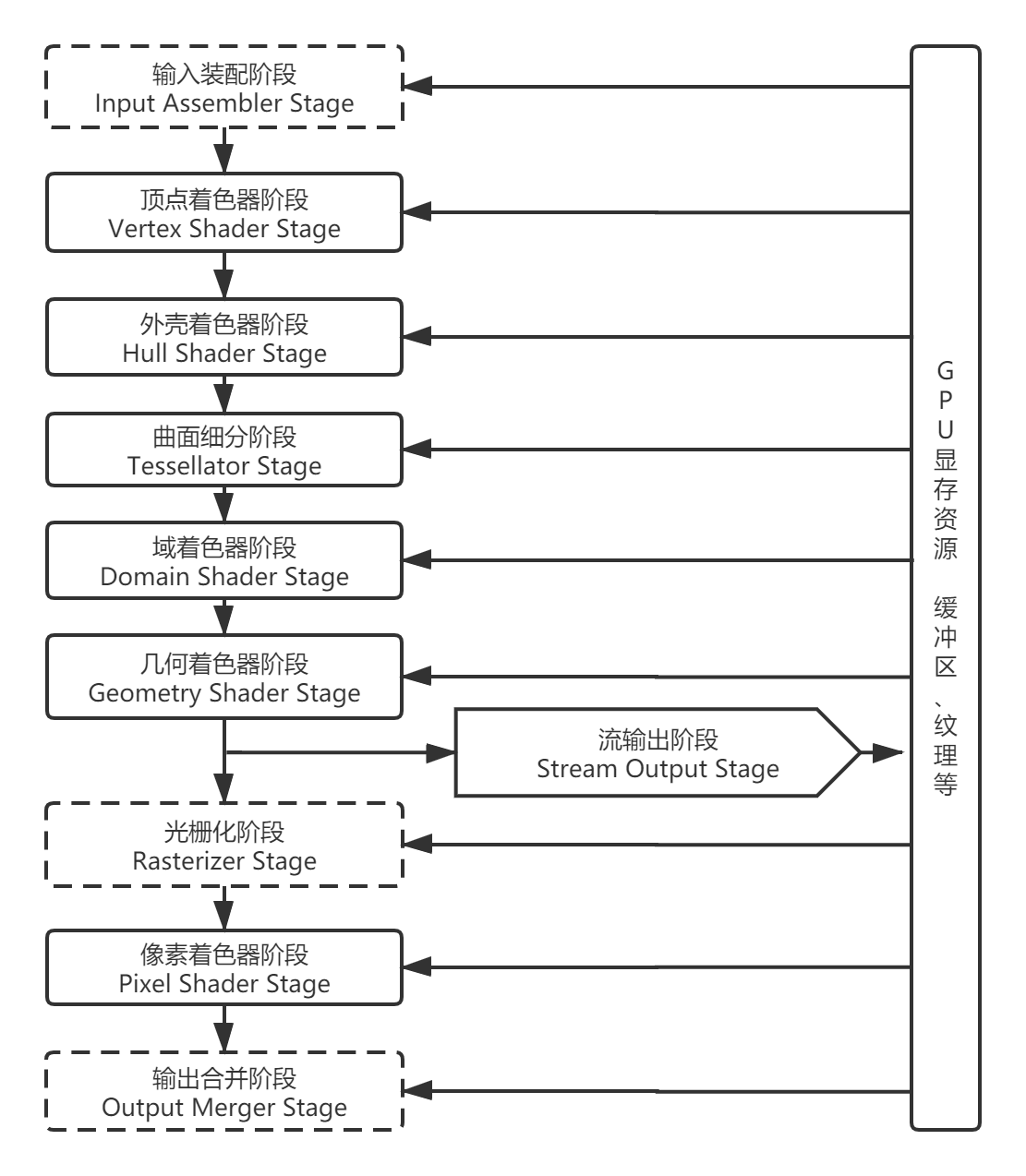


图2.1 Direct3D 11图形渲染管线

输入装配阶段：输入装配阶段是图形渲染管线的入口点。在这里为要渲染的对象提供顶点和索引数据，输入装配阶段会从显存中读取并将它们进行组合成为图元数据（点、线段和三角形），然后将结果输送到顶点着色器阶段。

顶点着色器阶段：顶点着色器阶段负责处理输入装配阶段输送进来的图元，对每个顶点进行处理。顶点着色器是一个在GPU当中执行的函数程序，它的输入是一个顶点属性构成的结构，顶点结构中至少包含顶点的位置信息，顶点着色器对每个顶点进行处理，并输出处理之后的顶点结构。

曲面细分阶段：曲面细分阶段利用镶嵌化技术对三角面进行细分，可以增加几何体表面的细节。曲面细分阶段是渲染管线可选的一个阶段，它是由外壳着色器、镶嵌器和域着色器组成的。镶嵌器是渲染管线的一个固定功能的阶段，它只能和镶嵌着色器一起使用，而外壳着色器和域着色器是可编程的。借助曲面细分技术可以实现细节层级机制（LOD），它可以根据物体和相机之间的距离动态的修改几何表面的细节程度，提高渲染效率。

几何着色器阶段：几何着色器是渲染管线一个可选的阶段，负责处理完整的图元（点、线段和三角形）。几何着色器的输入是单个对象及其相关的顶点，对象通常由点、线段或三角形组成，几何着色器可以在其中添加或删除几何图形，甚至将图元进行替换。几何着色器被设计用于修改传入的数据或者根据输入创建有限数量的拷贝。

流输出阶段：渲染管线的标准用途是通过顶点着色器发送数据，在管线中经过处理，对得到的三角形进行光栅化，并使用像素着色器对其进行着色。流输出阶段提供了在管线中间访问数据的功能，在顶点数据经过顶点着色器、曲面细分和几何着色器之后，可以通过流输出阶段输出，得到一个有序的顶点数组。

光栅化阶段：光栅化阶段负责将之前处理得到的图元转换为光栅化图像，它是像素组成的二维数组，光栅化阶段确定应该渲染的像素，并将每个像素包含的信息传递到像素着色器阶段。光栅化阶段首先要对输入的顶点信息进行透视除法和视口变换，然后对顶点属性进行插值，得到像素片段的位置以及其它属性信息。

像素着色器阶段：像素着色器接受的输入为光栅化对顶点信息插值之后得到的片段，像素着色器会对根据片段具有的属性对其进行处理，最后输出片段对应的像素颜色。像素着色器还具有丢弃片段的能力，即不进行输出。

输出合并阶段：输出合并阶段可以控制最终显示的像素颜色。输出合并阶段不可编程，但是可以通过设置渲染管线状态来控制其行为。设置深度缓冲状态可以根据像素的的前后关系来判定输出结果，设置模板缓冲状态可以决定哪些像素能够最终显示，而通过设置混合状态则可以决定先后渲染的像素以怎样的方式混合颜色，可以实现半透明效果[4]。

## 2.2 DirectX 11

DirectX 11是主流的图形API——DirectX的第11个大版本。DirectX，是微软提供的应用程序接口集(APIs)，被设计为在运行 Windows 操作系统平台上用来提供给开发人员控制硬件的底层接口。它的各个组件提供了访问不同硬件的能力，包括图形(显卡)，声音(声卡)，GPU，输入设备以及所有的标准接口[5]。

Direct3D是Windows平台的图形应用程序接口。Direct3D是DirectX的一部分，主要用于渲染3D图形，如果显卡允许，Direct3D可以使用硬件加速技术来加速部分或整个渲染管线的运作。Direct3D支持许多3D图形硬件的高级图形功能，包括Z-buffering、W-buffering、模板缓冲、空间抗锯齿、alpha混合、颜色混合、mipmapping、纹理混合、可编程的HLSL着色器等等。

## 2.3 Dear ImGui用户界面

Dear ImGui 是一个用于C ++的无膨胀图形用户界面库。 它输出优化的顶点缓冲区，可以随时在启用3D渲染管线的应用程序中进行渲染。它不依赖于渲染器，独立、快速且可移植。Dear ImGui 特别适合集成到游戏引擎（用于工具），实时3D应用程序（如渲染器），全屏应用程序，嵌入式应用程序或非标准操作系统功能的控制台平台上的任何应用程序中。Dear ImGui 的核心是独立于几个与平台无关的文件，其代码库可以轻松地在应用程序/引擎中进行编译，其核心代码都是存储库根文件夹中的文件（imgui.cpp，imgui.h，imgui\_demo.cpp，imgui\_draw.cpp等）。Dear ImGui不需要特定的构建过程，可以直接添加到项目当中[6]。

## 2.4 Assimp 模型资源加载

Assimp是Open Asset Import Library（开放的资产导入库），它能够加载多种不同的模型文件格式，将所有的模型数据加载为Assimp的通用数据结构。由于其通用数据结构是固定的，所以无论导入任何文件格式的模型，我们都可以用同一种方式访问所需的数据信息。

通过Assimp导入模型，会将整个模型文件载入一个场景（Scene）对象，包含导入模型的所有数据。场景会载入为一系列的节点（Node），每个节点包含场景对象中所存储数据的索引，每个节点可以有任意数量的子节点。Assimp数据结构的模型如图2.2所示。

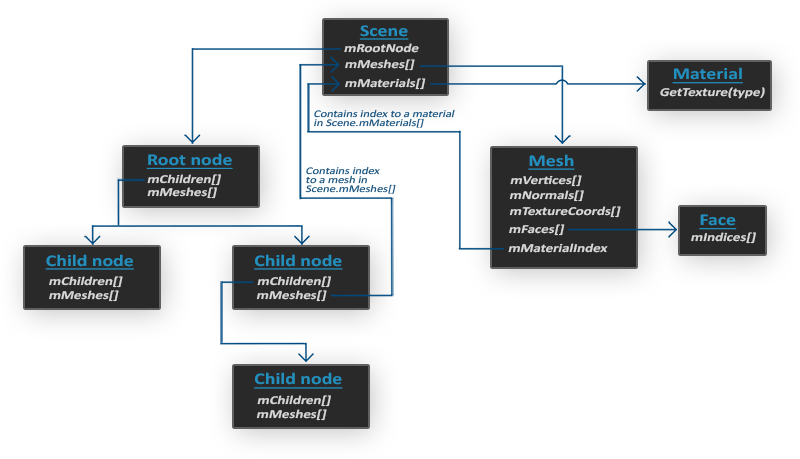


图2.2 Assimp数据处理模型

像材质和网格数据一样，所有的场景/模型数据都包含在Scene对象中。Scene对象也存储了对场景根节点的引用。场景的根节点可能拥有子节点，子节点拥有指向场景对象mMeshes数组中存储的网格数据的索引。节点中的mMeshes数组存储的只是场景中网格数据的索引值，Scene对象中的mMeshes数组才存储真正的网格（Mesh）对象。

网格（Mesh）对象包含了渲染模型所需的所有相关数据，如顶点位置坐标、法线向量、纹理坐标、面和材质信息等。一个网格包含了多个面（Face），面代表的是几何体的渲染图元（点、线、三角形、四边形）。面包含了组成图元的顶点索引，也就是构造索引缓冲所需的数据信息[7]。

## 2.5 基于物理的渲染

基于物理的渲染（Physically Based Rendering，PBR）指的是使用基于物理原理和微平面理论进行建模的着色/光照模型，使用现实中测量的物体表面参数，通过各种数学方法推导模拟出一系列渲染方程，并依赖计算机硬件和图形API渲染出拟真画面的技术。

在渲染算法的实现中，需要满足一定的要求才能够称之为基于物理的渲染。

渲染算法的光照模型应基于微平面理论（Microfacet Theory）。微平面理论认为每个可见的物体表面点都是由很多朝向不同的微小平面组成的，它们的法线方向不一，当光线射向这些平面，会在不同的方向反射光。物体的表面越粗糙，反射的光线朝向就更无序，使其反映的环境细节更加模糊，导致更暗但是也更广泛的高光效果；物体的表面越光滑，反射的光线朝向就更统一，能够更清晰的反映环境细节，导致更亮但是也更窄小的高光效果。在微观的角度来讲，任何物体的表面都是粗糙的，没有物体表面是完全光滑的，但是单个微平面的方向是随机的，因此对其以统计分布的方式进行建模是有意义的。基于物理的渲染使用一个粗糙度（Roughness）参数来定义物体表面的粗糙程度，它表示微平面和半角向量（Halfway Vector）方向一致的概率，半角向量是光线向量和视线向量的中间向量[8]。

渲染算法的光照计算应该符合能量守恒（Energy Conservation）。即出射光的总能量不能够超过入射光的总能量，随着物体表面粗糙度的提高，完成镜面反射的范围增大，则会降低其整体的亮度。

当光线射入到物体的表面，通常会发生反射（reflection）和折射（refraction）两种现象。反射部分即为镜面反射（specular），光线直接从物体表面反射出去，不进入物体内部。折射部分则进入物体内部，其中一部分可能被物体吸收，剩余的则继续前进，在物体内部持续碰撞，最终没有被吸收而射出的光线为散射（Scattering）光线。一般情况下，基于物理的渲染会简化折射部分光线的计算，即假设其均被吸收，只计算镜面反射部分；但是也有将观察尺度继续缩小的，讨论散射部分的光线计算，这被称为次表面散射技术（Subsurface Scattering）。

基于物理的渲染应基于渲染方程（Render Equation）实现。渲染方程可以模拟光的能量在物体表面的运作，它基于能量守恒定理。基于物理的渲染使用的渲染方程称为反射方程（Reflectance Equation），其基本形式见公式（2.1）。

(2.1)

渲染方程公式中的项即为双向反射分布函数(Bidirectional Reflectance Distribution Function, BRDF)。BRDF的作用是计算每个单独的光线在物体表面对最终反射光线的贡献程度。系统使用的BRDF为Cook-Torrance BRDF，它是游戏业界目前最主流的基于物理的镜面反射BRDF模型。

## 2.6 本章小节

本章介绍了系统实现相关的关键技术，对于DirectX 11图形渲染管线结构、基于物理的渲染等图形渲染相关知识进行了比较详细的介绍。对于开发涉及的代码库、应用程序接口等进行了比较浅显的介绍，如DirectX 11、Assimp和Dear ImGui的核心功能、适用情况和主要优点等。这些关键技术的具体使用将在后续章节当中阐述。

第3章 需求分析

基于DirectX 11的光栅化渲染器是应用于Windows平台的交互式应用程序。其运行逻辑基于模型网格、着色器和材质之间的关系。用户可以通过程序加载模型资源并显示，为其指定预置或自定义的着色器资源，通过材质系统来调节着色器参数，进而实现最终的渲染效果。系统在底层封装渲染所需的数据结构，架构出合理的渲染管线处理流水线，并以交互界面和键鼠设备操作的方式提供外部接口，实现基本功能，并在此基础上添加更丰富的操作逻辑，优化用户体验，扩展程序功能。

## 3.1 功能需求

本系统主要用于进行渲染算法的实现和测试和对渲染效果的调节。运行程序之后，需要创建或者加载渲染所需的模型网格，模型网格生成之后作为对象存在于场景当中，用户可以通过显式的层次结构面板来选择需要处理的模型网格对象。在选择指定的对象之后，需要细节信息面板来显示对象拥有的具体信息，并且提供参数输入和调节接口来便于用户调整渲染效果及场景内容。对于每个对象，其拥有的功能通过组件来实现，对其添加不同类型的组件，则可以产生不同的效果。在针对场景内容本身的处理之外，系统还需提供对于渲染器设定和渲染通用选项的调整操作支持，便于用户定制效果。

基于以上的分析进而归纳出系统的功能需求有以下几个方面：

1. 系统提供新建场景、打开场景和保存场景功能。用户可以通过运行程序来创造一个新的空场景，也可以通过工具栏操作来打开并加载一个指定格式的场景文件，在对场景进行一定的操作之后，可以通过工具栏操作来保存一个自定义名称的场景文件。
2. 具备场景信息的显示和处理功能。通过用户界面来展示场景内容信息，层次结构面板（Hierarchy）显示所有的场景对象；信息细节面板（Inspector）展示选中对象的详细信息，且用户可以对对象的组件信息进行自由调节；窗口工具栏（ToolBar）提供全局的处理和操作接口，用户可以借此完成针对整个场景的操作或者调节渲染器的设定。
3. 提供资源处理功能。工具栏提供模型加载功能，在针对渲染对象的细节调整时，可以改变其网格资源、纹理资源和着色器资源等。
4. 场景对象的组件处理功能。在生成基本对象之后，可以通过为其添加组件使其拥有新的功能，如添加材质组件可以提供自定义材质属性的功能。

## 3.2 用户需求

用户运行程序，打开一个新的临时空场景。用户可以通过点击工具栏按钮选择生成基本模型（常用简单模型如立方体、球体等）或者从文件当中加载自定义的模型。将模型加载到场景当中后，其名字会出现在场景层次结构信息面板（Hierarchy）当中，用户可以在面板中选定创建的模型对象，之后其具体信息就会出现在信息细节面板（Inspector）当中，具体出现的信息类型会和选定对象具有的组件相关。

属性组件（Attributes）包含对象具有的基本信息如名称、标签、是否激活等等。变换组件（Transform）控制对象在空间当中的变换，包括位置、旋转和缩放信息，用户可以调节各个参数值来控制对象在场景当中的变换。作为渲染器程序，最重要的组件即为材质组件（MaterialManager），材质组件包含对象具有的表面材质信息，它关联渲染对象背后的着色器内的资源参数，用户可以通过调节材质组件参数来调整渲染效果。其余组件这里暂不介绍。通过用户操作，调整各类参数，达到了所需的渲染效果，用户可以选择保存场景至文件当中，通过工具栏提供的功能来加载文件中保存的场景信息。在主要功能之外，用户还可以对渲染器程序进行系统设置，根据需求自定义渲染器功能和一些渲染相关的全局设置。

系统用例图如图3.1所示。

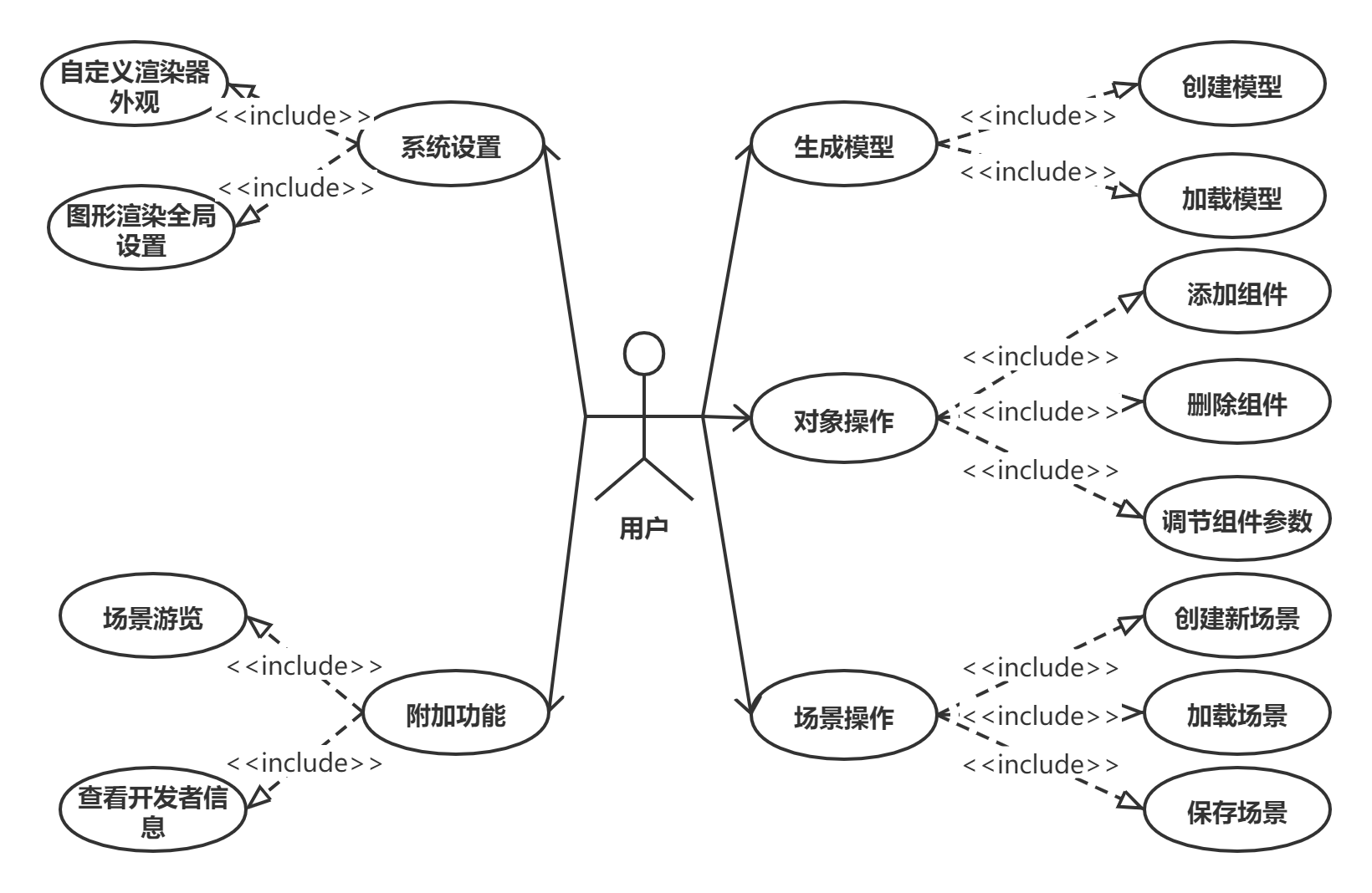


图3.1 系统用例图

## 3.3 性能需求

通用性：渲染相关的资源如模型、纹理和着色器等等都是多种多样的，系统需要拥有对应多种类型的功能接口，拥有良好的分辨和处理能力。

扩展性：由于系统的实现永远不会做到完美，所以在不断的开发迭代过程中，需要系统具备健康的底层架构，拥有良好的扩展性，这样在之后的功能拓展和更新过程中，才能维持系统的稳定，保证开发效率。

易用性：系统需提供清晰简洁的用户界面，简单合理的交互逻辑，优化用户体验，并提供渲染器所需的常用功能。如果操作有误或者运行错误，应给出操作或错误提示。

鲁棒性：系统在设计和实现时应注意降低模块和模块、功能和功能之间的耦合度，对于模块修改、扩增和删减不会影响到其它的模块，从底层开始建立良好的系统生态，避免因为单个模块运行错误而导致整个程序崩溃的现象发生。

