编 号

本科生毕业设计（论文）

题目：基于 DirectX 的全局光照算法

SVO 的框架软件开发

数字媒体 学 院 数字媒体技术 专 业

学　　号 1030513408

学生姓名 华清沁

指导教师 张　军　副教授

二〇一七年六月

设计总说明

实时渲染指的是将画面以可交互的帧率在屏幕上进行渲染，它的本质是图形数据在实时的不断进行计算和输出.如何在保持画面帧率的同时得到照片级的画面质量一直是实时渲染领域最重要的研究方向.近年来，由于硬件的高速发展，进行图形运算的显卡的性能大大提高，高质量的实时渲染技术逐渐进入人们的视野.国外的次世代游戏使用优秀的游戏引擎，这些游戏引擎集成了前沿的实时渲染技术，使得制作出的游戏可以达到逼真的画面效果，而国内由于传统图形学热度不高，自研引擎在高质量的实时渲染模块表现平平.

评价一款实时渲染引擎有两个不可忽视的指标：

1. 渲染速度.渲染速度是游戏引擎的核心，而实时渲染技术更是游戏中渲染速度的核心和基础，在游戏场景中，渲染速度必须达到可以交互的帧率.
2. 渲染质量.在游戏中渲染高质量画面必须应用到前沿算法和高级渲染技术，如何设计实时全局光照算法就是这些高级渲染技术的研究热点和方向.

由于玩家对于画面的质量要求越来越高，实时全局光照算法在实时渲染引擎的重要性也越来越突出，传统的方法无法实现动态的光源变化，近年来由于动态光源的需求越来越大，实时全局光照算法也逐渐进入了游戏引擎进行应用，但是如何平衡速度与质量，在实时帧率上获得更好的全局光照效果仍是学术界致力于研究的方向.

本课题是基于一种实时全局光照算法，SVO 算法的实时渲染框架的实现，本框架可以充分利用 GPU 在图形处理方面的优势，集成包括以全局光照算法为核心的高级渲染算法.这些高级渲染算法不止是生成照片级画面的基础，也能够加速画面的渲染，是渲染引擎必不可少的核心.

本课题主要完成了以下的工作：

1. 设计并实现一个游戏引擎的渲染系统，此框架基于 DirectX11，可以进行实时的全局光照渲染,集成了材质读取、模型解析、shader 编译、摄像机系统、I/O 接口等基本渲染功能，并且可以在此框架的基础上拓展实现动画系统、粒子系统等引擎模块.
2. 研究并实现了基于 GPU 的体素化，此体素化方法可以实时地进行动态场景的体素化.
3. 研究并实现了基于体素的实时全局光照算法，此算法应用了光线追踪的原理，达到了可以交互的帧率，同时可以拓展到包括 Unity 的主流游戏引擎中进行应用.
4. 框架集成了包括基于物理的渲染(Physically-based Rendering)、可交互材质灯光、卡通渲染(Cartoon Shading)等渲染技术和渲染效果.

关键词：全局光照；实时渲染；渲染引擎；光传输模拟

General description of design

Real-time rendering refers to the image which rendering on the screen with the interactive frame rate, its essence is the real-time continuous calculation for graphics data. How to keep the image rendering in fast frame rate with the photo-level quality has been the most important research direction in the field of real-time rendering. In recent years, due to the rapid development of hardware, the performance of the graphics card greatly improved, high-quality real-time rendering technology gradually come into our lives. The foreign games of next generation always use excellent game engine. These game engines integrate the forefront of real-time rendering technology, making the game achieve realistic picture effect. But due to the low attention in domestic graphics technology, self-research engine in the high-quality real-time rendering don't have good performance.

We use two factors to evaluate a real-time rendering engine:

1. Rendering speed. Rendering speed is the core of the game engine, real-time rendering technology is the core of the game speed and the basis in rendering the game scene, the speed must reach the interactive frame rate.
2. Rendering quality. Due to the need of photo-level picture, it is necessary to apply advanced rendering algorithm and technology in game developing. The most difficult problem in the research is how to design real-time global illumination algorithm.

