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基于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和NPR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 chosen based on the game engine in game development, because the game engine is the most needed for the game development in the technology package, developers often only need to call the relevant interface rich function can be realized, and do not need to write from the bottom 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 rich rendering algorithm and provides a simple interface for developers call, it greatly improves the development efficiency, but also allows developers to understand the rendering technology to realize the underlying mechanism, 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 and NPR algorithms commonly used in game development are integrated in the program. By placing the model, texture, and custom shader resources in the program file directory, the user can run the program to load and use these resources to achieve the rendering effect and test.

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 课题研究背景于意义 1](#_Toc24502)

[1.2 国内外研究现状 2](#_Toc2550)

[1.2.1 国内研究现状 2](#_Toc8998)

[1.2.2 国外研究现状 2](#_Toc24674)

[1.3 论文研究内容 3](#_Toc32336)

[1.4 论文的组织结构 3](#_Toc32336)

[第2章　相关技术 5](#_Toc28980)

[2.1 DirectX 11 5](#_Toc32544)

[2.1.1 HLSL 5](#_Toc11079)

[2.1.2 DirectXTK 5](#_Toc18429)

[2.2 基于物理的渲染 5](#_Toc27733)

[2.3 非真实感渲染 6](#_Toc21972)

[2.4 Dear ImGui 6](#_Toc12351)

[2.5 Assimp 7](#_Toc31185)

[第3章 需求分析 8](#_Toc18888)

[3.1 功能需求 8](#_Toc23433)

[3.2 用户需求 8](#_Toc16266)

[3.3 性能需求 8](#_Toc15084)

[第4章 系统设计 13](#_Toc26403)

[4.1 整体架构设计 13](#_Toc24586)

[4.2 实体组件架构设计 13](#_Toc24586)

[4.3 功能模块设计 13](#_Toc20993)

[4.3 数据结构设计 14](#_Toc17031)

[4.4 工作时序图 14](#_Toc17031)

[4.5 主体类设计 14](#_Toc17031)

[4.6 用户界面设计 14](#_Toc17031)

第5章 系统实现

[5.1 功能模块实现 13](#_Toc24586)

[5.2 数据结构实现 13](#_Toc20993)

[5.3 渲染算法实现 14](#_Toc17031)

[5.4 工作时序图 14](#_Toc17031)

[5.5 用户界面设计 14](#_Toc17031)

第6章 系统测试

[6.1 运行环境 13](#_Toc24586)

[6.2 程序界面 13](#_Toc20993)

[6.3 PBR效果测试 13](#_Toc20993)

[6.4 NPR效果测试 13](#_Toc20993)

第7章 总结和展望

[参考文献 15](#_Toc18532)

[致 谢 16](#_Toc28784)

第1章　绪 论

## 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。因此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。

## 国内外现状

## 1.4 论文研究内容

## 1.5 论文的组织结构

第1章，绪论。

第2章，相关技术。

第3章，需求分析。

第4章，系统设计。

第5章，系统实现。

第6章，系统测试。

第7章，总结与展望。

第2章　相关技术

## 2.1 DirectX 11

DirectX 11是目前主流的图形API——DirectX的第11个大版本。DirectX，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。【DX11游戏编程入门(中文版)】DirectX 11是DirectX 10的改良，只对性能和扩展性进行优化，主要改进在提供Shader Model 5.0，加入对MSAA的直接采样控制。大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，输入设备以及所有幅改进多线程性能，提供三个独立的接口。进一步提高纹理压缩，镶嵌（tessellation）处理的性能，并加入Compute Shader（DirectCompute）来支持GPGPU，提供新版HLSL语言。【[DirectX - 维基百科，自由的百科全书 (wikipedia.org)](https://zh.wikipedia.org/wiki/DirectX)】

## 2.2 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、3D场景物体以及光源等场景要素来产生或者渲染一副2D的图像。

广义渲染管线

DirectX 11渲染管线

## 2.3 Dear ImGui

## 2.4 Assimp

## 2.5 PBR

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：

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。

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。

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。

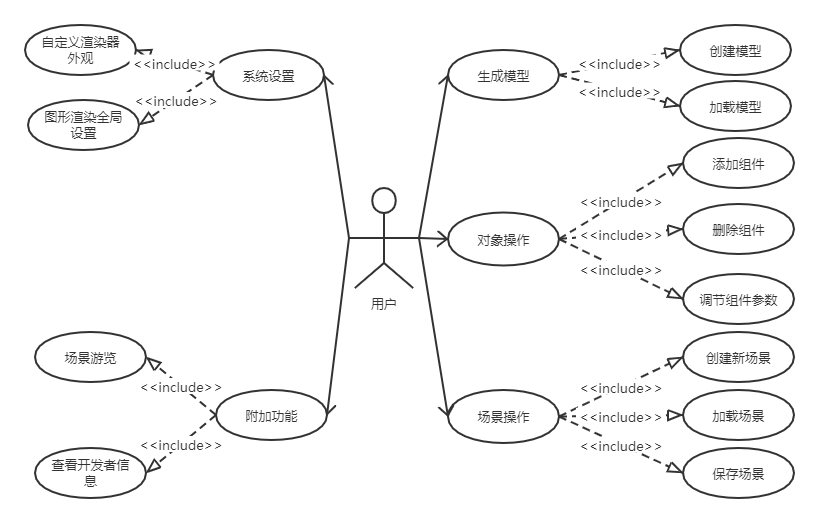
场景对象的组件处理功能。在生成基本对象之后，可以通过为其添加组件使其拥有新的功能，如添加材质组件可以提供自定义材质属性的功能。