## 3.4 本章小节

本章对基于DirectX 11的光栅化渲染器开发进行了需求分析，对需求进行分解，从功能需求、用户需求和性能需求三个部分并分别进行了详细的讨论。系统的设计和实现都会依据需求分析得出的结果而进行。

第4章 系统设计

本文的核心内容是基于DirectX 11的光栅化渲染器的设计与实现，本章阐述了DirectX 11的光栅化渲染器系统的设计思路。为实现基本渲染功能，须对DirectX 11渲染管线和相关的资源信息进行合理的封装；作为可交互程序须支持获取外部设备输入并合理解析使用的功能；创建用户界面来支持高级的信息显示和输入功能。

## 4.1 总体架构设计

4.1.1 基本逻辑

光栅化渲染器程序采用C++面向对象的模式实现。渲染器程序即渲染引擎对象，在程序运行期间以单体的形式存在，通过初始化、更新、渲染和输入控制等方法实现程序框架顶层的逻辑循环，此逻辑循环如图4.1所示：

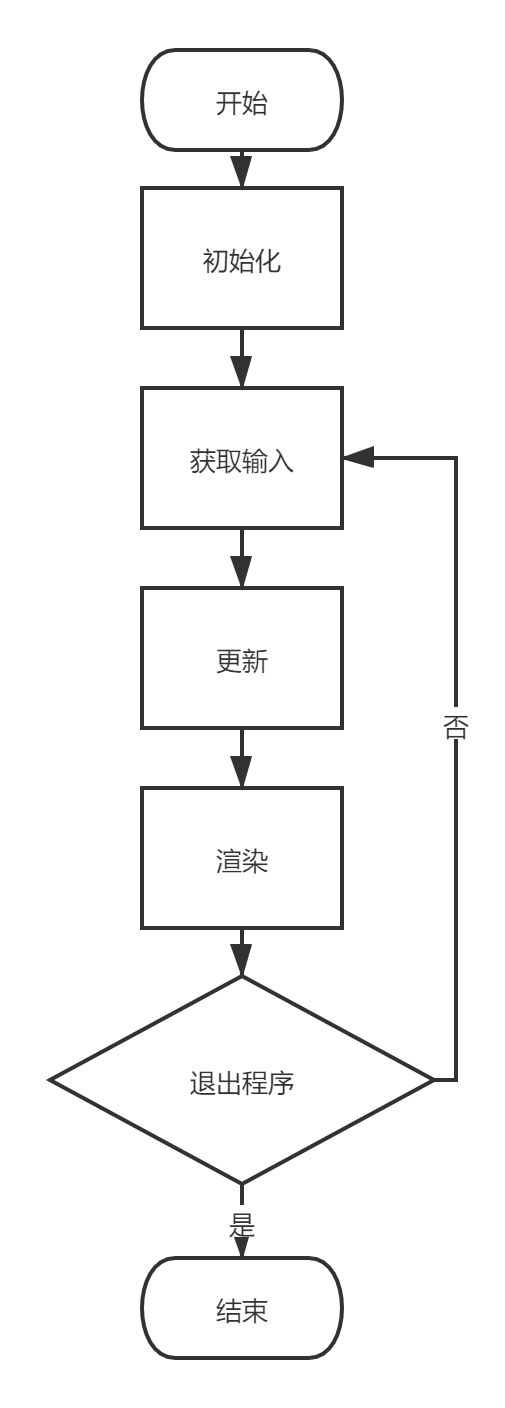


图4.1 系统顶层逻辑

程序在运行之后完成初始化，然后进入循环，在循环期间的每一帧当中，其接受用户在输入设备的输入，并根据输入对程序状态信息进行更新，在更新之后，使用新的状态信息对场景进行渲染，用户得到显示设备的反馈信息，并进行下一次输入。系统的主体框架包含这四部分基础逻辑，系统内的所有子系统基于这四部分基础逻辑来执行。

4.1.2 顶层模块概述

系统的顶层功能模块从属于渲染引擎对象，以渲染引擎对象成员的形式存在，这些模块在渲染引擎的基本逻辑循环当中执行各自的方法来实现自身功能。顶层功能模块和渲染引擎对象的从属关系如图4.2所示。

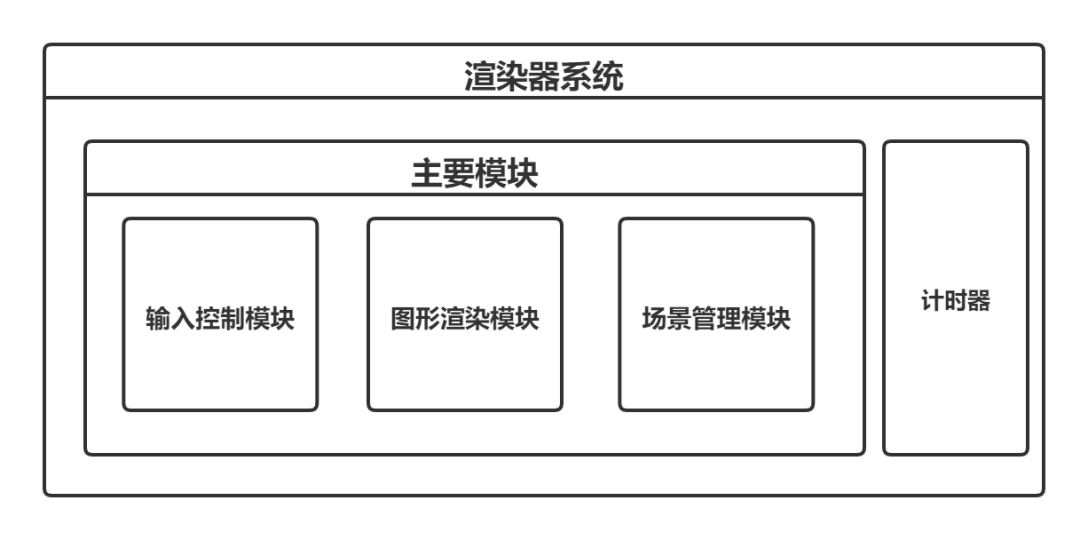


图4.2 顶层功能模块从属关系

系统的主要模块包括图形渲染模块、场景管理模块和输入模块，计时器并非系统的必须模块。程序开始执行之后，渲染引擎对象执行初始化操作逻辑，所有顶层模块也在此执行各自的初始化逻辑（如果需要）。随后渲染引擎对象进入主循环逻辑，输入模块不断地获取用户的外部输入，场景管理模块执行更新操作，图形渲染模块执行每一帧的渲染操作，计时器则根据单帧时间来统计所需的时间信息。直到输入模块获取到退出程序的指令，渲染引擎对象跳出主循环，结束程序。

各个模块的实现也采用面向对象的模式，但是并非所有模块都由单个对象组成，例如图形渲染模块，它是由图形渲染管理对象和渲染窗口对象组成的，由于二者的实现调用的是两种不同的应用程序接口，考虑到程序设计的原因，强行将这两个对象合并为一个会使对象内容变得臃肿，且杂揉了不相干的方法，造成耦合，不利于之后的开发。

## 4.2 实体-组件架构设计

4.2.1 实体-组件模式概述

实体是渲染器系统运行期间操作的基本单元，不具备任何功能，具体的功能是由组件来实现的。实体拥有一个组件列表，可以通过添加组件的形式为实体附加功能。

系统当中并非所有的操作单元都具有相同的功能，比如一些实体作为渲染的对象使用，一些实体作为相机使用，还有一些作为光源使用，它们的功能并不相同，但是它们也会具有相同的功能比如可以进行移动、旋转这样的变换操作，如果以传统的父子类继承的方式来实现，可能会导致代码的冗余或者父子继承关系的复杂化，导致架构混乱，且不利于之后更多功能的扩展。以实体-组件模型来设计的优点在于，将具体的功能和实体分离，需要怎样的功能就添加对应的组件，需要新的功能就创建新的组件，这样实体类本身永远不需要改动，而组件类则可以根据需求任意扩展。

实体-组件模式和著名的ECS（Entity-Component-System）即实体组件系统是不同的。ECS的实体（Entity）和组件（Component）不拥有任何方法，实体拥有一个ID，而组件只拥有相应组件类型的特定数据信息，所有的逻辑都由系统（System）来完成，系统负责处理哪些实体拥有哪些组件，组件的数据信息如何更新[9]。实体-组件模型中的实体则自己来管理组件，控制组件调用哪些逻辑，组件也同时拥有数据和方法，并提供接口便于外部调用。

4.2.2 实体

实体（Object）拥有一个组件列表，保存所有添加到实体上的组件的基类指针。实体按系统需求提供接口，例如初始化、更新、渲染以及销毁，所有实体拥有的功能都可以用这些接口来概括。实体提供组件操作方面的接口如添加组件，删除组件、获取组件以及判断实体是否拥有指定类型的组件。这些接口基本构成了实体操作的整个生态。

4.2.3 组件

组件拥有一个基类（Component），包含指向实体的指针，并拥有虚函数接口，同样为初始化、更新、渲染以及销毁，和实体相对应，提供给实体来调用。所有类型的组件均继承自组件基类，并根据自身定义重写实现虚函数接口，实体的组件列表存储组件基类的指针，并按需调用这些接口，即可实现组件定义的功能。除虚函数接口的重写之外，组件也可以自定义额外的接口，提供更加丰富的功能。

## 4.3 图形渲染模块设计

图形渲染模块是渲染引擎的核心，由渲染窗口对象和图形渲染管理对象组合实现。图形渲染模块的架构如图4.3所示。

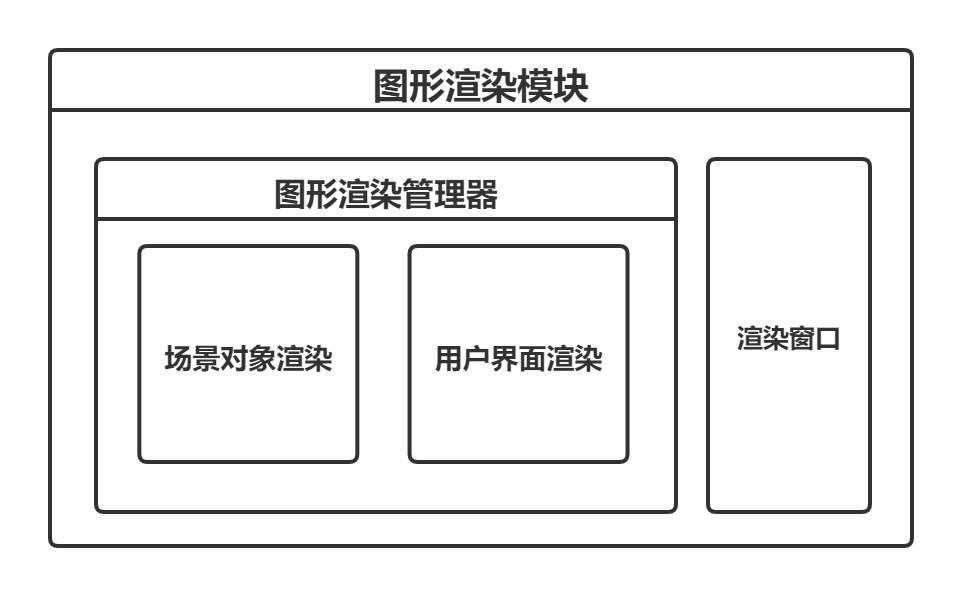


图4.3 图形渲染模块架构

4.3.1 渲染窗口

渲染窗口（RenderWindow）是图形渲染模块的基础和依托。渲染窗口的设计和实现决定于程序所在的具体平台，本系统基于Windows平台实现和运行，所以渲染窗口的设计依据Windows API进行。渲染窗口的主要功能是创建程序主窗口，获取返回的窗口句柄，作为图形渲染的输入目标。输入信息的接收也是由窗口完成的，在创建窗口之前，须注册窗口信息类，在此处设置窗口处理输入信息的具体方法。

4.3.2 图形渲染管理器

图形渲染管理器（Graphics）是全局图形渲染状态的管理者，负责在顶层处理3D图形的渲染和用户界面的渲染。

图形渲染管理器在图形渲染方面的主要功能是创建并维护全局通用的DirectX资源对象，并提供顶层的渲染逻辑。图形渲染全局公用的资源如图4.4所示。

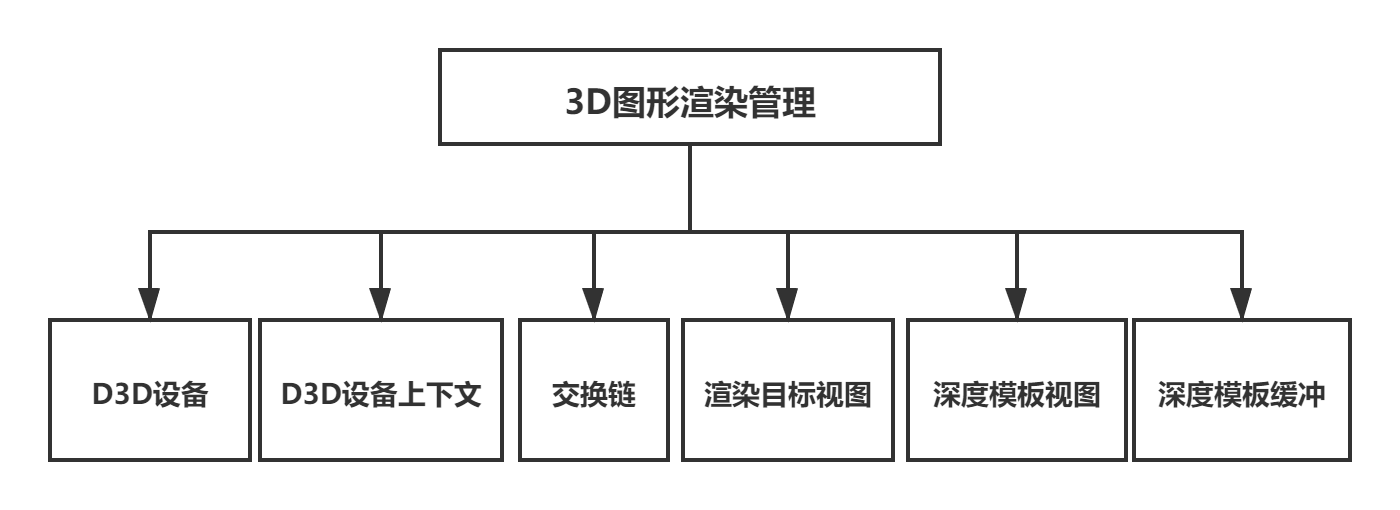


图4.4 图形渲染全局公用资源

实现DirectX 11的3D图形渲染，首先要对Direct3D进行初始化，在其初始化阶段需要创建Direct3D设备和Direct3D设备上下文。

Direct3D设备通常代表一个显示适配器，它最主要的功能是用于创建各种所需资源，此外，Direct3D设备还能够用于检测系统环境对功能的支持情况。Direct3D设备上下文可以看做是渲染管线。

一个Direct3D设备仅对应一个Direct3D立即设备上下文，并且只要我们拥有其中一方，就能通过各自的方法获取另一方。渲染管线主要负责进行资源的传递和渲染的计算工作，需要将Direct3D设备创建的各类资源、视图以及着色器等绑定到渲染管线，才能使其运作。除此之外，渲染管线还能够对资源进行直接的读写操作[10]。

除Direct3D设备和Direct3D设备上下文之外，还需要创建交换链。交换链是一系列的帧缓冲区，由显卡和图形接口用以实现稳定帧速率等功能。交换链至少有两个缓冲区，都可以称为后备缓冲区，后备缓冲区是系统进行渲染的场所，需通过合适的手段使其成为渲染管线的输出对象。

在创建了交换链之后，图形渲染管理器将交换链的后备缓冲设置为渲染目标，然后根据需求创建深度模板缓冲和深度缓冲视图，为之后的深度模板测试做准备。

视图是一种资源描述符，在渲染处理的过程中，GPU可能会对资源进行读、写两种操作。CPU发出绘制命令之前，需要将与本次绘制调用（draw call）相关的资源绑定到渲染管线上。一些资源可能在每次绘制调用时有所变化，所以我们也就要每次根据需求更新资源的绑定。但是一些GPU资源不是直接绑定至渲染管线，而是要通过一种名为描述符（即视图）的对象来对它进行间接引用，可以把资源描述符视为一种对送往GPU的资源进行描述的轻量级结构。指定了资源描述符，GPU不仅可以获得实际的资源数据，还能了解到资源的必要描述信息[11]。

最后图形渲染管理器进行视口设置。视口设置是设置渲染结果到窗口特定范围的映射，可以实现小窗口等功能。

除维护全局通用的DirectX资源对象之外，图形渲染管理器的另一个功能就是实现顶层的渲染逻辑。在每一帧当中，首先需要DirectX上下文来刷新渲染目标视图，即清空交换链的后备缓冲区，并且可以自定义刷新使用的颜色；然后刷新深度模板缓冲视图，即清空深度模板缓冲区，同样可以指定值来刷新；然后渲染场景当中的对象；最后通过交换链翻转后备缓冲区，将渲染的结果呈现在显示设备上。图形渲染模块顶层的渲染逻辑如图4.5所示。

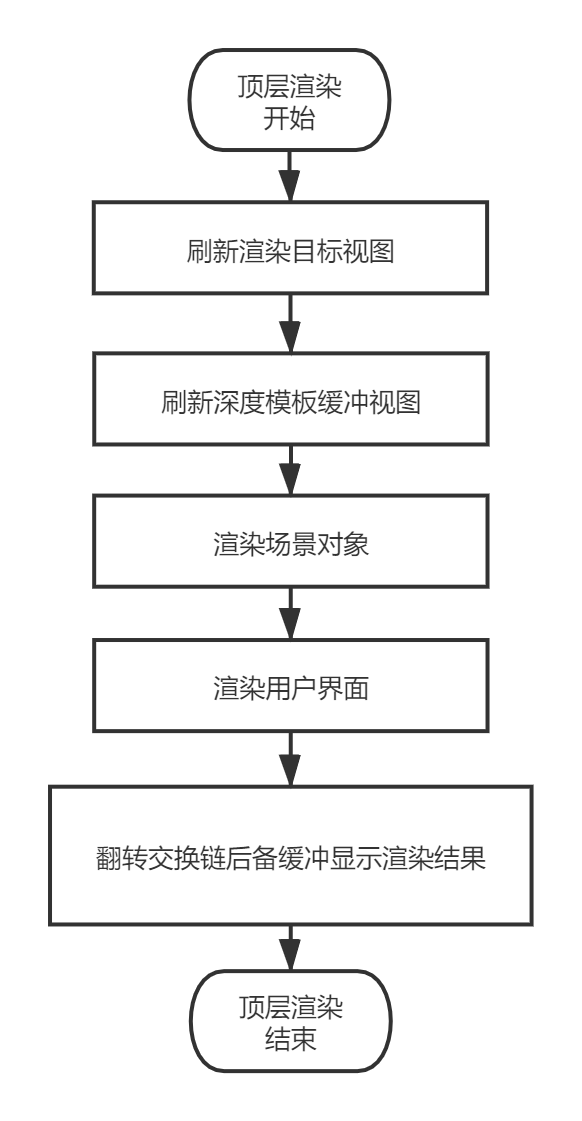


图4.5 顶层渲染逻辑

## 4.4 场景管理模块设计

场景管理模块由场景管理器（SceneManager）实现。场景管理器负责管理场景当中的所有实体，它拥有一个存储实体的列表，遍历并调用每一个实体的初始化方法和更新方法。另外，场景管理模块提供接口实现添加和删除实体的功能。

## 4.5 输入控制模块设计

输入控制模块由鼠标输入控制和键盘输入控制两个子模块构成。两个模块的实现方式类似，它们均自定义各自的输入事件枚举类型，并将事件抽象为一个对象，并各自维护一个消息队列以及输入相关的状态信息如按键状态、鼠标指针位置等，在程序窗口回调的函数当中将得到的消息类型抽象为自定义的事件对象，并推入到队列当中，并根据消息更新维护的输入状态。在使用时，可以根据需求获取输入状态或者从队列中推出事件，根据得到的结果决定要进行的操作。

## 4.6 资源结构设计

资源是Direct3D渲染管线可以访问的内存区块，包含了许多类型的数据如几何结构、纹理或者着色器数据，Direct3D用来它们渲染场景。这些资源结构需要创建并绑定到Direct3D渲染管线（上下文）中，DirectX 11 已经内置了这些资源的类和结构，但是其中大部分的资源使用都需要填充复杂的描述结构，并另外调用额外的绑定方法，在系统当中大量的使用容易造成代码的冗余和效率的下降。所以本系统对这些资源结构进行了二次封装，集创建、绑定、解绑、销毁于一体，便于使用且使代码逻辑更加清晰。

除DirectX 11内置的资源之外，本系统还定义了渲染常用的资源结构如网格（Mesh）、材质（Material）和着色器参数（ShaderParameter）等，这些结构的存在主要是为了将DirectX 11原有的内置资源结构结合并统一使用，使其更能够满足系统设计的渲染要求。