As the quality requirements is getting higher and higher, the real-time global illumination is becoming more and more important in the real-time rendering engine. Due to the traditional pre-calculating method can not achieve dynamic light source, real-time global lighting has gradually entered the game Engine, but how to balance the speed and quality is still a difficult problem.

This topic mainly completed the following work:

1. Design and implement a rendering system, which is based on DirectX11. It can do real-time global lighting, load material, analysis model, compile shader and other basic rendering feature, and can be developed to integrate animation systems, particle systems and other engine modules.
2. Study and implement GPU-based voxelization, which can be used to realize the voxelization of dynamic scenes in real time.
3. Study and realize the real-time global illumination algorithm based on voxel. This algorithm applies the principle of ray tracing to reach the frame rate and can be extended to the mainstream game engine like Unity.
4. Integrates the rendering techniques and rendering effects including physically-based rendering, interactive material and light and cartoon shading to the rendering frame-work.

Keywords: Global Illumination; Real-Time Rendering; Rendering Engine; Light Transport Simulation

目 录

第 1 章 绪论

1.1 研究背景与意义

数字游戏产业与影视动漫产业是本世纪以来我国大力发展的战略性产业，是信息科学人才与艺术人才交叉结合而成的创意产业.我国近年来数字游戏产业与影视动漫产业的全球市场规模为 8000 亿元人民币左右，据《2016 年中国游戏产业报告》[1]，2016 年中国游戏市场实际销售收入达到 1655.7 亿元人民币，游戏用户规模达到了 5.66 亿人次，可见我国的数字游戏文化产业具有重大的潜力.

相较于游戏行业而言，我国的动漫影视行业规模较小.据《2016 中国数字创意产业发展报告》统计，2016 年进入中国城市主流院线和影院上映的动画电影 62 部，票房为 70 亿元.其中，进口动画电影 23 部，票房 46.62 亿元，国产动画电影 39 部，票房 23.43 亿元，国外动画电影能同时在口碑上与收益上胜过国产动画电影.据赛迪咨询统计，全球数字动漫产业 2008 年总产值达到 7000 亿美元.由此可见，我国数字动画产业与美国、日本等国家相比，规模仍然较小，技术相对落后，且效益也相对较低.同时，数字动漫影视产业对一个国家来说不仅在经济发展上有重要意义，对各国文化的宣传、国家形象的塑造也有重要的意义.因此，大力发展动漫影视产业，在内容、技术等各方面加强数字动画影视制作技术符合我国的战略需求.

渲染技术在动漫和游戏产业中都发挥着至关重要的作用，代表了动漫和游戏场景的表现力，是动漫和游戏表现形式的核心和基础.三维渲染是在虚拟场景中对几何模型进行材质、光源、摄像机等属性进行设置，将这些属性依据艺术设计人员想要得到的效果进行分别计算，得到最终显示的图像的过程.

实时渲染技术是 3D 计算机图形学中的研究课题之一，也是最重要的研究方向之一，在游戏、仿真模拟、动画、电影等方面都有着广泛的应用.

在游戏产业中，游戏需要平衡画面质量与渲染速度的关系，为了流畅的游戏体验，不低于 30 帧的交互帧率是必须的.然而随着硬件的发展，玩家越来越倾向于照片级的游戏画面，这就必须使用高级实时渲染技术，在确保帧率的情况下得到更好的渲染画面.

在动画电影产业中，为了实现真实的光影效果，渲染一帧高质量的画面通常需要一个小时以上的时间，大大降低了工作效率，如何借助实时渲染技术提高渲染速度，缩短渲染周期，是动画电影行业关注的问题.

在 VR 产业中，渲染技术更是必须的.在沉浸式体验中，帧率与画面的平衡更加重要，要在 VR 体验过程中得到良好的沉浸式体验，需要至少 60 帧的渲染速度，而在 VR 显示器上画面也需要更加的真实，如何能够达到高速的渲染速度和照片级的画面质量是实时渲染技术在 VR 行业中急需解决的.

渲染技术尤其是实时渲染技术，在众多的应用领域特别是数字游戏影视产业中有着广泛的应用前景，我国要在数字文化领域超过传统数字媒体技术强国，对渲染技术的关注和发展是迫切并且必不可少的.