## 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。

系统用例图如图所示。



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

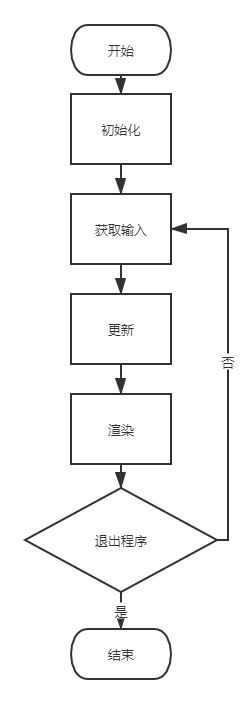
1. 系统设计

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

4.11 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，此逻辑循环可以用图表示：

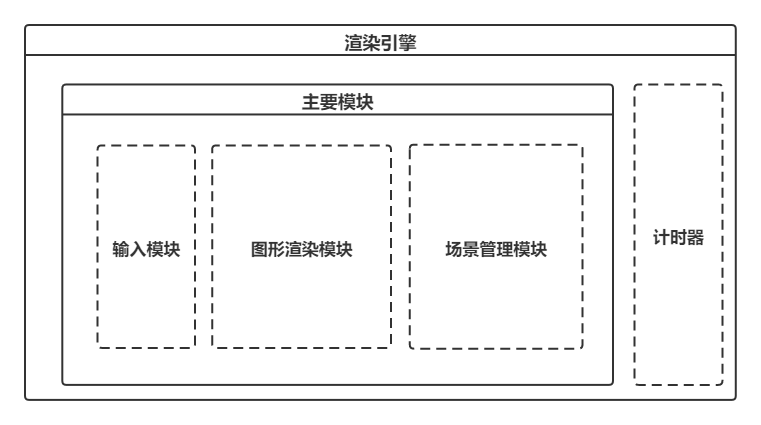


渲染器程序顶层逻辑

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。

4.12 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所示。



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

4.21 实体-组件模型概述

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，那么必然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。而以实体-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-组件模型和著名的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。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。

4.22 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。

4.22 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。

4.23 实体-组件类设计

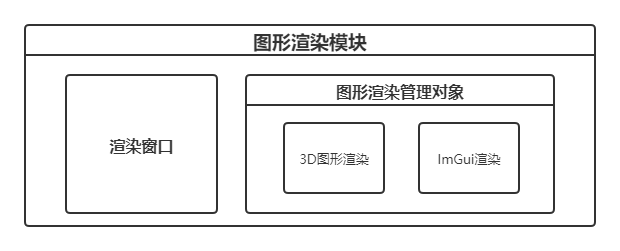
实体-组件模型的UML类图如图所示。

## UML类图学习

## 4.2 功能模块设计

4.21 图形渲染模块

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所示。



图

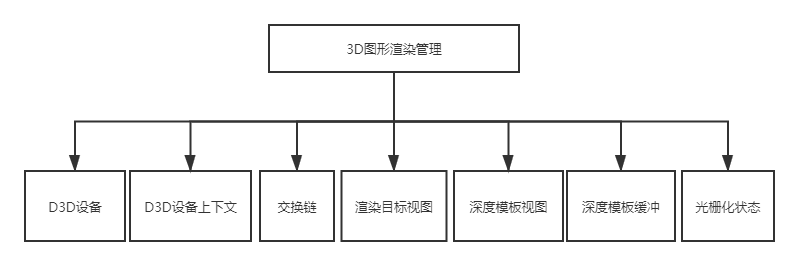
1. 渲染窗口

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。

1. 图形渲染管理

图形渲染管理对象（Graphics）是全局图形渲染状态的管理者，负责在顶层处理DirectX 11 3D图形的渲染和用户界面的渲染。

图形渲染管理对象在DirectX 3D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。全局通用的资源对象如图所示。



图

实现DirectX 11的3D图形渲染，首先要对Direct3D进行初始化，Direct3D初始化阶段首先需要创建D3D设备和D3D设备上下文。

D3D设备（ID3D11Device）通常代表一个显示适配器（即显卡），它最主要的功能是用于创建各种所需资源，最常用的资源有：资源类（ID3D11Resource, 包含纹理和缓冲区），视图类以及着色器。此外，D3D设备还能够用于检测系统环境对功能的支持情况。D3D设备上下文（ID3D11DeviceContext）可以看做是一个渲染管线。通常我们在创建D3D设备的同时也会附赠一个立即设备上下文（Immediate Context）。一个D3D设备仅对应一个D3D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。渲染管线主要负责渲染和计算工作，它需要绑定来自与它关联的D3D设备所创建的各种资源、视图和着色器才能正常运转，除此之外，它还能够负责对资源的直接读写操作。【[DirectX11 With Windows SDK--01 DirectX11初始化 - X\_Jun - 博客园 (cnblogs.com)](https://www.cnblogs.com/X-Jun/p/9069608.html" \l "_label6)】