系统定义的资源结构之间的关系如图4.6所示。

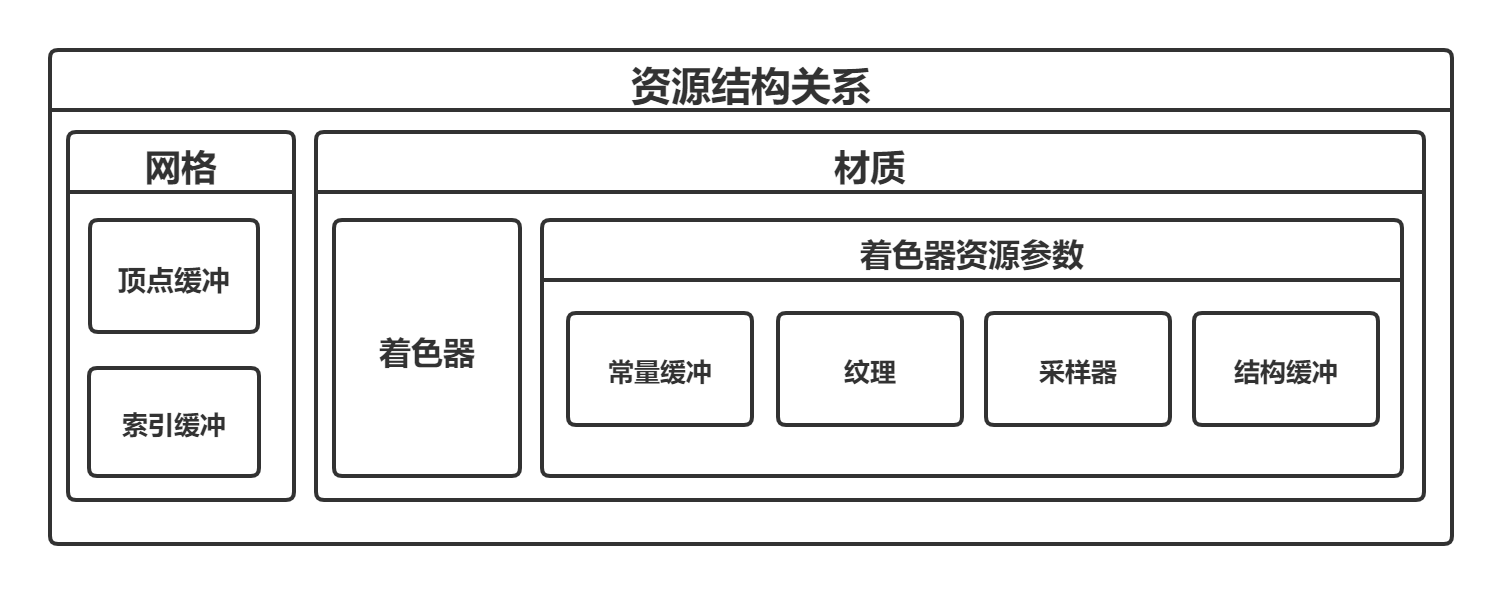


图4.6 资源结构关系

4.6.1 顶点缓冲和索引缓冲设计

顶点缓冲包含构成几何体的一系列顶点数组，数组的每个元素由顶点的具有的属性构成，如顶点位置坐标、纹理坐标、法线向量、切向量等等。缓冲区既非多维资源，也不支持mipmap、过滤器以及多重采样技术。

索引缓冲通常需要和顶点缓冲结合使用，它的作用就是以索引的形式来引用顶点缓冲区中的某个顶点，并按索引缓冲区中定义的顺序和图元类型来组装图元。使用相同的索引值就可以多次引用同一个顶点，它可以有效地减少顶点缓冲区中重复的顶点数据，从而减小网格数据占用的空间。

DirectX 11并没有专门为顶点缓冲和索引缓冲定义相关的类和结构，它们的本质都是缓冲区，只是缓冲区元素的具体数据类型不同。系统分别对顶点缓冲和索引缓冲进行了二次封装，在创建缓冲区时，根据需要的具体类型对描述结构进行填充。

4.6.2 网格设计

网格是系统自定义的资源结构类型。通常每个模型由几个子模型组合而成，组合模型的每个单独的形状即一个网格。一个网格是在渲染流水线当中绘制物体所需的最小单位。网格资源结构由系统定义的顶点缓冲和索引缓冲组成，从模型文件当中读取对应的信息对其进行构造。

4.6.3 常量缓冲设计

常量缓冲也是一种缓冲区资源，可以通过C++代码创建常量缓冲区并将其绑定到渲染管线对应着色器HLSL代码的cbuffer，从应用端传递数据到可编程着色器。

每个着色器阶段最多允许绑定15个常量缓冲，每个缓冲区最多可以容纳4096个标量。在C++创建常量缓冲区时其字节大小值必须为16的整数倍，HLSL的常量缓冲区本身以及对它的读写操作严格按16字节对齐。当一个着色器使用了多个常量缓冲时，它们的成员不能同名，一个常量缓冲也可以同时绑定到渲染管线不同的可编程着色器阶段，因为常量缓冲是只读的，不会导致内存访问冲突。

系统对常量缓冲进行封装，提供创建、绑定、解绑以及更新数据的接口。常量缓冲和顶点缓冲、索引缓冲不同的地方在于，顶点缓冲和索引缓冲的数据在一次设定之后，几乎没有机会改变，而常量缓冲作为一种类似参数的形式在C++端和着色器端传递，其值会经常发生改变，所以需要在创建时将其缓冲类型设置为可以更改，且允许CPU进行访问。

4.6.4 纹理设计

纹理资源是一种结构化的数据集合，用来存储像素。像素代表渲染管线可以读写的纹理资源的最小单元，每个像素包含1~4个分量。不像是缓冲区资源，纹理资源可以被纹理采样器过滤，纹理的类型会决定如何过滤纹理。

DirectX 11提供了三种纹理接口分别为1D纹理、2D纹理和3D纹理，在系统中主要使用的是2D纹理。2D纹理可以看作是存有纹理数据元素的二维数组，每个元素可以存储像素的颜色，也可以存储多维向量，例如在使用法线纹理映射时，每个纹理元素存储的实际上是一个3D向量而非颜色数据。纹理资源和缓冲区资源不同，缓冲区资源只能够存储资源数据，但是纹理可以拥有多个mipmap层级，GPU可以基于这些层级实现相应的特殊操作如使用过滤器或者多重采样。

系统对纹理资源进行了二次封装，除了包含2D纹理资源对象之外，还将着色器资源视图加入到资源结构中，因为纹理不能向缓冲区资源一样直接绑定到渲染管线上，需要借由着色器资源视图进行绑定。所以系统在创建纹理资源时，也要对着色器资源视图描述结构进行填充。

4.6.5 采样器状态设计

采样器资源用来定义在采集纹理资源时使用的过滤器和寻址模式。系统对采样器资源进行了封装，提供创建、绑定、解绑接口。在着色器HLSL代码中定义采样器对象，在C++代码端将创建的采样器资源绑定到渲染管线对应的着色器阶段进行使用。

4.6.6 结构缓冲设计

结构缓冲类似常量缓冲，都是以缓冲为基础的变体形式，作为一种类似参数的形式在C++端和着色器端传递数据，但又和常量缓冲有许多不同之处。结构缓冲可以理解为是缓冲区的复合形式，其中的结构可以使用自定义的类型，即结构缓冲区存储的内容可以解释为结构体数组。结构缓冲在着色器HLSL代码当中的定义是泛型的，需要在其中事先定义结构体，再以此结构体类型定义结构缓冲。另外，结构缓冲无法直接绑定至渲染管线，和纹理资源类似，它需要借由着色器资源视图进行绑定。

系统对结构缓冲资源进行了封装，包含对应结构体数组的泛型数据列表，缓冲资源对象和着色器资源对象，并提供创建、绑定、解绑接口。

结构缓冲在系统中的主要应用在于前向渲染技术当中的光源列表信息传递，将C++代码端定义的光源数据传递到着色器中并实现光照模型，这部分将在之后的渲染算法实现小节当中进行具体阐述。

4.6.7 着色器设计

着色器是渲染管线当中重要的组成部分，DirectX 11可编程着色器由HLSL语言编写，然后被编译成虚拟机字节码也被称为中间语言（Intermediate Language），中间语言再被驱动程序转换为指令集（ISA）在GPU上运行。

着色器在HLSL代码中的实现形式是一个函数，拥有输入和输出，输入即为函数的参数，参数来自于上一个渲染管线阶段的结果或者应用层的输入，输出即为函数的返回值，传递到渲染管线的下一个阶段。着色器分为多种不同的类型，并在渲染管线的不同阶段执行，如图4.7所示。

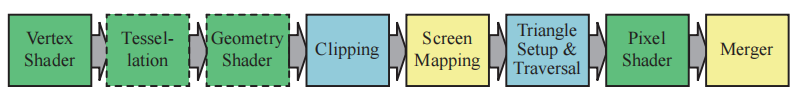


图4.7 各类着色器在渲染管线中的执行顺序

DirectX 11针对以上各个类型的着色器都提供了类的定义，本系统对其进行了进一步的封装，系统将所有类型的着色器指针整合在一个着色器类当中，在使用时，根据所需的具体着色器类型对其中的一种进行实例创建、绑定或者解绑操作。

4.6.8 着色器资源参数设计

着色器资源参数是系统自定义的资源结构类型。将常量缓冲资源、纹理资源和采样器资源封装到一个结构类型当中，并根据所需的具体资源类型选择性实例化。在实现用户界面的材质参数调节功能时，需要通过反射获取着色器定义的常量参数如常量缓冲、纹理和采样器等，定义着色器资源参数的目的是便于材质使用单类型的数据容器如列表或哈希表对以上的资源对象进行存储，进而可以通过遍历来显示所有的参数信息。

4.6.9 材质设计

材质是系统自定义的资源结构类型，材质代表的是对象表面的物理属性和渲染属性。材质拥有存储各类型着色器和着色器资源参数结构的容器，决定材质物理属性的是其拥有的着色器中定义的资源参数。在材质实例化时，对着色器进行反射操作得到其定义的资源参数信息并填充到容器当中，然后通过遍历容器将资源参数显示在用户界面，以此实现材质参数的调节功能。材质还包含渲染管线部分阶段的状态对象如深度模板状态、混合状态和光栅化状态，通过将这些状态结构的属性值暴露给用户界面，即可针对渲染属性进行调节，实现半透明渲染效果等。

## 4.7 实体渲染系统设计

在图形渲染模块控制的顶层渲染逻辑当中，完成后备缓冲区和深度模板缓冲的清空之后，即是对于场景对象和用户界面的渲染，其中场景对象的渲染为实体的渲染，实体的渲染功能由其具有的组件协作实现。

实体渲染所需的核心组件为变换组件（Transform）、网格渲染组件（MeshRenderer）和材质管理组件（MaterialManager），相关的组件包括相机组件（Camera）和光源组件（Light）。变换组件决定实体渲染结果的位置、旋转和缩放，网格渲染组件决定实体渲染使用的模型网格并提供渲染逻辑，材质管理组件决定渲染使用的材质及着色器，相机组件和光源组件则在着色器的渲染算法实现当中提供额外的数据信息。

4.7.1 实体初始化阶段

在实体的初始化阶段，为组件分别添加变换组件、网格渲染组件和材质管理组件之后，组件分别执行各自的初始化方法。

在网格渲染组件的初始化方法当中，它使用给定的文件路径对模型网格资源进行加载，获取模型的顶点及索引数据，并将其抽象为系统定义的网格结构（Mesh），加入到自身维护的容器当中。

在材质管理组件的初始化方法当中，它根据实体的网格渲染组件拥有的网格数量实例化对应数目的材质结构（如果实体不具有网格渲染组件，则无法添加材质管理组件），并添加到自身维护的材质结构容器中，每个材质的实例化需要一个着色器资源文件，系统定义默认的初始化使用的着色器为PBR着色器，保证创建一个对象时其渲染是有效的，在完成创建之后，用户可以通过用户界面对其进行更改。

材质在实例化过程中需要完成对成员的构造，包括着色器实例化、输入布局的创建、着色器资源参数容器的填充、深度模板缓冲状态对象、混合状态对象和光栅化状态对象的创建。这些成员在初始化阶段完成构造，在渲染阶段被使用。

在上述的组件完成初始化之后，实体即拥有了实现渲染功能的基本要素——变换、网格、材质和着色器，之后进行的操作即为每一帧的渲染。

4.7.2 实体渲染阶段

实体的渲染由网格渲染组件完成，其根本逻辑就是将渲染管线所需的资源进行绑定，因为DirectX 11的渲染管线是基于状态的，即渲染每一帧的每一个实体前都需要对资源进行更新。如果不进行资源的更新，则之后使用的都将是之前使用的资源，造成错误的渲染结果。

网格渲染组件遍历列表存储的所有网格，将网格结构包含的顶点缓冲和索引缓冲绑定至渲染管线。并同时访问实体的材质管理组件，获取其维护的材质列表当中对应位置的材质（因为材质列表的填充是根据网格组件的网格列表进行的，所以二者的列表元素数目一定相同），将其拥有的渲染管线资源成员如输入布局、深度模板缓冲状态、混合状态、光栅化状态以及着色器资源列表包含的所有资源结构绑定至渲染管线，完成所有资源的绑定之后，调用Direct3D的绘制方法，完成最终的渲染。

相机组件和光源组件提供的数据以着色器资源参数的形式传递到着色器当中，这些数据是完成一个完整3D图形渲染和光照模型的必要信息。相机组件为顶点着色器提供空间变换所需的矩阵，实现3D渲染效果。光源组件则提供场景的光源列表数据，支撑着色器实现光照计算。

## 4.8 用户界面模块设计

用户界面本应该作为图形渲染模块的一个主要组成部分存在，但是因为它的实现较为复杂，不仅涉及到渲染而且也涉及到很多处理逻辑，所以这里另开一个模块对其进行介绍。

用户界面的实现引用了Dear ImGui库，其运行逻辑和DirectX的图形渲染的逻辑并不相同，却又依附于DirectX 11运行，因其渲染所需要的准备工作和DirectX 3D图形渲染相同，如果强行将其从图形渲染管理器中剥离，那么就需要图形渲染管理对象实现外部调用的方法来供其使用，这样反而使代码逻辑变得混乱。所以，为了代码架构的健康性考虑，将其纳入到了图形渲染管理器的成员当中，和3D图形一同进行渲染。

用户界面的实现采用面向对象的模式，用户界面对象（UserInterface）作为图形渲染管理器的成员，负责管理所有的界面元素，系统的界面元素主要由三部分组成：场景对象层次结构面板（Hierarchy）、细节信息面板（Inspector）和窗口工具栏（TopToolBar），其组成关系如图4.8所示。

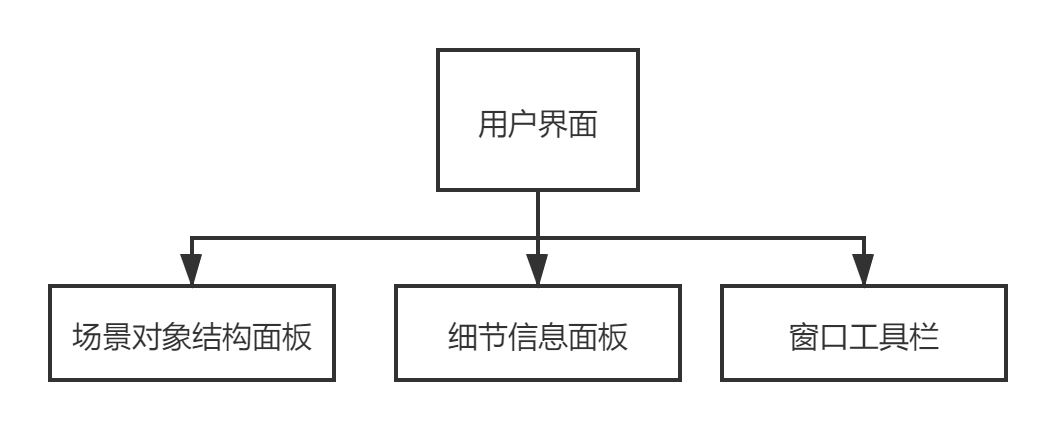


图4.8 用户界面组成

场景对象层次结构面板（Hierarchy）：以列表的形式显示当前场景当中能够被用户选定并进行操作的对象。场景对象层次结构面板会遍历存储场景对象的列表，并显示每个对象的名字，Dear ImGui提供了渲染可选择对象的接口，场景对象层次结构面板可以利用此接口方便的渲染所有可选择对象对应的界面元素。

细节信息面板（Inspector）：显示当前选定对象的具体信息。由于场景对象以实体-组件的形式实现，所以当选定一个场景对象，细节信息面板会获取选定的具体对象的指针，进而获取其拥有的所有组件，并逐个展示其具体信息。当用户在细节信息面板修改了组件的参数值后，Inspector也会将修改后的值应用于对应的组件，实现交互调参的功能。

窗口工具栏（TopToolBar）：提供全局的操作接口。窗口工具栏提供了如场景文件

加载和保存、对象的创建、渲染器外观设置和其他一些个性化的操作功能。

## 4.9 本章小节

本章根据需求分析的结果对系统进行了由上至下、由浅到深的设计。从系统的总体架构开始，将其设计为由三个核心模块组成，包括图形渲染模块、场景管理模块和输入控制模块。然后针对每个模块进行了更细致的设计，确定每个模块具体应该由哪些类型实例进行实现。除此之外，对于实现渲染所必须的资源结构进行设计，明确各个资源结构的含义和功能。最后，对针对整个实体渲染系统流程进行分析设计。

第5章 系统实现

在完成对基于DirectX 11的光栅化渲染器进行的需求分析和系统设计之后，开始对系统进行具体的实现。系统实现参照系统设计提出的设计思路对核心模块的功能实现进行具体阐述，以关键代码、程序流程图和效果展示图等方式进行叙述的补充。

本系统实现使用的核心编程语言是C++，利用其面向对象的特性，丰富的标准库资源，以及完美对接DirectX 11和Dear ImGui的应用程序接口的优势，能够高效、快速地实现系统需求。着色器部分则使用DirectX 11的标准图形语言HLSL进行图形编程，实现具体的渲染算法。

## 5.1 总体架构实现

5.1.1 程序入口

基于DirectX 11的光栅化渲染器程序执行的起点是Windows平台的提供的图形化应用程序入口WinMain函数[12]，函数签名如下。

int WINAPI WinMain(

HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nShowCmd

);

hInstance表示应用程序当前的句柄；hPrevInstance表示应用程序先前实例的句柄，对于同一个程序打开两次，出现两个窗口的第一次打开的窗口就是先前的实例。对于32位程序，该参数为NULL。lpCmdLine为指向应用程序命令行的字符串的指针；nShowCmd 为指明窗口如何显示的标志，指示主应用程序窗口是否将被最小化，最大化或正常显示。

系统需要的主要是第一个参数——应用程序句柄，因为渲染窗口的构造需要其作为参数。另外三个参数没有使用，可以使用C++ 的UNREFERENCED\_PARAMETER宏函数将其标记，防止编译器报错。

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

UNREFERENCED\_PARAMETER(nShowCmd);

系统将基于DirectX 11的光栅化渲染器程序抽象为一个对象，并将其类型定义为RenderingEngine，因为程序运行期间该类型只有一个实例存在，所以对其实现单例模式，私有化构造函数，并提供获取单例的接口，以获取全局唯一实例，在WinMain入口函数中调用渲染引擎实例的初始化和运行方法，核心代码如下。

try

{

//渲染引擎初始化

RenderingEngine::GetInstance()->Initialize(hInstance,"StemCell Engine","WndClassName", 1800, 900);

}

catch (COMException& exception)

{

ErrorLogger::Log(exception);

return -1;

}

//渲染引擎运行

return RenderingEngine::GetInstance()->Run();

5.1.2 初始化

RenderingEngine在其初始化函数Initialize当中会对拥有的成员进行初始化，如系统设计部分所提到的，RenderEngine应包含渲染窗口对象、图形渲染管理对象和场景管理对象，这些成员在此处分别完成各自的初始化操作。

5.1.3 运行

RenderingEngine的运行函数Run实现的则是程序循环，其程序流程图如图5.1所示，可以理解为系统设计阶段对应的运行函数部分的细化。

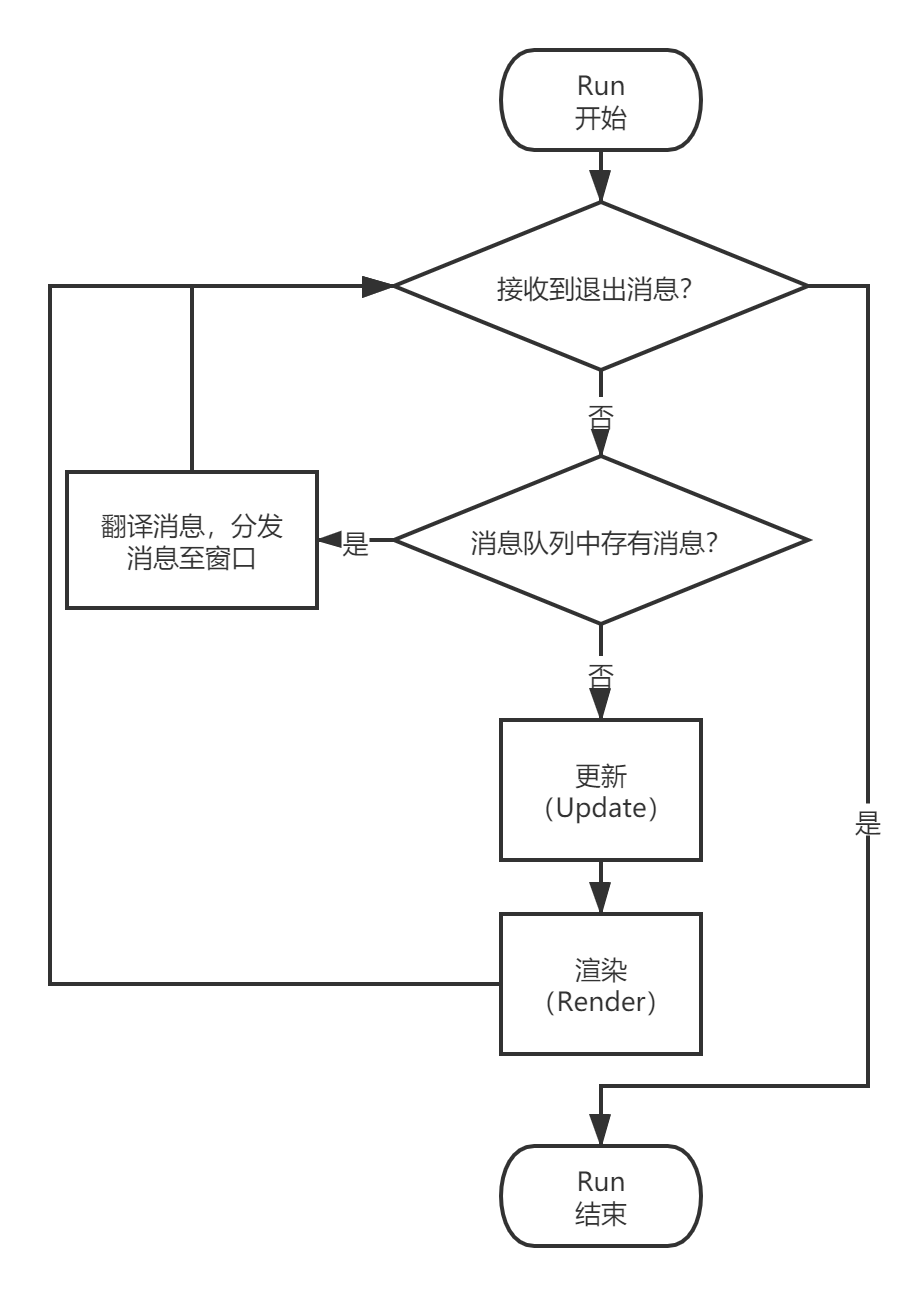


图5.1 Run函数程序流程

Windows是由事件驱动的系统，事件驱动围绕着消息的产生和处理展开，是靠消息的循环机制来实现的。

消息（Message）指的是Windows操作系统发送给程序的一个通告，系统在发生输入事件(用户敲击键盘、移动鼠标)时产生消息。应用程序可以产生消息使窗体执行任务，或者与基他应用程序通信，消息提供了应用程序与应用程序之间、应用程序与Windows 系统之间进行通讯的手段。每个GUI线程维护一个线程消息队列，线程消息队列中的消息会被送到相应的窗口过程处理。Run函数程序流程的第一步即是对消息队列检查，判断队列当中是否存有消息，此处调用PeekMessage函数，其函数签名如下。