1.2 国内外研究现状

1.2.1 国内研究现状

国内游戏行业的自研商用引擎非常少，最著名的也是唯一的游戏引擎是 Cocos 引擎，专注于移动端的 2D 游戏开发，也有向 3D 发展的趋势.Cocos 引擎开源但是工具链较弱，可视化编程组件不够完善，美术人员难以上手.

除了商用引擎之外，各大网络游戏公司会根据需求开发自己的游戏引擎，其中有网易的 NeoX 引擎，已经开发了 10 年，经历了数款游戏的开发；网易的 Messiah 引擎，用于《天下》的开发；腾讯的自研引擎 Quicksilver.

不同于游戏行业的迅猛发展，在游戏引擎方面，国内与与国外的技术差距还是很大.主要有下面几个原因：

1. 国内游戏公司追求快速利润，从上面列举的自研游戏引擎来看，只有大公司才有自研引擎的团队也只有大公司才有自研引擎的能力，国内的游戏行业基本处于被网易、腾讯垄断的状态，除了这两个公司之外很少有公司愿意投入巨大的成本进行游戏引擎的研发.

2. 国内很少有传统的主机游戏平台 3A 级大作(A 级资金投入,A 级人员规模,A 级广告包装) ，游戏公司更加喜欢开发移动端休闲游戏，没有主机市场也没有高品质游戏来驱动优秀游戏引擎的发展.

1.2.2 国外研究现状

国外的游戏行业非常成熟，商业游戏引擎不仅数量多而且质量好，通常国外的引擎公司可以进行全套引擎的开发，不止集成了渲染引擎，也包括了动画系统、逻辑系统、编译解析系统等[错误!未找到引用源。].

第一款世界上的商用授权引擎由 John D. Carmack II 在 90 年代主导开发，是 Id Software 公司自主研发的 Id Tech 1. 目前国外的主流商业引擎包括：

1. Unity Technologies 公司开发的 Unity 游戏引擎.有丰富的第三方资源和完善的工具链，开发效率高，在移动端表现优秀.诞生了很多优秀的独立游戏作品.

2. 由 Epic 公司开发的 Unreal 游戏引擎.可以使用蓝图进行可视化编程，在视觉效果方面表现优异，适合开发大型游戏.同时 Unreal 游戏引擎开源，方便进行维护.

3. 由 Crytek 公司开发的 CryEngine 游戏引擎.CryEngine 的渲染引擎非常优秀，在渲染引擎模块集成了相当多的高级渲染技术，但是，CryEngine 的工具链和生态圈不够完善.

除了商业引擎之外，国外的很多游戏工作室都使用自研引擎，包括由 RockStar 公司开发的 RAGE 引擎，用于开发 GTA 系列；Ubisoft 的 Anvil 引擎，用于开发刺客信条系列；Square Enix 公司的月光引擎，用于开发最终幻想系列.这些游戏公司的自研引擎大多比商业引擎更加先进，例如顽皮狗引擎采用了基于体素的人物渲染技术[错误!未找到引用源。]和水面渲染技术[错误!未找到引用源。]渲染出来的画面可以达到照片级画面，即时演算在 PS4 上可以达到 30 帧.

在学术界的实验性渲染引擎方面，国外也存在着 PBRT [错误!未找到引用源。]，Mistsuba，LuxRender 等优秀的开源引擎，这些开源引擎都是基于 CPU 的离线渲染引擎，会将学术界的最新成果

集成到渲染引擎中，是不可多得的科研学习材料.

1.3 本课题主要工作内容

1. 研究并实现了基于体素渲染的实时全局光照算法，全部的渲染计算都在 GPU 中实现，并且在场景中实现了动态光源以及基于物理的渲染，在场景中可以达到 40 帧以上的可交互帧率.

2. 研究并实现了基于 GPU 的体素化，此体素化方法可以实时地进行动态场景的体素化，并且可以拓展到 Unity 等主流游戏引擎中进行应用.

3. 本课题搭建了一套 DirectX11 底层的框架，通过设置 DirectX 的渲染管线，使其可以用于基础渲染，框架中包括光栅化、多种缓存的设置，相机模型系统，材质模型读取系统，动态光源以及材质的计算，CPU 与 GPU 之间的数据传输等.此框架可拓展实现动画系统、粒子系统等渲染引擎模块以及各种高级渲染技术.