除D3D设备和D3D设备上下文之外，还需要创建交换链（SwapChain）。交换链是一系列虚拟帧缓冲区，由显卡和图形接口用来实现稳定帧速率等功能。每个交换链至少有两个缓冲区，都可以称作后备缓冲区（BackBuffer），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。

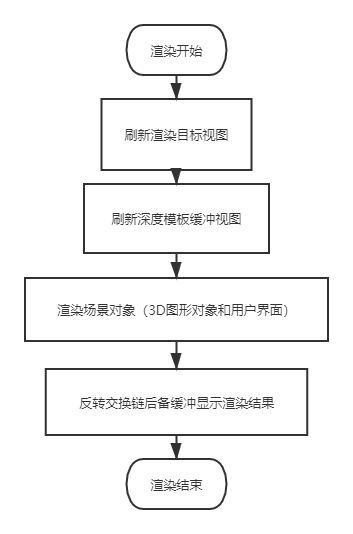
在创建了交换链之后，图形渲染管理对象将交换链的后备缓冲设置为渲染目标，然后根据需求创建深度模板缓冲和深度缓冲视图，为之后的深度模板测试做准备。

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（例如，从描述物体表面样貌的纹理或者存有3D场景中几何体位置信息的缓冲区中读取数据）和写（例如，向后台缓冲区或深度/模板缓冲区写入数据）两种操作。在发出绘制命令之前，需要将与本次绘制调用（draw call）相关的资源绑定到渲染流水线上。部分资源可能在每次绘制调用时都会有所变化，所以我们也就要每次按需求更新绑定。但是GPU资源并非直接与渲染流水线相绑定，而是要通过一种名为描述符（即视图）的对象来对它间接引用，我们可以把描述符视为一种对送往GPU的资源进行描述的轻量级结构。从本质上来讲，它实际上即为一个中间层；若指定了资源描述符，GPU将既能获得实际的资源数据，也能了解到资源的必要信息。【DirectX 12 3D游戏开发实战】

最后渲染管理对象进行视口设置。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。

除了维护全局通用的DirectX资源对象，渲染管理对象的另一个功能就是实现顶层的渲染逻辑。顶层的渲染逻辑可以由下图表示。



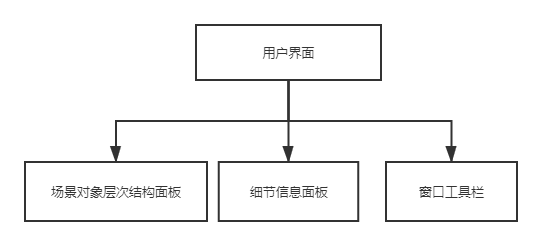
顶层渲染逻辑

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。

1. 用户界面

用户界面本应该作为图形渲染模块的另一个主要组成部分独立存在，而不是作为图形渲染管理对象的一个子成分，但是其实现引用了Dear ImGui库，其运行逻辑和DirectX的图形渲染的逻辑并不相同，但是又依附于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对象中剥离，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的对象中的一员，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）作为图形渲染管理对象的成员，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），如图所示。



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。

4.23 场景管理模块

场景管理模块由场景管理对象（SceneManager）实现。场景管理对象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（Initialize）和更新方法（Update）。场景管理模块提供接口实现添加和销毁实体的功能。

4.24 输入控制模块

1. 鼠标输入
2. 键盘输入

4.25 计时器模块

计时器

4.26 资源处理模块

4.27 工具模块

## 4.3 资源结构设计

4.31 材质

4.32 着色器

4.33 着色器资源参数

4.34 顶点缓冲和索引缓冲

4.35 常量缓冲

4.36 纹理

4.37 采样器状态

4.38 结构缓冲

## 4.4 工作时序图

第5章 系统实现

## 5.1 功能模块实现

5.11 图形渲染模块实现

5.12 逻辑处理模块实现

5.13 用户界面模块实现

5.14 输入控制模块

5.141 鼠标输入控制

5.142 键盘输入控制

5.15 资源处理模块

5.151 场景资源处理

5.152 模型资源处理

5.153 着色器资源处理

5.154 纹理资源处理

5.16 工具模块实现

5.161 计时器

5.162 错误侦测

5.163 类型转换

5.164 资源加载

## 5.2 数据结构实现

5.21 材质

5.22 着色器

5.23 着色器资源参数

5.24 顶点缓冲和索引缓冲

5.25 常量缓冲

5.26 纹理

5.27 采样器状态

5.28 结构缓冲

## 5.3 渲染算法实现

5.31 前向渲染实现

5.32 光照系统实现

5.33 PBR实现

5.34 NPR实现

5.35 阴影效果实现

5.36 天空盒实现

第6章 系统测试

6.1 运行环境

6.2 程序界面

6.3 PBR效果测试

6.4 NPR效果测试

第7章 总结和展望

参考文献

致 谢