BOOL PeekMessage(

LPMSG IpMsg,

HWND hWnd,

UINT wMSGfilterMin,

UINT wMsgFilterMax,

UINT wRemoveMsg

);

函数将队列中的消息存储至第一个参数IpMsg中，这里将第二、三、四个参数均设置为NULL或0，表示需要函数传回所有视窗的所有消息，并将最后一个参数设置为PM\_REMOVE，表示获取消息后将其从队列中删除。如果消息队列中有消息，则函数返回true，否则返回false。

当获取了消息之后，需要对消息进行进一步的处理以满足程序需求。此处先后调用TranslateMessage和DispatchMessage函数，TranslateMessage函数将虚拟键消息转换成字符消息。该字符消息又被发送给对应线程（调用TranslateMessage的线程）的消息队列，当线程再次调用GetMessage函数或PeekMessage函数获取消息的时候被读取。DispatchMessage函数将把处理后的消息发送给窗口中已设定的回调函数，在回调函数设置消息分支判断处理逻辑，可实现更多自定义的程序功能。

如果消息队列中没有消息需要处理，则进入到了真正的运行逻辑中，先后执行更新函数（Update）和渲染函数（Render）。和更新函数相关的成员是场景管理对象，在其中调用自身的更新方法（Update），和渲染函数相关的是图形渲染管理对象，在其中调用自身的渲染方法（Render）。

## 5.2 实体-组件架构实现

5.2.1 实体实现

系统定义的实体类型为Object，其基础私有成员包含全局唯一标识（m\_UID），激活标识（m\_active）以及自己的管理者（m\_ownerManager），除全局唯一标识不可修改之外，另外两个成员均提供修改接口。Object拥有的和实体-组件模式相关的高级私有成员则是组件列表（m\_components）和组件哈希表（m\_componentsMap），二者的声明如下。

std::vector<Component\*> m\_components;

std::map<const std::type\_info\*, Component\*> m\_componentsMap;

m\_components的实现使用了C++ 标准库的vector容器，元素类型为组件基类的指针，主要负责存储实体拥有的所有组件，并提供遍历功能。m\_componentsMap的实现使用了标准库的map容器，元素的左键类型为type\_info的指针，右键同样为组件基类的指针（Component\*），主要为实体提供泛型获取组件的功能。

Object拥有的基础外部调用接口为更新（Update）、渲染（Render）以及销毁（Destroy）。Update由实体的对应的场景管理对象进行调用，Render由对应的图形渲染管理对象进行调用。两个函数均遍历m\_components组件列表，Update调用每个组件的更新方法，Render函数调用每个组件的渲染方法。由于逻辑简单，这里不进行详细解释。

实体Object拥有的和实体-组件模式相关的高级外部调用接口分别为添加组件（AddComponent）、获取组件（GetComponent）、组件存在判断（HasComponent）以及删除组件（RemoveComponent），四个函数都基于C++泛型特性[13]进行实现。

获取组件（GetComponent）和组件存在判断（HasComponent）函数的实现逻辑比较简单，都是使用组件的类型作为键并基于哈希表返回结果，相关代码实现如下。

组件存在判断（HasComponent）函数：

template <typename T>

bool HasComponent()

{

//如果参数类型的键对应的键值对数量为1，说明组件存在

return m\_componentsMap.count(&typeid(T)) == 1;

}

获取组件（GetComponent）函数：

template <typename T>

T\* GetComponent()

{

//如果实体并没有此类型组件，则返回空指针

if (!HasComponent<T>())

{

return nullptr;

}

//从map当中返回参数类型键对应的值

return static\_cast<T\*>(m\_componentsMap[&typeid(T)]);

}

添加组件（AddComponent）函数的实现逻辑则相对复杂，其具体代码实现如下。

template <typename T, typename...TArgs>

bool AddComponent(TArgs&&...args)

{

//如果对象已经拥有该类型组件，则函数返回

if (this->HasComponent<T>())

{

return false;

}

//实例化组件

T\* newComponent(new T(std::forward<TArgs>(args)...));

//设定组件对应的实体对象

newComponent->owner = this;

//组件初始化

newComponent->Initialize();

//将组件加入到组件列表

m\_components.emplace\_back(newComponent);

//将组件加入到组件map

m\_componentsMap[&typeid(\*newComponent)] = newComponent;

return true;

}

AddComponent函数的实现使用了C++模板和可变参数模板，T类型对应添加的组件类型，TArgs则是一个模板参数包，它是零个或多个类型参数的集合，可变模版参数（Variadic Templates）是C++11的强大的特性之一，它对参数进行了高度泛化，它能表示零到任意个数、任意类型的参数[14]。

函数首先根据T类型判断实体是否已经拥有对应的组件，如果没有，才能进行接下来的操作，否则直接退出，并返回false，表示未能成功添加组件。

在添加组件之前，首先要调用组件的构造方法对其进行实例化，因为在不确定组件具体类型的情况下，无法确定其构造函数所需参数的类型甚至数目，使用TArgs可变参数模板可以应对这种情况。

完成组件的实例化之后，将组件对应的实体设置为调用者（即组件添加的目标实体），然后调用组件的初始化方法（Initialize）。这里其实无法判定T类型有对应的Initialize方法，所以如果添加组件的类型并非继承自组件基类（Component）或者本身不具有Initialize方法，则会出现错误。

在完成对组件的构造和初始化之后，将组件对象指针添加到组件vector当中，将组件类型-组件对象指针键值对添加到组件map当中，为之后的使用做准备。

删除组件（RemoveComponent）函数的代码实现如下。

template <typename T>

bool RemoveComponent()

{

//如果对象没有拥有该类型组件，则返回

if (!this->HasComponent<T>())

{

return false;

}

unsigned int targetIndex = 0;

//遍历组件列表

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

{

//如果当前组件的类型名和参数类型名相同，则记录其位置

if (typeid(T) == typeid(\*m\_components[i]))

{

targetIndex = i;

break;

}

}

//从列表中删除组件

m\_components.erase(m\_components.begin()+ targetIndex);

//从哈希表中删除组件

m\_componentsMap.erase(&typeid(T));

return true;

}

函数的开始和添加组件类似，首先判断实体是否已经拥有参数类型对应的组件，如果没有则直接返回false，表示并没有删除组件，如果有，则进行之后的逻辑。

首先删除vector当中的组件，遍历组件列表，获取到和参数类型相同的组件对象指针的索引值，然后结束遍历，根据索引值得到的迭代器删除对应的组件，因为不能在遍历vector的过程中删除元素，会引发错误。然后删除map当中的组件类型-组件对象指针键值对，因为已经确定了map中含有参数类型的键，所以直接调用erase方法传入类型参数即可删除元素。

5.2.2 组件实现

1. 基类Component实现

系统定义的组件基类类型为Component，所有的功能组件均继承自该类。Component只拥有一个公有成员变量——实体（owner），为组件附着的实体对象指针。

Component定义了基础的虚函数，包括初始化（Initialize）、更新（Update）、渲染（Render）和销毁（Destroy）。以上虚函数和实体拥有的基础公有方法相对应，在实体当中调用。继承自Component的组件子类重写以上虚函数，根据自身具体功能自定义如何实现，以下介绍的系统内置组件均继承自Component类。

1. 属性组件（Attributes）实现

属性组件负责维护实体拥有的基本信息，系统中所有的实体均需要且默认添加属性组件。属性组件拥有的成员变量如表5.1所示。

表5.1 Attributes成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_objectName | 黑名单 |
| 2 | m\_tag | 用户间关系 |
| 3 | m\_displayInHierarchy | 是否显示在场景层次结构界面中 |

属性组件提供针对以上成员的获取、更改接口，属性组件虽然继承自Component基类，但是没有对其虚函数重写的需求。

1. 变换组件（Transform）实现

变换组件负责维护实体在场景世界中的空间位置信息，系统中所有的实体均需要且默认添加变换组件。变换组件拥有的成员如表5.2所示：

表5.2 Transform成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_position | 世界空间坐标 |
| 2 | m\_rotation | 世界空间旋转 |
| 3 | m\_scale | 世界空间缩放 |

变换组件提供针对以上成员的获取、更改接口，变换组件虽然继承自Component基类，但是没有对其虚函数重写的需求。

变换组件是协助渲染系统实现的关键组件，它提供了各类便捷的空间变换高级接口如旋转、平移和获取变换矩阵等。在渲染管线的实现当中，顶点着色器需要对顶点坐标进行各类空间变换，变换组件可以为其提供相关的矩阵参数如模型空间到世界空间、世界空间到视空间等，这些矩阵都是从渲染的实体对象和相机对象拥有的变换组件中获取的。

1. 材质管理组件（MaterialManager）实现

材质管理组件负责维护实体渲染所需的材质列表，其核心成员变量是由材质结构填充的vector列表，代码如下。

std::vector<Material> materials;

一个实体之所以对应的是一个材质列表而不是单个材质，是因为一个实体对应一个模型文件，每个模型文件包含一个或多个网格，每个网格材质渲染的最小单位，对应一个材质，所以这里使用材质的列表。材质管理组件重写了基类Component的初始化方法（Initialize），在初始化过程中，获取网格渲染组件（MeshRenderer）维护的网格数量，并实例化相同数量的材质，加入到列表当中，为之后的渲染做准备。

1. 网格渲染组件（MeshRenderer）实现

网格渲染组件负责加载模型文件，获取网格，以及实现渲染方法。网格渲染组件的核心成员变量为m\_meshes（网格结构列表）以及m\_meshFilePath（模型文件路径）。

网格渲染组件重写基类Component的初始化方法（Initialize）和渲染方法（Render）。

在初始化方法当中，网格渲染组件根据模型文件路径加载模型、读取数据，并将数据抽象为系统定义的网格结构（Mesh）添加到列表中。网格渲染组件的Render方法是实体渲染的系统核心操作，其程序流程如图5.2所示。

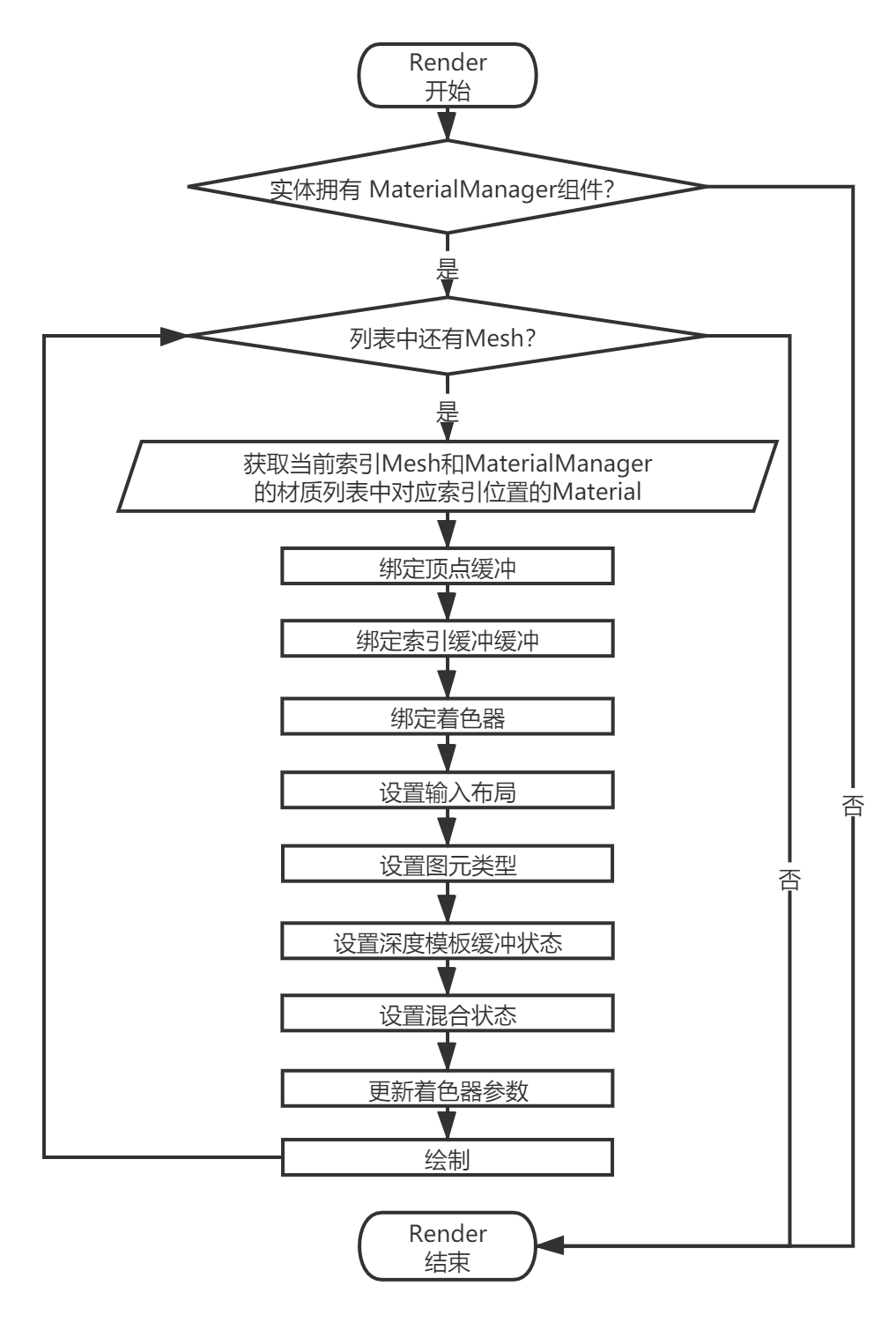


图5.2 Render函数程序流程

1. 相机组件（Camera）实现

相机组件使实体成为一个虚拟相机，虚拟相机的主要作用是在渲染场景时模拟人眼，提供位置、视线方向以及空间变换参数。相机组件拥有的成员变量如表5.3所示。

表5.3 Camera成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_fovY | 相机垂直方向视角 |
| 2 | m\_aspect | 相机投影平面横纵比 |
| 3 | m\_nearClip | 相机近裁剪面距离 |
| 4 | m\_farClip | 相机远裁剪面距离 |

相机组件为以上成员变量提供了获取和更改接口，另外定义了方法获取世界空间到相机空间的矩阵、投影矩阵以及相机位置。这些方法在传递参数至顶点着色器进行坐标变换时会使用。

获取投影矩阵函数为GetProjectionMatrix，直接调用DirectX数学库提供的函数接口即可，函数签名如下。

XMMATRIX XM XMMatrixPerspectiveFovLH(

float FovAngleY,

float AspectRatio,

float NearZ,

float FarZ

);

函数的四个参数即输入相机组件对应的四个成员变量，它能够构建一个基于视场的左手坐标系投影矩阵（DirectX使用左手坐标系）。

1. 光源组件（Light）实现

光源组件使实体成为一个场景中的一个光源，光源的主要作用是在实现渲染的光照模型时提供相关数据信息。光源组件拥有的成员变量如表5.4所示。

表5.4 Light成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_type | 光源类型 |
| 2 | m\_color | 光照颜色 |
| 3 | m\_intensity | 光照强度 |
| 4 | m\_range | 光照范围 |

在一般渲染引擎当中，常用的实时光源有三种类型，分别为直射光、点光源以及聚光灯，m\_type为自定义枚举类型，表示如下三种光源的一种。

enum class LightType

{

Directional = 0,

Spot = 1,

Point = 2

};

Directional表示直射光，Spot表示聚光灯，Point表示点光源，它们在渲染当中影响物体表面的方式是不同的，具体的实现将在之后的光照系统实现小节中进行介绍。

光源组件同样为各个成员变量定义了获取、更改接口，并且另外提供了和Transform组件相关联的快捷接口，如获取相机位置、朝向等。

## 5.3 图形渲染模块实现

5.3.1 渲染窗口实现

系统定义的渲染窗口类型为RenderWindow。其成员变量如表5.5所示。

表5.5 RenderWindow成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_hAppInstance | 应用程序句柄 |
| 2 | m\_hMainWindow | 窗口句柄 |
| 3 | m\_mainWndCaption | 窗口标题字符串 |
| 4 | m\_mainWndClassName | WNDCLASS注册类名 |
| 5 | m\_clientWidth | 窗口宽度 |
| 6 | m\_clientHeight | 窗口高度 |
| 7 | m\_minimized | 窗口是否最小化 |
| 8 | m\_maximized | 窗口是否最大化 |
| 9 | m\_resizing | 窗口是否正在缩放 |

渲染窗口的核心功能是为渲染管线提供输出窗口。RenderWindow定义了初始化函数（Initialize），在函数中实现了Windows窗口的创建，并获取其句柄，为之后的渲染做准备。

在Windows平台创建窗口，首先需要注册一个WNDCLASS类。WNDCLASS是一个由Windows系统支持的结构，用来储存某一类窗口的信息，通过对其成员进行自定义，可以创建出个性化窗口类。以下代码为本系统创建的WNDCLASS类。

WNDCLASS wc = {};

wc.style = CS\_HREDRAW | CS\_VREDRAW;

//设置窗口的应用程序实例句柄

wc.hInstance = m\_hAppInstance;

//设置窗口的消息回调函数

wc.lpfnWndProc = MainWndProc;

//设置注册类的名称

wc.lpszClassName = m\_mainWndClassName.c\_str();

wc.hIcon = LoadIcon(NULL, IDI\_QUESTION);

wc.cbClsExtra = NULL;

wc.cbWndExtra = NULL;

wc.hbrBackground = (HBRUSH)GetStockObject(NULL\_BRUSH);

wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

wc.lpszMenuName = NULL;

比较关键的设置是应用程序句柄和窗口回调函数。这里的应用程序句柄即RenderingEngine类由WinMain函数传入的实参获取而来，在RenderingEngine初始化阶段传递给RenderWindow对象。窗口回调函数则是总体架构实现小节当中提到的，由RenderingEngine调用DispatchMessage函数分发至窗口后应该执行的方法。这里使用的函数为MainWndProc，该函数并没有定义在RenderWindow源文件当中，所以使用extern关键自声明，表示其定义在项目其他文件当中（声明代码须放置在注册操作之前）。

extern LRESULT CALLBACK MainWndProc(

HWND hwnd,

UINT msg,

WPARAM wParam,

LPARAM lParam

);

由于系统将消息模块解释为是属于整体架构的一部分，所以消息处理的回调函数定义在RenderingEngine的源文件当中，具体的实现将在输入控制模块实现部分进行阐述。

完成WNDCLASS类的注册之后，使用RegisterClass方法对其进行注册然后就可以进行窗口实例的创建了。调用CreateWindow方法，创建一个WNDCLASS的实例窗口，以下为CreateWindow的函数签名。

HWND WINAPI CreateWindow(

LPCTSTR lpClassName, //窗口类名称

LPCTSTR lpWindowName, //窗口标题

DWORD dwStyle, //窗口风格

int x, //窗口x坐标

int y, //窗口y坐标

int nWidth, //窗口宽度

int nHeight, //窗口高度

HWND hWndParent, //父窗口句柄

HMENU hMenu, //窗口菜单句柄

HINSTANCE hInstance, //程序实例句柄

LPVOID lpParam //创建参数

);

这里将RenderWindow各成员对应传入即可，窗口的x、y坐标设置为屏幕中心位置，hWndParent、hMenu和lpParam传入NULL。函数返回HWND类型的窗口句柄，把它赋值给成员m\_hMainWindow，m\_hMainWindow将会传递给图形渲染管理对象，作为DirectX 11渲染管线的输入目标。

5.3.2 图形渲染管理器实现

系统定义的图形渲染管理器的类型为Graphics。Graphics的核心功能之一是维护全局的DirectX资源对象以及渲染相关的公用成员。Graphics拥有的DirectX资源对象成员变量如表5.6所示。

表5.6 Graphics 资源成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_dxDevice | Direct3D设备 |
| 2 | m\_dxSwapChain | Direct3D 交换链 |
| 3 | m\_dxDeviceContext | Direct3D 上下文 |
| 4 | m\_dxRenderTargetView | 渲染目标视图 |
| 5 | m\_dxDepthStencilView | 深度模板视图 |
| 6 | m\_dxDepthStencilBuffer | 深度模板缓冲 |
| 7 | m\_renderTargetBackgroundColor | 后备缓冲刷新颜色值 |

1. Graphics初始化

Graphics定义初始化方法（Initialize），在方法当中对上述的DirectX资源变量进行实例化，并设置到渲染管线当中。

首先调用D3D11CreateDeviceAndSwapChain函数创建D3D设备、交换链和上下文[15]。其函数签名如下。

HRESULT D3D11CreateDeviceAndSwapChain(

IDXGIAdapter \*pAdapter,

D3D\_DRIVER\_TYPE DriverType,

HMODULE Software,

UINT Flags,

const D3D\_FEATURE\_LEVEL \*pFeatureLevels,

UINT FeatureLevels,

UINT SDKVersion,

const DXGI\_SWAP\_CHAIN\_DESC \*pSwapChainDesc,

IDXGISwapChain \*\*ppSwapChain,

ID3D11Device \*\*ppDevice,

D3D\_FEATURE\_LEVEL \*pFeatureLevel,

ID3D11DeviceContext \*\*ppImmediateContext

);

函数调用的关键参数有\*\*ppSwapChain、\*\*ppDevice和\*\*ppImmediateContext，分别传入Graphics对应三个成员调用GetAddressOf函数获取的指针。另外\*pSwapChainDesc代表的是交换链信息描述结构体指针，因结构体需要填充的成员过多，这里只列举其中几个关键成员的设置，其余省略。

DXGI\_SWAP\_CHAIN\_DESC scd = {};