4. 本框架采用了基于物理的渲染模型，此模型比传统的局部光照模型更真实也更方便美术人员调整渲染效果，是近年来才在主流游戏引擎上应用的高级渲染技术.同时集成了包括基于图像的光照(Image-based Lighting)、后期处理(post processing), 卡通渲染(Cartoon Shading)等渲染技术和渲染效果.

5. 在本框架上搭建了一套 GUI 系统，使得本框架可以实际的让美术人员进行效果的调整和应用.

1.4 本文组织结构

本文的研究是基于 DirectX11 的渲染框架的设计与实现，在此框架上的核心工作为基于体素化的实时全局光照算法 SVO 的研究与实现，本文共分为 5 个章节，每个章节的结构如下：

第一章，绪论.介绍了课题的研究背景以及意义，国内外目前在光照渲染技术上的研究现状，说明了课题的主要工作.

第二章，相关理论与技术.主要介绍 3D 渲染编程管线，包括 DirectX 渲染管线以及 shader 渲染管线，并且介绍了主要的离线渲染算法以及实时渲染算法.

第三章，渲染框架的底层架构.详细介绍了渲染框架的总体设计以及各个模块的实现方法.

第四章，实时全局光照算法的实现.介绍了基于 SVO 的全局光照算法的各个部分的实现细节，包括直接光照，间接光照以及阴影的实现.

第五章，算法在 Unity 中的实际应用.阐述了如何将算法集成到 Unity 中，并且搭建复杂场景和集成高级渲染技术.

第六章，结果与比较.展示了系统的渲染结果，并与 DirectX11 固定管线，3DSMAX 渲染的结果进行对比.

第七章，结论与展望.总结全文并且探讨了本框架以及算法在未来仍需改进的地方.

1.5 本章小结

本章为第一章，绪论.首先介绍了本课题的研究背景，阐述了光照渲染技术在游戏开发，动画开发以及仿真应用中的不可或缺性，以及国内对于此技术需求的紧迫性.接着概述了国

内以及国外光照渲染技术的发展，以及目前存在的问题，最后介绍了本课题的主要工作并对本文的组织结构进行了安排说明.

第 2 章 相关理论与技术

2.1 渲染管线

2.1.1 渲染流程概述

渲染流程指的是由 CPU 发起指令，到显卡中执行的过程，该过程从几何体到渲染出最终图像，需要计算以及传输数据.其中每一个计算传输数据阶段都有各自的流水线.

渲染流程共分为 3 个大的概念性阶段，第 1 阶段为应用阶段，第 2 阶段为几何阶段，第 3 阶段为光栅化阶段.第 2 阶段和第 3 阶段由于都在 GPU 上进行，合称为 GPU 渲染管线.[错误!未找到引用源。] 下面分别介绍 3 个阶段各自的流水线.流水线图示如下图 2-1：

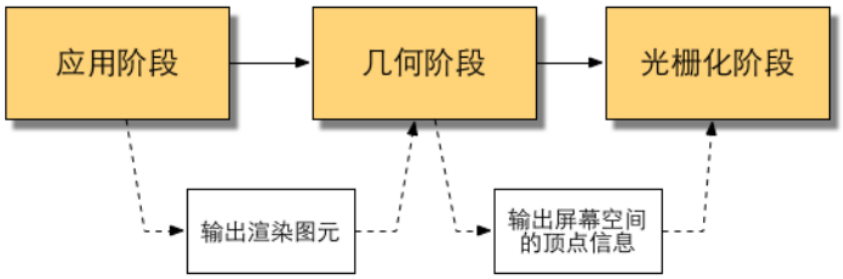


图 2-1 渲染管线总阶段

第 1 阶段，应用阶段.此阶段通常由 CPU 进行执行，由应用的开发者负责实现.以 3DSMAX 为例，设计师在 3DSMAX 中建立模型数据，建立摄像机的坐标位置，光源的坐标位置，光源的属性以及各模型所包含的材质属性，设置场景的渲染属性.开发者设置了这些渲染所需的属性，将这些渲染图元输入到下一个阶段.