ZeroMemory(&scd, sizeof(DXGI\_SWAP\_CHAIN\_DESC));

scd.BufferCount = 1;//设置为1 则拥有两个缓冲

scd.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

scd.OutputWindow = hwnd;

scd.BufferDesc.Format = DXGI\_FORMAT\_B8G8R8A8\_UNORM;

scd.BufferDesc.Width = width;

scd.BufferDesc.Height = height;

……

将BufferCount成员设置为1，表示提供两个后备缓冲；BufferUsage 设置为DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT，表示将交换链后备缓冲设置为渲染输出目标；OutputWindow设置的值hwnd即为从渲染窗口（RenderWindow）创建窗口后得到的窗口句柄值；设置BufferDesc的Format为DXGI\_FORMAT\_B8G8R8A8\_UNORM，即缓冲的每个元素为8位BGRA值；BufferDesc的Width和Height则设置缓冲的宽高值。

接下来是渲染目标视图的创建，渲染目标对应的即为交换链的后备缓冲区，所以首先需要获取交换链的后备缓冲，这里通过交换链调用其GetBuffer方法，获取后备缓冲，然后通过D3D设备创建渲染目标视图，具体代码实现如下。

Microsoft::WRL::ComPtr<ID3D11Texture2D> backBuffer;

//从交换链获取后备缓冲

hr = m\_dxSwapChain->GetBuffer(0, \_\_uuidof(ID3D11Texture2D), &backBuffer);

//创建渲染目标视图

hr = m\_dxDevice->CreateRenderTargetView(

backBuffer.Get(),nullptr, m\_dxRenderTargetView.GetAddressOf());

然后是深度模板缓冲和视图的创建。深度模板缓冲是DirectX 11的纹理资源，但是其中存储的并非图像数据，而是特定像素的深度信息和模板值，之所以叫做深度模板缓冲，是因为这个缓冲区可以同时存储深度值和模板值。既然是纹理资源，则首先要对其进行创建，创建之前要填充一个纹理资源描述的结构体，同样由于其成员较多，所以只给出部分关键成员的设置。

D3D11\_TEXTURE2D\_DESC depthStencilDesc;

ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));

depthStencilDesc.Width = width;

depthStencilDesc.Height = height;

depthStencilDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthStencilDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

……

如代码所示，深度模板缓冲的宽高和后备缓冲相同；其缓冲每个像素元素的格式为DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT，为32值，前24为提供深度值，后8位提供模板值；缓冲的绑定标志为D3D11\_BIND\_DEPTH\_STENCIL，表示这是一个用于实现深度模板缓冲的纹理资源。完成了纹理资源描述的填充，就可以创建深度模板缓冲和深度模板视图。然后将二者设置到渲染管线当中，具体代码实现如下。

//创建纹理资源即深度模板缓冲

hr = m\_dxDevice->CreateTexture2D(

&depthStencilDesc, nullptr, m\_dxDepthStencilBuffer.GetAddressOf());

//创建深度模板缓冲视图

hr = m\_dxDevice->CreateDepthStencilView(

m\_dxDepthStencilBuffer.Get(), nullptr, m\_dxDepthStencilView.GetAddressOf());

//设置渲染目标视图和深度模板缓冲视图

m\_dxDeviceContext->OMSetRenderTargets(

1, m\_dxRenderTargetView.GetAddressOf(), m\_dxDepthStencilView.Get());

Graphics初始化的最后一部分就是设置视口映射。视口映射是将渲染得到的结果映射到输出窗口的部分范围，当然也可以不进行映射，即默认输出到整个窗口，这里为了程序的完整性，设置视口映射，并输出到整个窗口，代码实现如下。

D3D11\_VIEWPORT vp;

ZeroMemory(&vp, sizeof(vp));

//设置宽高和窗口相同，则映射至整个窗口范围

vp.Width = (float)width;

vp.Height = (float)height;

vp.MaxDepth = 1;

vp.MinDepth = 0;

vp.TopLeftX = 0;

vp.TopLeftY = 0;

//设置视口映射

m\_dxDeviceContext->RSSetViewports(1, &vp);

1. Graphics渲染

Graphics定义渲染方法（Render）。Render函数实现的是渲染的顶层逻辑，即它只负责处理全局的渲染资源，调用每个需要渲染的对象的渲染方法，不考虑每个对象具体如何渲染。函数的部分关键代码实现如下。

void Graphics::Render()

{

//刷新渲染目标视图（清空后备缓冲区）

m\_dxDeviceContext->ClearRenderTargetView(

m\_dxRenderTargetView.Get(), this->m\_renderTargetBackgroundColor);

//刷新深度模板缓冲视图（清空深度模板缓冲区）

m\_dxDeviceContext->ClearDepthStencilView(

m\_dxDepthStencilView.Get(),

D3D11\_CLEAR\_DEPTH | D3D11\_CLEAR\_STENCIL, 1.0f, 0);

……

//场景对象渲染

for (auto object : m\_sceneManager->GetObjects())

{

object->Render();

}

……

//显示渲染后的交换链后备缓冲区

HRESULT hr = m\_dxSwapChain->Present(0u, 0u);

}

其中ClearRenderTargetView函数的最后的实参即为Graphics的成员，ClearDepthStencilView函数的后两个实参分别为刷新使用的深度值和模板值。关键的部分为场景对象渲染，m\_sceneManager同样为Graphics的非DirectX资源成员变量，它是场景对象管理器的实例指针，在RenderingEngine初始化阶段传递给Graphics，通过调用它的GetObjects方法可以获取场景中的实体列表，然后对其进行遍历，并调用每个实体的渲染方法（Render），让每个实体实现自身定义的渲染逻辑。最后交换链调用Present方法进行两个缓冲区的交换，将渲染后的结果呈现在显示设备上。

## 5.4 场景管理模块实现

场景管理模块由场景管理器实现，系统将其类型定义为SceneManager。场景管理器的核心功能是场景的初始化、程序更新逻辑的实现以及场景处理接口的提供。

SceneManager的主要成员变量如表5.7所示。

表5.7 SceneManager成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_mainCamera | 场景主相机 |
| 2 | m\_objects | 场景实体列表 |
| 3 | m\_lights | 场景光源列表 |

场景主相机表示的是目前正在使用的作为人眼位置参考的虚拟相机实体，程序正在渲染的也就是主相机当前“看到”的场景。场景实体列表包含的则是当前场景生成的所有实体对象。场景光源列表也是一个实体列表，只是它包含的是所有拥有Light组件的实体对象，在渲染进行计算时，光源列表的数据会传递到着色器当中作为光照信息。

5.4.1 场景管理器初始化

SceneManager定义初始化方法（Initialize），在初始化方法当中只进行一个操作即初始化场景，SceneManager将其定义为InitializeScene。初始化场景方法的定义可以根据需求随时改变，可以什么都不做，生成一个空场景。系统默认添加一个主相机，其代码实现如下。

Object\* editModeCamera = new Object(this, GetNewObjectID());

editModeCamera->AddComponent<Attributes>("EditMode Camera", "Camera", false);

editModeCamera->AddComponent<Transform>(

DirectX::XMFLOAT3(0, 80.0f, -120.0f),

DirectX::XMFLOAT3(DirectX::XM\_PI / 12, 0, 0), DirectX::XMFLOAT3(1, 1, 1));

editModeCamera->AddComponent<Camera>();

editModeCamera->AddComponent<CameraControl>();

m\_mainCamera = editModeCamera;

this->m\_objects.push\_back(editModeCamera);

以上代码可以展示实体-组件模式的操作方式，即首先实例化一个实体，然后分别为其添加各个组件，并且可以传递实参来调用组件对应的构造函数。

这里将SceneManager的成员m\_mainCamera赋值为创建的相机，将某个相机设置为主相机，使用此相机作为渲染投射的参考点。最后将实体加入到m\_objects实体列表当中。

5.4.2 场景管理器更新

SceneManager定义更新方法（Update），其逻辑非常简单，直接遍历m\_objects列表，调用每个实体元素的更新方法（Update），具体更新的逻辑由实体定义。

## 5.5 输入控制模块实现

5.5.1 输入控制模块总体实现

系统的输入模块是基于程序窗口的回调函数来实现的。在本文渲染窗口实现小节当中提到，渲染窗口在创建时设定的窗口回调函数为MainWndProc，该函数定义在RenderingEngine类的源文件当中，其代码实现如下。

LRESULT CALLBACK MainWndProc(

HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

return RenderingEngine::GetInstance()->EngineMsgProc(hwnd, msg, wParam, lParam);

}

RenderingEngine在函数当中获取自身单例并调用了一个自定义的成员函数——EngineMsgProc，该函数即为对消息的分支判断处理函数，之所以在RenderingEngine当中定义该成员函数，是因为可以使消息处理函数能够获取RenderingEngine类的其它成员。EngineMsgProc的函数主体代码如下。

LRESULT RenderingEngine::EngineMsgProc(

HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case NotificationsType1:

//根据消息类型实现自定义功能

…

return 0;

case NotificationsType2:

//根据消息类型实现自定义功能

…

return 0;

//其它消息类型

…

default:

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

}

}

EngineMsgProc函数的四个参数由回调函数MainWndProc传入，其中hwnd为窗口句柄，msg为消息，wParam和lParam均为消息的附加信息，具体内容由msg决定。EngineMsgProc根据msg的值即消息类型进行分支判断，最后，默认情况下调用DefWindowProc函数，使用系统定义的默认消息处理机制。

msg所属的消息类型很多，其中包括键盘输入通知和鼠标输入通知，系统的鼠标和键盘输入控制模块则在此时调用键鼠模块的相关方法，实现输入反馈功能。

5.5.2 鼠标输入控制模块实现

系统定义的鼠标输入控制器类型为Mouse，因为系统程序运行期间只存在一个鼠标输入模块，所以对其实现了单例模式。Mouse拥有的主要成员变量如表5.8所示。

表5.8 Mouse成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | eventBuffer | 鼠标事件队列 |
| 2 | leftIsDown | 左键是否按下 |
| 3 | rightIsDown | 右键是否按下 |
| 4 | mbuttonDown | 中键是否按下 |
| 5 | x | 鼠标指针x坐标 |
| 6 | y | 鼠标指针y坐标 |

eventBuffer使用了C++标准库的queue来实现，其类型为std::queue<MouseEvent>，MouseEvent为自定义的事件对象类型，其内部定义了枚举类型EventType来标识自己属于哪一种事件类型，EventType包含各类鼠标事件，代码如下。

enum class EventType {

LPress,

LRelease,

RPress,

RRelease,

MPress,

MRelease,

WheelUp,

WheelDown,

Move,

RAW\_MOVE,

Invalid

};

在EngineMsgProc函数当中，如果接收到鼠标输入消息，则Mouse会调用接口根据具体消息类型构造一个MouseEvent并将其推入到eventBuffer当中，另外还会变更自身成员的状态，如鼠标指针坐标、按键按下情况等。以接收到“鼠标移动”类型的消息为例，以下为对应代码的实现。

case WM\_MOUSEMOVE:

{

int x = LOWORD(lParam);

int y = HIWORD(lParam);

Mouse::GetInstance()->OnMouseMove(x, y);

return 0;

}

当MainWndProc接收到WM\_MOUSEMOVE消息，将lParam作为参数，调用宏函数LOWORD和HIWORD可以获取鼠标指针移动后的x、y坐标。然后调用Mouse的对应处理函数OnMouseMove，即可创建对应类型的MouseEvent对象然后推入队列并更改自身成员状态，相关代码实现如下。

void Mouse::OnMouseMove(int x, int y)

{

//更改鼠标坐标

this->x = x;

this->y = y;

//创建事件对象并推入队列

this->eventBuffer.push(MouseEvent(MouseEvent::EventType::Move, x, y));

}

5.5.3 键盘输入控制模块实现

系统定义的键盘输入控制器类型为Keyboard，因为系统程序运行期间只存在一个键盘输入模块，所以对其实现了单例模式。Keyboard拥有的主要成员变量如表5.9所示。

表5.9 Keyboard成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | keyBuffer | 键盘事件队列 |
| 2 | autoRepeatKeys | 按键一直按下是否一直产生事件 |
| 3 | autoRepeatChars | 按键一直按下是否一直产生字符 |
| 4 | keyStates | 键盘所有按键状态 |
| 5 | CharBuffer | 字符队列 |

keyBuffer使用了C++标准库的queue来实现，其类型为std::queue<KeyboardEvent>，KeyboardEvent为自定义的事件对象类型，其内部定义了枚举类型EventType来标识自己属于哪一种事件类型，EventType包含键盘按键可能的事件类型，代码如下。

enum class EventType {

Press,

Release,

Invalid

};

在EngineMsgProc函数当中，如果接收到键盘输入消息，则Keyboard会调用接口根据具体消息类型构造一个KeyboardEvent并将其推入到keyBuffer当中，另外还会变更keyStates当中对应按键的状态。以接收到“键盘按键抬起”类型的消息为例，以下为对应代码的实现。

case WM\_KEYUP:

{

unsigned char keycode\_up = static\_cast<unsigned char>(wParam);

Keyboard::GetInstance()->OnKeyReleased(keycode\_up);

return 0;

}

当MainWndProc接收到 WM\_KEYUP消息，将wParam作为参数，将其强制转换类型为unsigned char可以获得其对应的按键索引值，然后调用Keyboard的对应处理函数OnKeyReleased，即可创建对应类型的KeyboardEvent对象然后推入队列并更改对应按键的状态，相关代码实现如下。

void Keyboard::OnKeyReleased(const unsigned char key)

{

//更改对应按键的状态，false表示按键没有被按下

this->keyStates[key] = false;

//创建事件对象并推入队列

this->keyBuffer.push(KeyboardEvent(KeyboardEvent::EventType::Release, key));

}

## 5.6 资源结构实现

5.6.1 顶点缓冲和索引缓冲实现

顶点缓冲和索引缓冲资源结构类型的实现是类似的，都是将DirectX的缓冲资源进行封装，其核心功能都是创建DirectX缓冲资源，并提供获取缓冲的接口。只是因为缓冲类型不同，所以在创建时部分参数设置和以及调用时的方法不同。两个缓冲类型的实现都使用了C++泛型特性，类型分别代表顶点缓冲的顶点结构类型和索引缓冲的索引值类型。

系统定义的顶点缓冲类型为VertexBuffer，索引缓冲类型为IndexBuffer，二者的核心成员分别为m\_vertexBuffer和m\_indexBuffer（均为DirectX缓冲资源），其类型均为Microsoft::WRL::ComPtr<ID3D11Buffer>。两个类型对缓冲资源的创建和获取进行封装。缓冲的创建方法均为为Instantiate，二者的函数签名如下。

bool Instantiate(ID3D11Device\* dxDevice, T\* data, UINT numVertices)；

bool Instantiate(ID3D11Device\* dxDevice, T\* data, UINT numIndices)

dxDevice参数为D3D设备，DirectX 11创建资源都需要基于D3D设备进行。两个方法的data参数分别为顶点列表数据和索引列表数据，numVertices和numIndices分别为顶点列表元素数目和索引列表元素数目。如果创建成功，则函数返回true，否则返回false。Instantiate方法的核心部分在于创建缓冲，这里以顶点缓冲列举，关键代码如下。

D3D11\_BUFFER\_DESC vbd = {};

vbd.Usage = D3D11\_USAGE\_IMMUTABLE;

vbd.ByteWidth = (m\_vertexStride \* m\_vertexCount);

vbd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

……

D3D11\_SUBRESOURCE\_DATA srd = {};

srd.pSysMem = data;

……

HRESULT hr = dxDevice->CreateBuffer(&vbd, &srd, m\_vertexBuffer.GetAddressOf());

创建DirectX缓冲需要调用CreateBuffer方法，前两个参数分别为缓冲描述结构指针和子资源描述结构指针，对应的描述结构需要实现进行填充。

缓冲描述结构的Usage设置为D3D11\_USAGE\_IMMUTABLE，代表只允许GPU读，通常用于不会进行修改的缓冲；ByteWidth设置为顶点结构的字节大小和顶点数目的乘积，对应缓冲区的总字节大小；BindFlags设置为D3D11\_BIND\_VERTEX\_BUFFER，表示顶点缓冲区，实现索引缓冲时这一成员则设置为D3D11\_BIND\_INDEX\_BUFFER，表示索引缓冲区。

子资源结构的pSysMem设置为传入的数据，这是在创建时完成资源的初始化，由于设置了缓冲描述结构的Usage为D3D11\_USAGE\_IMMUTABLE，所以之后也无法再进行修改。

另外，两个资源结构均提供获取其内部的缓冲资源的接口（GetAddressOf），其最终的使用方式是将维护的DirectX缓冲成员绑定到渲染管线。

5.6.2 网格实现

系统定义的网格资源结构类型为Mesh。网格资源结构的主要功能是将顶点缓冲和索引缓冲整合在一起，以整体的形式结合使用。Mesh类型主要应用在模型文件的加载阶段，负责将网格数据抽象读入并将其实例化为顶点缓冲和索引缓冲资源结构。

Mesh的成员变量为m\_vertexBuffer（顶点缓冲实例）和m\_indexBuffer（索引缓冲实例），Mesh在构造函数当中对两个缓冲进行实例化，构造函数标签如下。

Mesh(ID3D11Device\* device, std::vector<Vertex3D> vertices, std::vector<DWORD> indices);

Vertices为顶点列表，Vertex3D为系统自定义的默认顶点类型；indices为索引列表，DWORD为系统使用的默认索引值类型。构造函数使用以上参数对顶点缓冲和索引缓冲进行实例化，具体代码如下，这里不作赘述。

this->m\_vertexBuffer.Instantiate(device, vertices.data(), vertices.size());

this->m\_indexBuffer.Instantiate(device, indices.data(), indices.size());

除构造函数之外，Mesh提供了获取顶点缓冲结构和索引缓冲结构的外部接口。

5.6.3 常量缓冲实现

系统定义的常量缓冲资源结构类型为ConstantBuffer。常量缓冲和顶点缓冲及索引缓冲一样均是对DirectX缓冲资源（ID3D11Buffer）的封装，但常量缓冲要比上述两种缓冲复杂，因为常量缓冲的数据可能是需要经常修改的，需要提供数据的修改和获取功能，且常量缓冲在渲染管线当中绑定的位置也不固定，它可以应用于各类型的着色器阶段。

常量缓冲要在着色器当中使用首先要在HLSL代码中进行定义，定义的形式如下。

cbuffer BufferName : register(b#)

{

……//变量定义

}

首先使用cbuffer关键字表明这是一个常量缓冲，BufferName 为自定义的缓冲名称，，register关键字则定义缓冲所在寄存器当中的槽位，b表示缓冲（buffer），每个着色器阶段最多允许使用15个常量缓冲区，所以#的范围为0到14。在常量缓冲当中可以定义变量（实际上并非变量，因为常量缓冲作为常量一般并不能够修改，但这里可以以变量的形式来理解），每个变量在缓冲区内存中顺序分布，每个缓冲区最多可以容纳4096个变量。

ConstantBuffer的成员变量如表5.10所示。

表5.10 ConstantBuffer成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_constantBuffer | DirectX缓冲资源对象 |
| 2 | m\_data | 常量缓冲存储的数据 |
| 3 | m\_dirty | 常量缓冲的数据是否被修改过 |
| 4 | m\_slot | 常量缓冲所在寄存器槽位 |
| 5 | m\_byteWidth | 常量缓冲字节大小 |

ConstantBuffer的核心功能包括创建常量缓冲、绑定、解绑设置数据以及更新数据。

常量缓冲的创建和顶点缓冲及索引缓冲类似，但是部分参数的设置不同，函数为Instantiate，具体代码实现如下。

D3D11\_BUFFER\_DESC cbd = {};

cbd.Usage = D3D11\_USAGE\_DYNAMIC;

cbd.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

cbd.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

……

cbd.ByteWidth = static\_cast<UINT>(m\_byteWidth + (16 - (m\_byteWidth % 16)));

HRESULT hr = dxDevice->CreateBuffer(&cbd, nullptr, this->m\_constantBuffer.GetAddressOf());

将缓冲描述结构的Usage设置为D3D11\_USAGE\_DYNAMIC，并将CPUAccessFlags设为D3D11\_CPU\_ACCESS\_WRITE，表示允许CPU写和GPU读，因为常量缓冲的数据需要经常更新；BindFlags 设置为D3D11\_BIND\_CONSTANT\_BUFFER，表示常量缓冲；ByteWidth的设置采用了一个数学计算，因为HLSL的常量缓冲区本身以及对它的读写操作需要严格按16字节对齐，所以在C++创建常量缓冲区时大小必须为16字节的倍数，这里假如m\_byteWidth并非16的整数倍，则将值设置为大于m\_byteWidth的最小16整数倍的数值。最后就是使用描述结构创建缓冲。

常量缓冲的绑定和解绑都是面向着色器类型的，二者的函数签名如下。

void Bind(ID3D11DeviceContext\* deviceContext, Shader::ShaderType shaderType);

void UnBind(ID3D11DeviceContext\* deviceContext, Shader::ShaderType shaderType);

deviceContext为D3D上下文，即渲染管线，由它执行绑定或解绑操作。ShaderType 为系统自定义的着色器枚举类型，将在着色器实现的小节中阐述。这里根据着色器类型进行分支判断，并调用相应的绑定和解绑方法，以顶点着色器为例，代码如下。

case Shader::ShaderType::VertexShader:

deviceContext->VSSetConstantBuffers(m\_slot,1,m\_constantBuffer.GetAddressOf());

如果着色器类型为顶点着色器，则上下文调用其对应的方法VSSetConstantBuffers，第一个参数代表的是绑定的寄存器槽位，第二个参数表示绑定缓冲的数量，最后的参数则是缓冲资源对象。

解绑调用的方法和绑定是一样的，只是将绑定的对象设置为空，代码如下。

ID3D11Buffer\* pBuffers[] = { nullptr };

case Shader::ShaderType::VertexShader:

deviceContext->VSSetConstantBuffers(m\_slot, 1, pBuffers);

常量缓冲的核心功能之一是可以进行数据的修改、更新和获取。ConstantBuffer维护的数据对象类型为字节数组（BYTE[]），即它可以对任意类型的数据结构进行存储。设置数据的方法为SetData，其代码实现如下。

void ConstantBuffer::SetData(const void\* data, UINT byteOffset, UINT byteCount)

{

if (byteOffset > m\_byteWidth)

return;

if (byteOffset + byteCount > m\_byteWidth)

byteCount = m\_byteWidth - byteOffset;

memcpy\_s(m\_data.get() + byteOffset, byteCount, data, byteCount);

m\_dirty = true;

}

参数data为const void\*，即空指针类型，意味着可以输入任意类型的数据实参；参数byteOffset代表设置数据的目标位置在成员m\_data字节数组当中的偏移量，即可以通过调整byteOffset来决定从哪一个变量开始进行数据修改；参数byteCount代表输入数据的字节大小，决定数据修改的跨度。函数首先判断字节偏移量是否已经大于整个缓冲区的字节大小，如果大于则没有意义，然后修正byteCount的数值。最后使用memcpy\_s将输入数据拷贝到成员m\_data的内存位置上，并设置m\_dirty 为true，表示数据被更改过了，且没有被提交至渲染管线的常量缓冲中。memcpy\_s的作用是将缓冲区资源进行拷贝并传递到另一个缓冲区位置，其函数签名如下。

errno\_t memcpy\_s(void \*dest, size\_t destSize, const void \*src, size\_t count );

参数dest表示目标缓冲区，destSize表示目标缓冲区的大小，src为从中进行复制操作的缓冲区，count为要复制的字节数。

从常量缓冲获取数据和SetData的实现逻辑相同，只是换成将m\_data的数据拷贝输出到其它的缓冲区，这里不再赘述。

在完成了ConstantBuffer的数据修改之后，还需要将更新后的数据提交到渲染管线真正的常量缓冲资源当中，ConstantBuffer定义UpdateConstantBuffer方法进行实现，其代码实现如下。

void ConstantBuffer::UpdateConstantBuffer(ID3D11DeviceContext\* dxDeviceContext)

{

if (m\_dirty)

{

D3D11\_MAPPED\_SUBRESOURCE mapped\_data;

HRESULT hr = dxDeviceContext->Map(

this->m\_constantBuffer.Get(),

0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mapped\_data

);

memcpy\_s(mapped\_data.pData, m\_byteWidth, m\_data.get(), m\_byteWidth);

dxDeviceContext->Unmap(this->m\_constantBuffer.Get(), 0);

m\_dirty = false;

}

}

首先判断ConstantBuffer的数据成员m\_data是否被修改过，防止无效提交。接下来对缓冲区资源机型更新，首先Direct3D上下文调用Map函数，它可以获取指向缓冲区中数据的指针并拒绝GPU对该缓冲区的访问，将映射到内存的指针存储至mapped\_data当中，然后同样调用memcpy\_s函数，将ConstantBuffer成员m\_data中的数据拷贝到映射后的内存当中，最后调用Unmap让指向资源的指针无效并重新启用GPU对该资源的访问权限。以上就完成了对渲染管线常量缓冲资源的数据更新，最后将m\_dirty设为false，表示所有数据都已经提交。

5.6.4 纹理实现

系统定义的纹理资源结构类型为Texture。其核心功能是将DirectX纹理资源和着色器视图资源封装，并提供创建、绑定、解绑以及成员的外部获取接口。Texture的主要成员变量如表5.11所示。

表5.11 Texture成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_textureFilePath | 纹理文件路径 |
| 2 | m\_texture2D | 纹理资源对象 |
| 3 | m\_shaderResourceView | 着色器资源视图 |
| 4 | m\_slot | 纹理所在寄存器槽位 |

由于纹理资源加载的实现引入了DirectXTK库中DirectXTK\WICTextureLoader.h文件，它将纹理资源的读取、创建过程进行了封装，可以直接将纹理文件转化为资源并和着色器资源视图捆绑，且纹理资源的只需要通过着色器资源视图就可以实现在渲染管线的绑定和解绑，所以系统暂时没有使用到m\_texture2D成员。

Texture定义的实例化方法为Instantiate。其核心功能是读取给定路径的纹理文件并将结果转化为纹理资源，然后创建对应的着色器资源视图，其代码实现如下。

this->m\_textureFilePath = textureFilePath;

HRESULT hr= DirectX::CreateWICTextureFromFile(

m\_dxDevice,

DataTypeConverter::StringToWideString(m\_textureFilePath).c\_str(),

nullptr,

m\_shaderResourceView.GetAddressOf()

);

纹理资源的加载使用WICTextureLoader提供的方法CreateWICTextureFromFile，可以直接以纹理文件路径为参数生成着色器资源视图，并将其存储到Texture成员m\_shaderResourceView中。

纹理资源的绑定、解绑功能的实现和常量缓冲对应部分的实现原理相同，这里不再赘述。

5.6.5 采样器状态实现

系统定义的采样器状态资源结构类型为SamplerState。其核心功能是将DirectX采样器资源的创建、绑定和解绑操作进行封装。SamplerState拥有的成员为m\_samplerState（DirectX采样器资源，类型为Microsoft::WRL::ComPtr<ID3D11SamplerState>）和m\_slot（采样器资源在寄存器的槽位）。

SamplerState定义的采样器状态资源创建方法为Instantiate，其内部关键代码实现如下。

D3D11\_SAMPLER\_DESC sampDesc;

ZeroMemory(&sampDesc, sizeof(sampDesc));

sampDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

sampDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

sampDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

sampDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

……

HRESULT hr = dxDevice->CreateSamplerState(

&sampDesc, m\_samplerState.GetAddressOf());

首先需要填充采样器描述结构，其中成员AddressU和AddressV均设置为D3D11\_TEXTURE\_ADDRESS\_WRAP，这代表这采样器在对纹理采样时在U、V两个方向的寻址模式，当输入的uv值不在0到1范围内时，寻址模式可以解释边界外的采样情况。D3D11\_TEXTURE\_ADDRESS\_WRAP值代表将指定纹理坐标分量的值[t, t + 1], t ∈ Z映射到[0.0, 1.0]，实现的效果就是把原纹理进行平铺，这也是系统默认使用的寻址模式。完成描述结构的填充之后，DirectX3D设备就可以使用其作为参数调用CreateSamplerState创建采样器状态。

SamplerState的绑定和解绑操作实现和ConstantBuffer的实现同理，这里不再赘述。

5.6.6 结构缓冲实现

系统定义的结构缓冲资源结构类型为StructuredBuffer。结构缓冲资源同样是缓冲，即拥有的核心成员m\_structuredBuffer也是DirectX缓冲资源，但是StructuredBuffer对标的是渲染管线中的单类型结构体数组资源，所以它维护数据是列表形式的bufferData，其类型为std::vector<T>，使用泛型来实现。另外，结构缓冲资源无法向常量缓冲一样直接绑定到渲染管线，而是需要借助着色器资源视图，对应的成员为m\_structuredBufferSRV。

StructuredBuffer的核心功能是对结构缓冲资源的创建、获取、数据的修改和更新。

StructuredBuffer定义的创建方法为Instantiate，主要由两部分操作构成，一是结构缓冲资源的创建，二是对应着色器资源视图的创建。结构缓冲的创建时的描述结构关键成员填充代码如下。

D3D11\_BUFFER\_DESC cbd = {};

cbd.Usage = D3D11\_USAGE\_DYNAMIC;

cbd.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

cbd.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE;

cbd.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;

cbd.StructureByteStride = sizeof(T);

cbd.ByteWidth = numElements \* sizeof(T);

由于结构缓冲数据也需要经常变更，所以Usage设置为D3D11\_USAGE\_DYNAMIC；

BindFlags并非设为某一缓冲类型枚举值，而是D3D11\_BIND\_SHADER\_RESOURCE，因为结构缓冲需要借助着色器资源视图进行绑定，MiscFlags须设为D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED；StructureByteStride表示的是结构体的字节大小，所以设为sizeof(T)；最后的ByteWidth设置的是整个结构缓冲区的大小，所以设为结构体数组元素数目和结构体大小的乘积。

然后是着色器资源视图的创建，其代码实现如下。

D3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvd = {};

srvd.Format = DXGI\_FORMAT\_UNKNOWN;

srvd.ViewDimension = D3D11\_SRV\_DIMENSION\_BUFFER;

srvd.Buffer.FirstElement = 0; //起始元素的索引

srvd.Buffer.NumElements = numElements; //元素数目

hr = dxDevice->CreateShaderResourceView(

m\_structuredBuffer.Get(), &srvd, m\_structuredBufferSRV.GetAddressOf());

StructuredBuffer定义的设置数据的方法为SetData，其实现逻辑是将一个新的数据数组传入到StructuredBuffer维护的成员bufferData中，其实现代码如下。

void SetData(T\* data, UINT elementSize, UINT numElements)

{

T\* first = data;

T\* last = first + elementSize \* numElements;

bufferData.assign(first, last);

}

参数data代表传入数据的起始位置指针，elementSize为数据数组元素的大小，numElements为元素数目。使用C++ vector的assign方法可以直接进行数据覆盖。

StructuredBuffer的结构缓冲资源数据更新方法和常量缓冲对应部分的实现原理相同，这里不再赘述。

5.6.7 着色器实现

系统定义的着色器资源结构类型为Shader。Shader是对DirectX各类着色器资源对象的封装，包含各类着色器资源对象成员。由于材质拥有的着色器的类型和数目都是不确定的，而Shader结构可以代表任意着色器类型中的一种，这样材质结构只需要维护一个存储Shader对象的容器即可。

和ShaderParameter的实现逻辑类似，Shader也需要一个标识来表明自己具体属于哪一种着色器资源，系统定义了对应各类型着色器的枚举类型，代码如下。

enum class ShaderType

{

UnknownShaderType = 0, //未知着色器类型

VertexShader = 1, //顶点着色器

PixelShader = 2, //像素着色器

TessellationControlShader = 3, //外壳着色器

TessellationEvaluationShader = 4, //域着色器

GeometryShader = 5, //几何着色器

ComputeShader = 6 //计算着色器

};

Shader在实例化阶段根据输入的着色器类型对其着色器资源对象选择性初始化，除了属于DirectX着色器资源对象的成员，其它的成员如表5.12所示。

表5.12 Shader成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_shaderType | 着色器类型 |
| 2 | m\_shaderFilePath | 着色器文件路径 |
| 3 | m\_entryPoint | 着色器编译入口点 |
| 4 | m\_shaderModel | 着色器模型版本 |
| 5 | m\_blob | 着色器编译后的二进制数据对象 |

Shader在构造函数调用阶段对以上的成员进行初始化，确定着色器的基本信息，为之后的着色器实例化做准备。

Shader定义的实例化方法为Instantiate。在实例化阶段，Shader首先要对着色器代码文件进行编译，获取编译后的二进制数据对象并存储到成员m\_blob中，然后根据自身的着色器类型使用m\_blob的二进制数据创建着色器。

着色器的编译使用了系统定义的CreateShaderFromFile函数，其内部实现的核心是调用d3dcompiler.h库当中的D3DCompileFromFile函数，其函数签名如下。

HRESULT D3DCompileFromFile(

LPCWSTR pFileName, //编译的.hlsl文件

CONST D3D\_SHADER\_MACRO\* pDefines,

ID3DInclude\* pInclude,

LPCSTR pEntrypoint, //入口函数名

LPCSTR pTarget, //使用的着色器模型

UINT Flags1,

UINT Flags2,

ID3DBlob\*\* ppCode, //获得着色器的二进制块

ID3DBlob\*\* ppErrorMsgs);

关键的参数已通过注释标注。pFileName传入m\_shaderFilePath成员；pEntrypoint传入m\_entryPoint；pTarget传入m\_shaderModel；ppCode传入m\_blob，这是一个输出参数，将编译后的二进制数据传出到m\_blob当中。

在获取到着色器编译后的二进制对象之后，就可以使用它来创建具体的着色器，以顶点着色器为例，代码实现如下。

case ShaderType::VertexShader:

hr = dxDevice->CreateVertexShader(

m\_blob->GetBufferPointer(), m\_blob->GetBufferSize(), nullptr, &m\_vertexShader);

调用Direct3D设备的CreateVertexShader函数，将二进制数据信息作为参数传入，创建顶点着色器并其保存在成员m\_vertexShader中。其余类型着色器的创建实现同理，这里不再赘述。

5.6.8 着色器资源参数实现

系统定义的着色器资源参数结构类型为ShaderParameter。着色器资源参数代表的是上述四种资源结构——常量缓冲、纹理、采样器状态和结构缓冲的其中一种，之所以使用ShaderParameter将这些结构进行进一步的封装，是因为在材质界面的参数显示功能实现中，需要逐个显示材质对应着色器包含的可调节参数，但是由于材质的着色器是可更换的，所以对应的着色器资源也是不确定的，这样材质就无法维护一个稳定的着色器参数列表。而通过ShaderParameter进行封装之后，材质结构只需要维护一个确定的ShaderParameter列表即可，至于每个ShaderParameter究竟对应着哪一种资源结构，则根据当前着色器的内容来决定。由于ShaderParameter内容不定的特性，所以需要一个标识来表明它具体属于四种资源的哪一种。系统定义了枚举类型ShaderParameterType，表明ShaderParameter的具体类型，具体代码如下。

enum class ShaderParameterType

{

Invalid = 0, // 无效类型

ConstantBuffer = 1, //常量缓冲类型

Texture = 2, // 纹理类型

Sampler = 3, // 纹理采样器类型

StructuredBuffer = 4 //结构缓冲类型

};

ShaderParameter拥有对应四种资源结构的成员变量，在其构造阶段，根据输入的资源结构类型枚举值选择其中一个进行实例化，并将成员m\_shaderParameterType设置为对应的枚举值。

ShaderParameter的另一个关键成员是ownerShaderTypesMap，该成员是一个C++ map对象，其键值对类型为<Shader::ShaderType, bool>，键为着色器类型，值为布尔值。ownerShaderTypesMap的作用是确定着色器资源所属的着色器类型，之所以使用一个map来存储，是因为一个着色器资源可能会被多个着色器阶段使用，在对着色器进行反射时，如果发现资源被使用，则将该着色器对应类型在ownerShaderTypesMap中的值设置为true，表示该类型着色器使用了此着色器资源。

ShaderParameter提供了上层的绑定和解绑方法，并在底层根据ShaderParameterType的着色器资源类型和ownerShaderTypesMap的着色器类型调用Direct3D上下文对应的方法进行绑定和解绑。

5.6.9 材质实现

系统定义的材质资源结构类型为Material。材质结构是上述所有资源类型的聚合体，管理模型网格渲染实现除网格本身之外所有的资源信息。Material拥有的主要成员变量如表5.13所示。

表5.13 Material成员变量

|  |  |  |
| --- | --- | --- |
| **表序号** | **变量名** | **变量含义** |
| 1 | m\_shaderFilePath | 着色器文件路径 |
| 2 | m\_shadersMap | <着色器类型-着色器> map |
| 3 | shaderParametersMap | <参数名-着色器资源参数>map |
| 4 | m\_dxInputLayout | 输入布局 |
| 5 | m\_dxDepthStencilState | 深度模板状态 |
| 6 | m\_dxBlendState | 混合状态 |
| 7 | m\_dxRasterizerState | 光栅化状态 |

其中m\_shadersMap是C++ map类型，其键值对为<Shader::ShaderType, Shader\*>，m\_shadersMap的主要功能是存储着色器对象，且便于进行着色器的遍历和通过着色器类型获取着色器对象。shaderParametersMap同样是C++ map类型，其键值对为<std::string, ShaderParameter\*>，shaderParametersMap的主要功能是存储着色器资源参数对象，便于进行着色器资源参数的遍历和通过名称获取着色器资源参数对象。

Material的定义的实例化方法为Instantiate。在实例化方法当中，Material通过输入的着色器文件路径，创建指定类型的着色器资源结构（至少包含一个顶点着色器和一个像素着色器），并将其加入到m\_shadersMap容器当中。然后对着色器容器进行遍历，对每个着色器执行反射操作，反射需要的是着色器编译后的二进制数据，调用Shader的GetBlob()方法即可获取。编译好的着色器二进制数据中蕴含着丰富的信息，可以通过着色器反射机制来获取自己所需要的东西。

Material的Instantiate函数程序流程如图5.3所示。

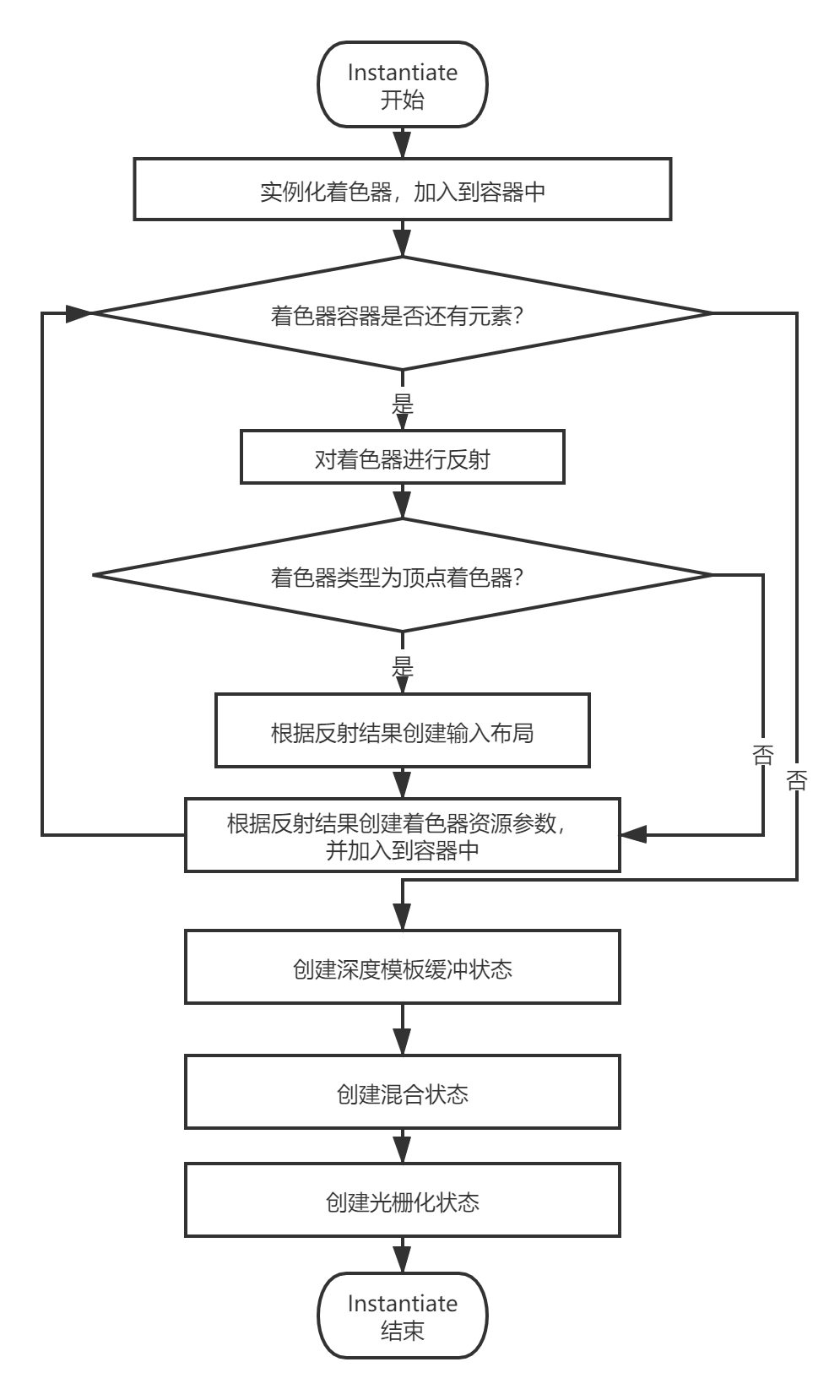


图5.3 Instantiate函数程序流程图

Instantiate函数实现的关键在于对着色器的反射以及对其反射结果的使用。通过调用d3dcompiler.h中的D3DReflect方法可以实现对着色器的反射操作，其函数签名如下。

HRESULT D3DReflect(

LPCVOID pSrcData, //编译着色器得到的二进制信息

SIZE\_T SrcDataSize, //二进制信息字节数

REFIID pInterface, //COM组件的GUID

void \*\*ppReflector //输出的反射结果

);

在进行着色器反射之前，首先需要创建一个对象shaderReflection来存储反射结果，其类型为Microsoft::WRL::ComPtr<ID3D11ShaderReflection>。然后调用D3DReflect函数，将着色器编译后的二进制对象和着色器反射对象等作为参数传入，获得反射结果。

shaderReflection中存储了大量的着色器相关信息，首先获取着色器描述信息，代码实现如下。

D3D11\_SHADER\_DESC shaderDescription;

hr = shaderReflection->GetDesc(&shaderDescription);

由于D3D11\_SHADER\_DESC结构包含的成员很多，这里不做展示，之后只对使用的成员进行阐述。在输入布局的创建当中，通过其InputParameters成员可以获取顶点着色器输入的顶点结构的变量数目。在读取着色器资源时，通过其BoundResources成员可以获取着色器使用的资源结构的数目。

在创建着色器资源参数过程中，需要遍历着色器使用的每个资源，获取其相关信息并抽象为系统定义的着色器资源参数结构，然后加入到容器当中。在每次遍历中，首先获取着色器资源绑定的描述信息，其代码实现如下。

D3D11\_SHADER\_INPUT\_BIND\_DESC shaderInputBindDesc;

hr = shaderReflection->GetResourceBindingDesc(i, &shaderInputBindDesc);

D3D11\_SHADER\_INPUT\_BIND\_DESC描述着色器资源如何绑定到着色器输入，可以提供如资源所在的寄存器槽位、资源名称、资源类型等信息。这里根据资源类型进行分支处理，针对不同类型实例化不同的着色器资源参数结构。在创建参数之前，首先需要判断当前资源是否已经处理过了，因为不同着色器可能会引用相同的着色器资源，但是这些资源是公用的，Material只需维护其中一份即可。如果资源已经被处理过并创建了对应的着色器资源参数，则将当前的着色器类型记录到shaderParametersMap当中，表示该资源同时被该类型着色器使用，然后跳过此资源的处理，相关代码的实现如下。

//如果Material已经记录了当前类型的资源

if (shaderParametersMap.count(resourceName) != 0)

{

//记录当前资源对应的着色器类型

shaderParametersMap[resourceName]->ownerShaderTypesMap[value.first] = true;

//跳过这一资源的处理

continue;

}

根据shaderInputBindDesc的Type成员可以获取着色器资源的类型，这里使用到其中四种类型。D3D\_SIT\_CBUFFER——常量缓冲资源

D3D\_SIT\_TEXTURE——纹理资源

D3D\_SIT\_SAMPLER——采样器资源

D3D\_SIT\_STRUCTURED——结构缓冲资源

接下来的操作就是针对不同的资源类型进行处理，创建对应类型的着色器资源参数结构，并将其加入到shaderParametersMap当中存储。

完成着色器资源参数的创建和存储之后，就是对状态对象的创建.Direct3D是基于状态机的，状态是对于渲染管线部分阶段操作属性的调节，我们可以通过修改这些状态来修改渲染管线的当前行为。首先创建深度模板缓冲状态，代码实现如下。

D3D11\_DEPTH\_STENCIL\_DESC dsc;

ZeroMemory(&dsc, sizeof(dsc));

dsc.DepthEnable = true;

dsc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

dsc.DepthFunc = D3D11\_COMPARISON\_LESS\_EQUAL;

dsc.StencilEnable = false;

hr = m\_dxDevice->CreateDepthStencilState(&dsc, m\_dxDepthStencilState.GetAddressOf());

创建深度模板缓冲状态前要对其描述结构进行填充。将DepthEnable成员设为true，表示开启深度测试；DepthWriteMask设为D3D11\_DEPTH\_WRITE\_MASK\_ALL表示写入深度；DepthFunc成员对应的是深度测试使用的像素深度比较方法，这里设为D3D11\_COMPARISON\_LESS\_EQUAL，表示源像素的深度需要小于等于帧缓冲当中像素的深度值才能够通过深度测试。StencilEnable设为false，表示关闭模板测试。最后调用CreateDepthStencilState函数创建深度模板缓冲状态。

接下来创建混合状态，混合状态的核心功能是决定源像素颜色和帧缓冲当中的像素颜色之间如何进行混合并得到最终的颜色。同样对描述结构进行填充，代码实现如下。

D3D11\_BLEND\_DESC bd;

ZeroMemory(&bd, sizeof(bd));

D3D11\_RENDER\_TARGET\_BLEND\_DESC rtbd;

ZeroMemory(&rtbd, sizeof(rtbd));

rtbd.BlendEnable = false;

rtbd.BlendOp = D3D11\_BLEND\_OP\_ADD; //颜色混合运算方法

rtbd.SrcBlend = D3D11\_BLEND\_SRC\_ALPHA; //源颜色因子

rtbd.DestBlend = D3D11\_BLEND\_INV\_SRC\_ALPHA; //缓冲区颜色因子

rtbd.SrcBlendAlpha = D3D11\_BLEND\_ONE; //源透明度因子

rtbd.DestBlendAlpha = D3D11\_BLEND\_ZERO; //缓冲区透明度因子

rtbd.BlendOpAlpha = D3D11\_BLEND\_OP\_ADD; //透明度混合运算方法

rtbd.RenderTargetWriteMask = D3D11\_COLOR\_WRITE\_ENABLE\_ALL; //颜色遮罩

bd.RenderTarget[0] = rtbd;

hr = m\_dxDevice->CreateBlendState(&bd, this->m\_dxBlendState.GetAddressOf());

描述结构各个成员的含义已在注释当中标注，

最后是创建光栅化状态。渲染管线的光栅化阶段是不可编程的，但是在这一阶段有许多任务需要完成。在顶点着色器、几何着色器输出处理过的顶点数据后，光栅化阶段会对传入的顶点数据尤其是齐次空间位置四维向量进行透视除法，并判断顶点是否位于NDC（标准设备坐标）空间中。然后根据设定的光栅化状态决定顶点顺（逆）时针排布顺序决定三角形是否被剔除。

创建光栅化状态也需要填充描述结构，因为描述结构成员较多，这里只给出系统用到的成员设置，其余成员采用默认值，具体代码实现如下。

D3D11\_RASTERIZER\_DESC rd;

ZeroMemory(&rd, sizeof(rd));

rd.FillMode = D3D11\_FILL\_SOLID;

rd.CullMode = D3D11\_CULL\_BACK;

//创建光栅化状态

hr = m\_dxDevice->CreateRasterizerState(&rd, m\_dxRasterizerState.GetAddressOf());

FillMode设为D3D11\_FILL\_SOLID表示填充绘制的三角形内部像素，如果设置为D3D11\_FILL\_WIREFRAME，则只绘制三角形的外框；CullMode设为D3D11\_CULL\_BACK表示光栅化阶段剔除三角形背面。最后调用Direct3D设备的CreateRasterizerState方法创建光栅化状态并保存到成员m\_dxRasterizerState当中。

## 5.7 实体渲染系统实现

如4.7小节实体渲染系统设计部分介绍的，渲染过程主要通过网格渲染组件和材质管理组件协作完成，二者缺一则无法实现渲染效果。网格渲染组件的Render函数首先使用组件判断接口HasComponent来判断实体是否拥有材质管理组件，如果没有则不进行渲染。

渲染的框架是对网格列表m\_meshes进行遍历，对每个网格使用材质进行渲染。每次迭代得到一个网格结构（Mesh），进而获取其顶点缓冲和索引缓冲并将其绑定至渲染管线，该网格的作用到此为止。

接下来是材质相关成员的绑定。首先根据网格所在列表当中的位置索引获取材质管理组件总材质列表相同位置的材质结构对象。将其拥有的输入布局、着色器资源、深度模板缓冲状态、混合状态、光栅化状态均绑定至渲染管线。相关代码实现如下。

//设置顶点缓冲

UINT offsets = 0;

this->m\_dxDeviceContext->IASetVertexBuffers(

0, 1, m\_meshes[i].GetVertexBuffer().GetAddressOf(), m\_meshes[i].GetVertexBuffer().StridePtr(),

&offsets);

//设置索引缓冲

this->m\_dxDeviceContext->IASetIndexBuffer(

m\_meshes[i].GetIndexBuffer().Get(), DXGI\_FORMAT\_R32\_UINT, 0);

//设置着色器

materialManager->materials[i].BindShaders(m\_dxDeviceContext);

//设置输入布局

this->m\_dxDeviceContext->IASetInputLayout(materialManager->materials[i].GetInputLayout());

//设置图元类型

this->m\_dxDeviceContext->IASetPrimitiveTopology(

D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

//设置深度-模板缓冲状态

this->m\_dxDeviceContext->OMSetDepthStencilState(

materialManager->materials[i].GetDepthStencilState(), 0);

//设置混合状态

this->m\_dxDeviceContext->OMSetBlendState(

materialManager->materials[i].GetBlendState(), nullptr, 0xffffffff);

//设置光栅化状态

this->m\_dxDeviceContext->RSSetState(materialManager->materials[i].GetRasterizerState());

完成了渲染管线基础资源的绑定，接下来需要将此次迭代的材质结构当中存储的着色器资源参数绑定至渲染管线对应着色器阶段，另外将每个着色器资源参数的修改提交至渲染管线。对材质结构的shaderParametersMap成员包含的元素进行遍历，调用每个着色器资源参数的Bind接口进行资源绑定，调用UpdateParameterResource接口来向渲染管线提交对资源的变更，具体代码实现如下。

pair\_name\_shaderParameter.second->Bind(m\_dxDeviceContext);

pair\_name\_shaderParameter.second->UpdateParameterResource(this->m\_dxDeviceContext);

上述的一系列操作即完成了一帧当中渲染一个实体所有网格所需的准备工作，接下来调用Direct3D上下文的绘制方法，对实体进行渲染，代码实现如下。

this->m\_dxDeviceContext->DrawIndexed(this->m\_meshes[i].GetIndexBuffer().IndexCount(), 0, 0);

DrawIndexed函数对传入渲染管线当中的顶点数据按索引顺序进行绘制。在完成了绘制之后，需要遍历材质结构的shaderParametersMap成员包含的元素，将每个着色器资源参数从渲染管线上解绑，以免影响下一次的绘制，解绑操作通过调用ShaderParameter的UnBind接口完成，其代码实现如下。

pair\_name\_shaderParameter.second->UnBind(m\_dxDeviceContext);

## 5.8 用户界面模块实现

5.8.1 用户界面总体架构实现

用户界面模块的实现采用面向对象的模式，系统定义UserInterface类型作为所有用户界面元素的顶层管理者。如用户界面模块设计小节当中提到的，由于用户界面的实现引入了Dear ImGui库，其运行需要依托于DirectX渲染管线，所以UserInterface属于图形渲染管理器（Graphic）类型当中的成员。UserInterface包含窗口内所有的用户界面元素对象，并定义初始化方法（Initialize）和渲染方法（Render），两个方法均由Graphics在其对应的初始化和渲染阶段调用。

Graphics定义了InitializeUI方法来对用户界面模块进行初始化，其代码实现如下。

bool Graphics::InitializeUI(HWND hwnd, SceneManager\* sceneManager)

{

return m\_userInterface->Initialize(

hwnd, this->m\_dxDevice.Get(), this->m\_dxDeviceContext.Get(), sceneManager);

}

UserInterface的渲染在Graphics的Render方法当中调用，位于场景对象渲染之后。

1. UserInterface初始化

Initialize函数的代码实现如下。

bool UserInterface::Initialize(

HWND hwnd, ID3D11Device\* device,ID3D11DeviceContext\* deviceContext, SceneManager\* sceneManager)

{

m\_sceneManager = sceneManager;

InitializeImGui(hwnd, device, deviceContext);

InitializeModules(sceneManager);

return true;

}

在初始化阶段，UserInterface首先要完成对Dear ImGui的初始化，然后完成对其包含的所有子用户界面对象的初始化。

UserInterface定义初始化Dear ImGui的方法为InitializeImGui，其关键代码的实现如下。

void UserInterface::InitializeImGui(

HWND hwnd, ID3D11Device\* device, ID3D11DeviceContext\* deviceContext)

{

IMGUI\_CHECKVERSION();

ImGui::CreateContext();

ImGuiIO& io = ImGui::GetIO();

ImGui\_ImplWin32\_Init(hwnd);

ImGui\_ImplDX11\_Init(device, deviceContext);

…

}

ImGui的运行前准备需要渲染窗口提供的窗口句柄（hwnd）和图形渲染管理器提供的Direct3D设备和上下文，Graphic将这些参数传递给UserInterface。在ImGui初始化完成后UserInterface则对用户界面各个子模块进行初始化，定义函数为InitializeModules，其实现即调用各个子模块的初始化方法，代码实现如下。

void UserInterface::InitializeModules(SceneManager\* sceneManager)

{

this->m\_hierarchy->Initialize(sceneManager);

this->m\_inspector->Initialize(sceneManager);

this->m\_topToolbar->Initialize(sceneManager);

}

1. UserInterface渲染

UserInterface实现的是用户界面渲染的顶层逻辑，负责调用每个子模块的渲染方法，而每个子模块则自定义渲染方法的具体实现。Render方法的代码实现如下。

void UserInterface::Render()

{

ImGui\_ImplDX11\_NewFrame();

ImGui\_ImplWin32\_NewFrame();

ImGui::NewFrame();

//=======子模块绘制=======

this->m\_topToolbar->Render();

this->m\_hierarchy->Render();

this->m\_inspector->Render();

//======绘制结束=====

ImGui::Render();

ImGui\_ImplDX11\_RenderDrawData(ImGui::GetDrawData());

}

ImGui在每一帧的界面元素绘制之前需要做准备工作，在界面元素绘制之后也要进行渲染工作，所以需要将用户界面的子模块的绘制放置在二者之间，否则无法完成渲染。

5.8.2 场景对象结构面板实现

场景对象结构面板负责以列表的形式显示场景中所有可操作的实体，为用户提供选择实体进行操作的接口。系统将其类型定义为Hierarchy，拥有的成员为m\_sceneManager（场景管理器实例）和selected（静态的选定对象的索引值）。

Hierarchy定义了初始化（Initialize）和渲染（Render）方法。在初始化阶段Graphics将场景管理器实例以参数形式传递并赋值给Hierarchy的m\_sceneManager成员，因为Hierarchy负责显示场景中的实体信息，所以一个场景管理器实例的引用是必需的。selected的作用是为之后的细节信息面板确定当前选定的实体。

Hierarchy渲染方法的实现是通过遍历场景管理器实例的所有实体对象并逐个绘制对应的UI元素来实现的。首先需要绘制一个窗口元素来承载所有的实体UI元素，调用ImGui的Begin方法，代码结构如下。

if (ImGui::Begin("HIERARCHY"))

{

//遍历场景管理器的实体列表，绘制实体UI元素

…

}

ImGui::End();

ImGui绘制一个窗口后需要以ImGui::End()收尾，表示窗口绘制完毕。针对每个实体UI元素则是绘制每个实体的名字，另外因为Hierarchy实现的不只是显示功能，还有交互选择功能，所以使用ImGui的Selectable元素对实体名进行绘制，相关代码如下。

if (ImGui::Selectable(label, selected == i))

{

selected = i;

}

其中label是遍历场景管理器的实体列表时当前实体的名字，i为实体在列表中的索引值。如果Selectable元素被选择，则函数返回true，选定后，将Hierarchy的静态的选定对象索引值设为该值，确定被选择的实体对象。

5.8.3 细节信息面板实现

细节信息面板负责显示从Hierarchy当中选定的实体对象的具体信息，系统将其类型定义为Inspector，拥有的成员为m\_sceneManager（场景管理器实例）和preSelected（静态的前一次选定的对象索引值）。

Inspector定义了初始化（Initialize）和渲染（Render）方法。在初始化阶段Graphics将场景管理器实例以参数形式传递并赋值给Inspector的m\_sceneManager成员，因为Inspector需要根据Hierarchy得到的索引值从实体列表获取对应实体对象，所以场景管理器实例的引用是必需的。preSelected的作用是和Hierarchy的selected成员做比较，以确定当前选定的实体UI元素有没有发生改变，如果发生了改变，则需要Inspector渲染新选择的实体的信息，否则就继续展示之前的实体信息。

5.8.4 窗口工具栏实现

窗口工具栏负责提供全局的功能接口，系统将其类型定义为TopToolbar，拥有的成员为m\_sceneManager（场景管理器实例）。

TopToolbar定义了初始化（Initialize）和渲染（Render）方法。在初始化阶段Graphics将场景管理器实例以参数形式传递并赋值给TopToolbar的m\_sceneManager成员，因为TopToolbar在其提供的功能当中包括针对场景对象的处理和引用，所以场景管理器实例的引用是必需的。另外，由于工具栏提供的功能随着系统的开发迭代在不断变化，所以TopToolbar拥有的成员也是不定的。

使用ImGui创建窗口工具栏，首先要调用其BeginMainMenuBar函数，绘制一个基础的工具栏容器，然后调用BeginMenu在容器当中添加菜单按钮，菜单按钮也是一个容器，其中可以容纳任意数量的MenuItem。TopToolbar渲染的代码基础结构如下。

if (ImGui::BeginMainMenuBar())

{

if (ImGui::BeginMenu(MenuName))

{

if (ImGui::MenuItem(MenuItemName))

{

//执行自定义操作

…

}

}

ImGui::EndMenu();

}

ImGui::EndMainMenuBar();

在用户点击MenuItem时，则会调用注释中“执行自定义操作”部分的代码，实现自定义功能。

## 5.9 渲染算法实现

5.9.1 前向渲染实现

前向渲染（Forward Rendering）通过光栅化场景当中的每个几何物体来实现。在着色的过程中，通过迭代一个光源列表来决定几何体是如何被照亮的，这意味着每一个几何体需要考虑到每一个光源的影响。当然，我们可以通过忽略掉哪些被遮挡或者不再视锥体范围内的光源来进行优化。如果光源的范围是已知的，那么我们可以在渲染每个几何体之前对灯光进行视锥体剔除，物体剔除和光源剔除给这种技术带来了有限的优化，当使用前向渲染的时候，光源剔除一般不会使用。更常用的是直接限制影响物体的光源的个数。例如，一些图形引擎会对距离物体最近的光源执行逐像素的光照，对于稍远的光源执行逐顶点光照。DirectX和OpenGL提供的传统的固定渲染管线任何时候提供的动态光源个数限制为8个。即使是现代的图形硬件，在不出现明显帧率问题的情况下，前向渲染的动态光源个数也仅限于100个左右。

本系统对前向渲染的实现原理是在着色器当中对存储光源数据的列表进行遍历，结合每个光源的属性和物体表面信息进行光照计算，得出每个光源对于物体渲染的贡献，然后将基于每个光源渲染后得到的结果整合，得到最终的颜色值。

渲染引擎当中常用的光源类型有三种，分别为直射光、点光源和聚光灯，这三种光源基本可以满足常用的光照渲染需求。

直射光（Directional Light）：直射光只有方向，没有位置（位置没有意义），且光照强度固定，不会随着物体和光源距离的改变而改变。

点光源（Point Light）：点光源有位置但是没有方向，类似一个球体向四周发射光线，点光源的光照强度会随着物体和光源间距离的增大而衰减。点光源还拥有范围概念，即如果物体片段在点光源范围之外，则不会接受光照。

聚光灯（Spot Light）：聚光灯既有位置也有方向，类似一个锥体向着自身朝向发射光线，聚光灯的光照强度会随着物体和光源间距离的增大而衰减。除了在距离上的衰减，聚光灯在其锥体的底平面上也有光照强度的衰减，由圆心向圆周光照强度递减，以此来模拟真实世界中的聚光灯效果。聚光灯也有范围的概念，如果物体片段在聚光灯范围之外，则不会接受光照。

由于系统在着色器中遍历光源列表，需要元素为相同的类型，所以将以上三种光源的属性整合为一个结构类型，其在HLSL着色器当中的代码定义如下。

struct Light

{

float4 Color; //光源颜色

float3 Position; //光源世界空间位置

float Range; //光源范围

float3 Direction; //光源方向

float Intensity; //光源强度

uint Type; //光源类型

float SpotAngle; //聚光灯光线角度

float2 Padding; //显式补齐16字节

};

系统在着色器中使用结构缓冲（Structured Buffer）存储应用层传入的光源数据信息，结构缓冲可以解释为缓冲区的复合形式，它允许模板类型T是用户自定义的类型，即缓冲区存放的内容可以被解释为结构体数组，即结构缓冲可以在着色器当中作为列表来使用。其代码定义如下。

StructuredBuffer<Light> Lights : register(t0);

系统将存储光源列表的结构缓冲定义为Lights，结构缓冲使用的是纹理寄存器，所以其register参数为t#。

要使着色器当中的结构缓冲起作用，需要在C++代码端将光源列表数据绑定到渲染管线并传递到HLSL端。如结构缓冲资源结构实现部分介绍的，系统在应用层对结构缓冲进行了封装，其类型为StructuredBuffer。由于光源信息是场景当中全局公用的数据，所以对应的结构缓冲对象定义在图形渲染管理器（Graphics）当中，其代码实现如下。

StructuredBuffer<SB\_PS\_Light> m\_lightSB;

SB\_PS\_Ligh为系统在应用层定义的光源信息结构体类型，对应HLSL端的Light类型。Graphics负责对该结构缓冲进行初始化、绑定和其内部数据的更新。其中初始化和绑定操作在Graphics初始化阶段完成，数据的更新则在Graphics每一帧的渲染操作之前完成，在数据更新阶段，遍历从场景管理器（SceneManager）获取的光源列表，并将所有光源属性传递至结构缓冲当中，然后进行数据更新，其具体代码实现如下。

for (UINT i = 0; i < lights.size(); i++)

{

this->m\_lightSB.bufferData[i].Type = (UINT)lights[i]->GetComponent<Light>()->GetType();

this->m\_lightSB.bufferData[i].Color = lights[i]->GetComponent<Light>()->GetColor();

this->m\_lightSB.bufferData[i].Direction = lights[i]->GetComponent<Light>()->GetDirection();

this->m\_lightSB.bufferData[i].Position = lights[i]->GetComponent<Light>()->GetPosition();

this->m\_lightSB.bufferData[i].Intensity = lights[i]->GetComponent<Light>()->GetIntensity();

this->m\_lightSB.bufferData[i].Range = lights[i]->GetComponent<Light>()->GetRange();

this->m\_lightSB.bufferData[i].SpotAngle=lights[i]->GetComponent<Light>()->GetSpotAngle();

}

//更新结构缓冲

this->m\_lightSB.UpdateStructuredBuffer(this->m\_dxDeviceContext.Get());

其中lights即为SceneManager维护的光源列表，存储场景中所有拥有光源组件（Light）的实体，这里遍历每个实体并从光源组件获取每个属性然后赋值到结构缓冲存储的元素中，完成属性信息的修改之后，调用StructuredBuffer的UpdateStructuredBuffer函数将更改后的数据传递至GPU端。

完成了应用端到GPU端的结构缓冲数据传递之后，就可以在像素着色器当中使用结构缓冲来获取光源信息了。在HLSL当中，结构缓冲可以当作列表来使用，即允许对结构缓冲进行遍历操作，然后根据光源的类型进行特殊处理，代码的结构如下。

for (uint i = 0; i < NUM\_LIGHTS; i++)

{

//根据光源类型分别处理

switch (Lights[i].Type)

{

case DIRECTIONAL\_LIGHT:

{

…//处理直射光

}

break;

case POINT\_LIGHT:

{

…//处理点光源

}

break;

case SPOT\_LIGHT:

{

…//处理聚光灯

}

break;

}

}

宏NUM\_LIGHTS为系统定义的动态光源的最大数量，而宏DIRECTIONAL\_LIGHT、POINT\_LIGHT和SPOT\_LIGHT则分别对应Lights类型的Type成员的三种可取值，对应三种光源类型，值分别为0，1，2。

在像素着色器当中进行逐像素的光照计算，所需的光源信息主要是两个，世界空间当中（系统在世界空间进行光照计算，所以使用世界空间）像素到光源位置的方向和进行光照衰减计算后的光照颜色，即需要在上述的代码结构当中针对每种光照类型计算对应的两个数据。系统将这两个数据整合为一个结构体类型，其定义代码如下。

struct DirectIlluminationData

{

float4 color;

float3 direction;

};

所有物体渲染最终所需的直接光照信息即为DirectIlluminationData，针对每种光源类型进行处理后的结果会被存放在该结构体当中，供渲染计算使用。

1. 首先是针对直射光的处理，其代码实现如下。

DirectIlluminationData ProcessDirectionalLight(Light light,float3 pixelWorldPos)

{

DirectIlluminationData lightData;

lightData.color = light.Color\*light.Intensity/20;

lightData.direction = normalize(-light.Direction);

return lightData;

}

直射光的强度不会衰减，所以其颜色只和其本身的强度有关；像素到光源的方向向量也和其位置无关，为直射光光源方向的逆向。

1. 然后是针对点光源的处理，其代码实现如下。

DirectIlluminationData ProcessPointlLight(Light light, float3 pixelWorldPos)

{

DirectIlluminationData lightData;

lightData.direction = normalize(light.Position - pixelWorldPos);

float lightDistance = distance(light.Position, pixelWorldPos);

float distanceAtt = 1 / (1 + lightDistance \* lightDistance);

float rangeAtt = smoothstep(light.Range, 0, lightDistance);

lightData.color = light.Color \* 10000 \* distanceAtt \* rangeAtt \* light.Intensity / 20;

return lightData;

}

像素到点光源的方向向量为点光源位置坐标和像素的位置坐标做差；点光源的最终颜色需要经过衰减处理获取，点光源的衰减计算可以用一个光子组成的球体来理解，由于光子的移动速度都相同，所以在同一个半径上的球面上的所有光子数量是相同的，这就导致半径越大，球体表面积越大，球面上的光子密度越低，所以光照强度和半径的平方成反比。除了强度的衰减，点光源还需要进行范围的判断，如果像素不在点光源的范围内，则其光照贡献为0，这里为了防止在范围极限附近出现光照突然消失的情况，对其使用了smoothstep函数进行了平滑处理，使其光照强度在像素超出范围时逐渐降为0。最后将经过衰减和范围计算的光照颜色输出。

1. 最后是针对聚光灯的处理，其代码实现如下。

DirectIlluminationData ProcessSpotLight(Light light, float3 pixelWorldPos)

{

DirectIlluminationData lightData;

lightData.direction = normalize(light.Position - pixelWorldPos);

float lightDistance = distance(light.Position, pixelWorldPos);

float coneAtt = pow(max(dot(-lightData.direction, normalize(light.Direction)), 0.0f),

180-light.SpotAngle);

float distanceAtt = 1 / (1 + lightDistance \* lightDistance);

float rangeAtt = smoothstep(light.Range, 0, lightDistance);

lightData.color = light.Color \* 10000 \* distanceAtt \* rangeAtt \* coneAtt \* light.Intensity / 20;

return lightData;

}

像素到聚光灯的方向向量为聚光灯位置坐标和像素的位置坐标做差；聚光灯的距离衰减计算以及范围计算和点光源相似，除此之外，聚光灯在其锥体照射区域也有光照强度的衰减，在同等照射距离下，越靠近照射中心，亮度越强。同样随着距离的增大，光照强度逐渐减弱，其衰减计算原理见公式（5.1）。

(5.1)

其中L表示像素到聚光灯的方向向量，d表示聚光灯本身指向的方向向量，指数spot表示聚光灯的光线汇聚程度，系统使用180度和聚光灯的光线发射角度做差来定义，即光线发射角越大，光线汇聚程度越低，光照的强度越低；光线发射角越小，光线的汇聚程度越高，光照的强度越高。聚光灯的处理最后将经过距离衰减、角度衰减和范围计算后的光照颜色输出。

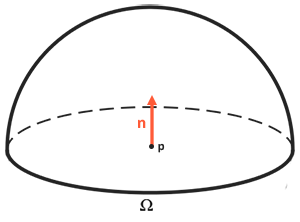
以上为前向渲染当中对于光源信息的处理流程，流程的最后获取到系统定义的DirectIlluminationData结构体信息，它包含光照计算所需的关键信息，之后所有的有关光照模型的渲染算法的实现都可以在此框架当中实现。

5.9.2 基于物理的渲染实现

基于物理的渲染（Physically Based Rendering，PBR）的具体实现方案并不固定，但所有的方案基于都是基于反射方程（Reflectance Equation）的。反射方程见公式（5.2）。

(5.2)

反射方程的核心含义是计算所有射向空间种一点的光线反射至观察点的总辐照度。渲染方程中的为入射点的信息，等式的右侧为对的积分式，它表示的是对于所有从点法线正向半球入射的光线进行积分计算，如图所示。



由于实际计算当中无法真正模拟所有的入射光线，所以只能采用离散的方式进行处理，即认为每个光源对于点都有一束入射光线，将这些光线信息代入到方程当中，并将计算得到的反射结果做和，得到近似于对所有光线积分的效果。

渲染方程当中的项即为入射光线信息，项为入射至点的光照辐射率，则用来计算光线入射至平面时由于入射角导致的强度衰减，光线越是斜着照射平面，强度越低，越接近垂直照射，强度越高。项则计算入射光线射入点后反射至观察点的比例，也就是PBR实现当中的重点——BxDF。BxDF是对于BRDF（Bidirectional Reflectance Distribution Function）、BSDF（Bidirectional Scattering Distribution Function）和BTDF（Bidirectional Transmittance Distribution Function）等双向分布函数的一般表示形式。

本系统对于PBR的实现基于目前主流的Cook-Torrance BRDF，其方程见公式（5.3）。

(5.3)

根据光和非光学平坦表面的交互原理，当一束光入射到物体表面，由于两种介质之间折射率的快速变化，光线会产生反射（Reflection）和折射（Refraction）两种现象。反射即镜面反射，直接从物体表面反射出去，而折射部分则会进入物体内部，一部分可能会被吸收，另一部分则在物体内部继续反射，最后从物体表面射出或者能量被耗尽，从物体内部射出的部分为散射（Scattering）。当实践的观察尺度大于散射的距离，则散射部分被视为漫反射（Diffuse）；当实践的观察尺度小于散射的距离，则散射部分被视为次表面散射（Subsurface Scattering）。本系统不考虑多次散射并将散射部分视为漫反射。

Cook-Torrance BRDF公式中，项为漫反射部分的比例，项为镜面反射部分的比例，由于PBR遵循能量守恒原则，所以二者之和为1。为漫反射光线的部分，其计算方式见公式（5.4）。

(5.4)

公式中表示物体表面的基础颜色，即反照率（Albedo）。除以π是为了规格化漫反射光，为之后的BRDF积分做准备。

为镜面反射光线的部分，其计算则相对复杂，计算方式见公式（5.5）

(5.5)

镜面反射部分算式的分子由三个函数（D、F、G）和标准化因子构成。三个函数都用于模拟物体表面对光线的反射属性。

D：法线分布函数（Normal Distribution Function），负责统计在表面粗糙度的作用下，法线朝向和半角向量一致的微平面所占的比例。

F：菲涅尔方程（Fresnel Equation），负责计算光线在不同入射角的情况下反射的光线所占的比例。

G：几何函数（Geometry Function），负责计算物体表面的微平面自成阴影的情况。当物体表面比较粗糙时，一些微平面可能会挡住其它微平面反射出的光线，导致能量减少，降低反射效果。

镜面反射部分算式的分母是校正因子（Correctionfactor），作为微观几何的局部空间和整个宏观表面的局部空间之间变换的微平面量的校正。

在PBR的实现中，D、F、G三个函数的使用是不固定的，可以根据需求选择甚至自定义每个函数的实现，本系统使用的是Unreal Engine 4中使用的函数，其中D项使用Trowbridge-Reitz GGX，F项使用Fresnel-Schlick近似法(Approximation)，而G项使用Smith's Schlick-GGX。

## 5.10 本章小节

本章详细介绍了以系统设计方案为依据而进行的系统具体实现，从顶层至底层将抽象的模块转化为具象的代码逻辑执行。首先介绍了系统的顶层架构的实现方案，然后介绍如何以实体-组件模式对系统的基本操作单元进行实现，接下来阐述的是系统各个核心模块、资源结构的实现以及相关的核心代码的展示，最后介绍了基于本系统进行的渲染算法实践。

第6章 系统测试

## 6.1 系统环境

6.1.1 系统调试阶段测试效果

调试阶段运行环境参数如表6.1所示。

表6.1 系统调试阶段运行环境参数

|  |  |  |
| --- | --- | --- |
| **表序号** | **参数** | **值** |
| 1 | CPU | Intel(R) Core(TM) i5-7300HQ CPU @2.50HZ |
| 2 | 显卡 | NVIDIA GeForce GTX 1050 Ti (4096MB) |
| 3 | 内存 | 24GB |
| 4 | 调试平台 | Windows x86 |
| 5 | 运行分辨率 | 1800x900 |

在不渲染任何实体的情况下，系统运行情况如表6.2所示。

表6.2 系统调试阶段运行状况

|  |  |  |
| --- | --- | --- |
| **表序号** | **参数** | **值** |
| 1 | 进程内存占用 | 24MB |
| 2 | FPS | 480~540 |
| 3 | CPU使用率 | 20%~25% |

6.1.2 系统发布阶段测试方案

系统程序可以打包为32位可执行程序，打包后将资源文件夹Assets放入到文件夹根目录下，否则系统将无法正确加载资源文件。系统发布后的可执行文件名为“DirectX Raster Renderer.exe”。要在系统中使用自定义资源文件，用户需要将根据文件类型将其放入到Assets文件夹下的对应目录下。各类型文件对应目录如下。

模型网格文件：\Assets\Custom\Meshes

纹理文件：\Assets\Custom\Audios

音频文件：\Assets\Custom\Audios

字体文件：\Assets\Custom\Fonts

完成准备工作后，可以开始对系统功能进行测试。测试的功能如表6.3所示。

表6.3 系统调试阶段运行状况

|  |  |  |
| --- | --- | --- |
| **表序号** | **测试功能** | **测试结果** |
| 1 | 加载模型文件 | 功能正常 |
| 2 | 为实体添加、删除组件 | 功能正常 |
| 3 | 调整实体位置、旋转、缩放 | 功能正常 |
| 4 | 调节材质参数 | 功能正常 |
| 5 | 替换材质资源（纹理、着色器） | 功能正常 |
| 6 | 替换网格资源 | 功能正常 |
| 7 | 开启第一人称游览模式 | 功能正常 |

## 6.2 系统界面

基于DirectX 11的光栅化渲染器的系统界面由主渲染区域、场景对象结构面板（Hierarchy）、细节信息面板（Inspector）和窗口工具栏（TopToolBar）组成，各个用户界面元素可以拖动。系统界面整体结构如图6.1所示。

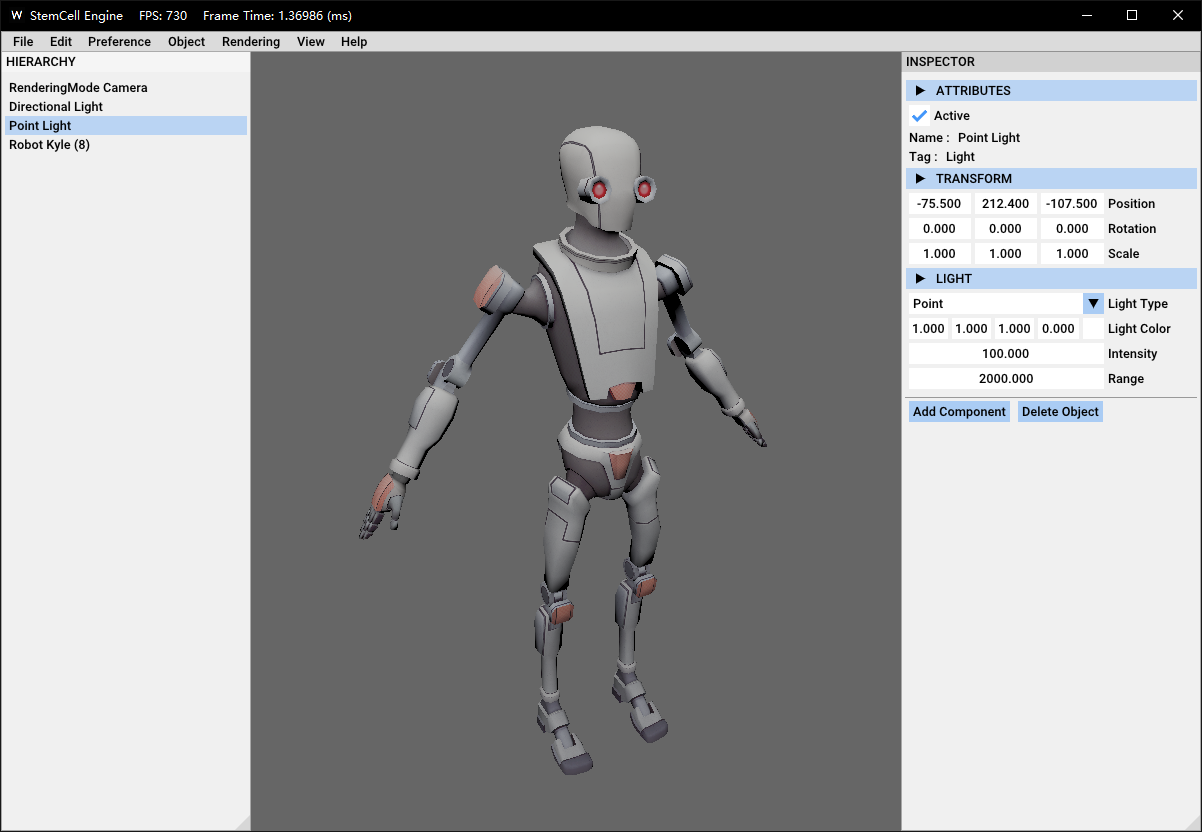
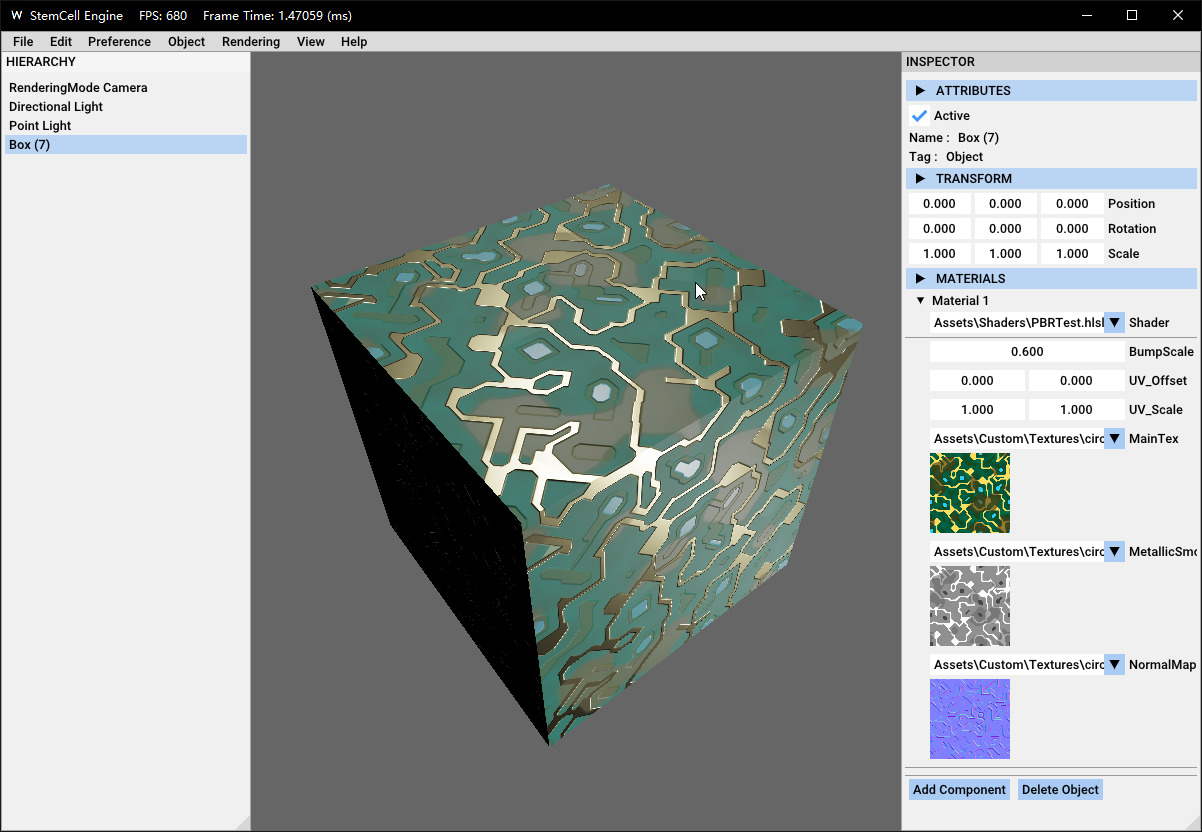


图6.1 系统界面整体结构图

界面中间为系统的场景渲染区域。左侧“HIERARCHY”窗口即场景对象结构面板，显示当前场景当中的所有实体对象，供用户选择。右侧“INSPECTOR”窗口即细节信息面板，显示用户在场景对象结构面板当中选定实体的具体信息，包括实体拥有的每个组件拥有的具体属性，且用户可以直接进行调节来自定义参数。窗口上方即为窗口工具栏，包含许多菜单按钮来提供更多的功能。

## 6.3 PBR效果测试

系统内置了基于PBR算法的着色器，在调节材质参数时选用对应的着色器文件，实体会使用PBR进行渲染，着色器对应的参数也会显示出来供用户调节。PBR着色器正确执行需要外部提供相应纹理资源并设置合理的参数数值，在材质参数当中选定符合要求的纹理即可，其渲染效果如图6.2所示。

图6.2 PBR效果图

6.4 本章小节

本章对系统实现的最终情况进行了测试，介绍了在调试和发布阶段程序运行的效率参数。对系统的核心功能进行交互测试，确保程序运转良好，符合预期。最后，展示了渲染器程序的用户界面样式和模型渲染效果。

第7章 总结和展望

7.1 工作总结

论文完成的主要工作如下：

1. 使用C++语言、基于面向对象的编程思想，结合DirectX 11应用程序图形接口，设计并实现了一个光栅化渲染器程序。
2. 引入了Dear ImGui图形用户界面库，为程序提供了审美优良、功能丰富且交互体验舒适的用户界面。系统的图形用户界面提供的功能非常贴合渲染器的需求，如数值调节、颜色选取、纹理显示等。
3. 结合DirectXTK、Assimp等开发库，为程序提供了加载用户自定义的纹理、模型等文件资源的功能。系统的资源加载功能拥有良好的兼容性，可以应用于绝大多数渲染相关的资源文件格式。
4. 利用HLSL编写着色器，基于DirectX 11渲染管线研究前向渲染、基于物理的渲染等渲染算法在系统中的实现，并在渲染器当中集成了PBR着色器。

7.2 未来展望

渲染技术本身涉及的知识就十分庞杂，而设计并实现一个渲染器需要的不仅是对于渲染技术的了解，还需要良好软件架构思想，强大的编码能力，甚至是对于数学、美术等基础学科知识的良好掌握，所以，要开发一个成熟的、高性能的、功能丰富的渲染器是十分困难的。由于开发时间有限，技术力尚不成熟，所以系统目前还有很多预想功能没有实现，部分功能也有不足，主要包括如下方面。

1. 场景当中的实体对象无法通过拾取的方式选择，只能通过在图形用户界面中选取来实现。
2. 未能实现工程文件系统，没有提供项目文件处理的相关功能和用户界面。相关的项目资源目前只能在界面提供的文件列表中进行简单选取，不能浏览所有的工程文件。
3. 未能实现控制台系统。作为一个渲染器程序，在使用过程中出现运行、设置错误是很常见的，但是目前没有相关的界面能够提示用户程序哪里出现了问题。

参考文献

1. 逆水. Corona渲染器发展史及3大特点[EB/OL]. https://www.jianyi80.com/27598.html
2. 倪嘉声. 国产知名渲染器D5 Render更新至1.6版 支持RTX显卡加速，AI降噪[J/OL]. https://www.expreview.com/74414.html. 2020-5-27
3. Tomas Akenine-Möller,Eric Haines,Naty Hoffman等. Real-Time Rendering[M]. Fourth edition. U.S:AK Peters/CRC Press,2018:12-13.
4. Varcholik,Paul. Real-Time3D Rendering with DirectX and HLSL[M]. U.S:

Addison-Wesley Professional,2014:13-20.

1. Frank Luna. Introduction to 3D Game Programming with DirectX 11[M]. US:Mercury Learning & Information,2012:19-20
2. dear imgui 1.6.2 documentation[EB/OL]. https://imgui-test.readthedocs.io/en/latest/. 2018
3. Learn OpenGL[EB/OL]. https://learnopengl.com/. 2014-6
4. Matt Pharr,Wenzel Jakob,Greg Humphreys. Physically Based Rendering[M]. Third Edition. US:Morgan Kaufmann,2018:539-583
5. 杨睿涵. 游戏开发中的ECS 架构概述[EB/OL]https://zhuanlan.zhihu.com/p/30538626. 2018-5-1
6. X\_Jun. DirectX11 With Windows SDK--01 DirectX11初始化[EB/OL]. https://

www.cnblogs.com/X-Jun/p/9069608.html. 2018-5-21

1. Frank Luna. DirectX 12 3D游戏开发实战[M]. 北京: 人民邮电出版社,2019: 83-84
2. MSDN[EB/OL]. https://docs.microsoft.com/en-us/windows/win32/api/winbase/

nf-winbase-winmain. 2018-5-12

1. Stanley B. Lippman,Josée Lajoie,Barbara E. Moo. C++ Primer 中文版[M]. 第五版. 北京:电子工业出版社, 2013:980-1054
2. Zero. C++ 可变参数模板[EB/OL]https://zhuanlan.zhihu.com/p/337932819. 2020-12-18
3. MSDN[EB/OL]. https://docs.microsoft.com/en-us/windows/win32/api/d3d11/

nf-d3d11-d3d11createdeviceandswapchain. 2018-5-12

致 谢

光阴似箭，日月如梭，转眼间四年的大学时光已经快要画上句号。回望这四年间，我从刚刚踏入校园的青涩少年，变为即将离开校园的成熟青年，也拥有了这一生都难忘的回忆。在东北大学的这段时光，让我结识了良师益友，让我获得了知识，找到了人生的目标。在此将要离别之际，我心中虽然怀着对未来的憧憬，但更多的是对关心、帮助我的人的感激之情。

感谢此篇毕业论文的指导老师马连博教授，在他的悉心教导、耐心督促下，我才能够顺利地完成毕业设计的制作和毕业论文的撰写。祝老师身体健康，工作顺利。

感谢在大学期间教授过我的所有课业老师，他们毫不吝啬地将知识传授给我，让我能够快速地成长，成为一个能够独立思考并解决问题的人。

感谢我的同学、室友和朋友们，遇见他们是我的幸运。是他们为我的生活增添了别样的色彩，让我的生活充满乐趣。

感谢养育我的父母，是他们的努力工作，辛勤劳动，才保证我能够有机会在大学当中学习和成长。虽然并不在身边，但是他们的关怀、鼓励、支持给了我前进的动力。

最后，感谢审阅论文的各位老师，感谢你们在百忙之中抽出时间对我进行指导。