第 2 阶段，几何阶段.此阶段通常在 GPU 上进行执行，负责处理上个阶段传入的每个渲染图元，主要处理模型的顶点变换，例如将模型的每个顶点进行坐标系变换.处理完之后将这些顶点信息传入第 3 阶段.

第 3 阶段，光栅化阶段.此阶段通常在 GPU 上进行执行，主要进行逐像素操作，包括进行逐顶点光照、着色等操作.用于匹配每个渲染图元的像素位置和屏幕上的像素位置.

下面将详细介绍 3 个阶段内部各自的流水线.

2.1.2 CPU 与 GPU 的通信

GPU 与 CPU 的通信需要 3 步，在应用阶段实现.

第 1 步，在 GPU 渲染管线之前，所需要的数据从硬盘加载到内存中，之后图形资源被加载到显存中.

第 2 步，设置渲染状态，定义设置着色器、光源、材质等属性.

第 3 步，调用 Draw Call 渲染命令，CPU 发起 Draw Call 后，GPU 就会进入 GPU 渲染管线，根据设置的渲染状态以及顶点数据进行渲染计算.

2.1.3 GPU 渲染管线

渲染概念管线中的几何阶段和光栅化阶段合称为 GPU 渲染管线，每个阶段都有不同的可配置性、可编程性以及 GPU 固定实现的不可配置性.GPU 渲染管线如图 2-2：

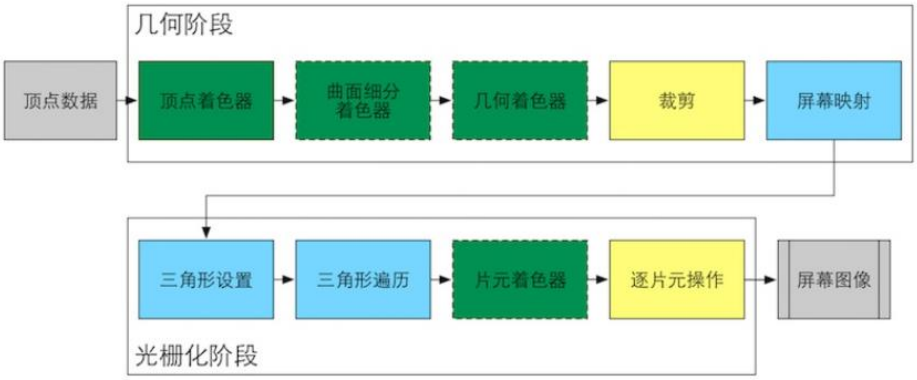


图 2-2 GPU 渲染管线

1. 几何阶段：

顶点着色器是管线的第一个阶段，它是完全可编程的着色器，输入数据来源于应用阶段.顶点着色器可使用的图元类型为单个顶点，主要工作为输出数据、逐顶点坐标变换.

曲面细分着色器是可选的着色器，可以将一个渲染图元分解成若干个小碎片.可以实现精度更高的凹凸贴图等渲染技术.

几何着色器也是可选的着色器，可以用于产生更多的图元，例如将一个点拓展为四边形.

裁剪阶段是可配置不可编程的阶段，用于裁剪在摄像机视野范围之外的几何图元.

屏幕映射是在 GPU 中固定实现的，工作是将每个顶点从他们的模型坐标系转换到屏幕空间坐标系中.

2. 光栅化阶段：

三角形设置阶段是 GPU 固定实现的，计算每个图元，每条边的像素坐标.

三角形遍历阶段仍旧是 GPU 固定实现的，用来遍历计算屏幕上的哪些像素被三角形图元所覆盖.

片元着色器是完全可编程的，此阶段给每个片元进行着色，是应用高级渲染技术的重要着色器.

逐片元操作是可配置不可编程的阶段，此阶段包括深度测试、模板测试和混合操作.

最终片元通过混合阶段输出到屏幕上，成为我们所看到的图像.

2.2 3D 图形编程管线

2.2.1 图形 API 概述

实时的 3D 图形编程需要依靠于图形 API，图形 API 用于调用 GPU、CPU 的接口，以实现与底层硬件的交互，图形程序员使用图形编程接口 API 来进行图形的渲染，高级渲染技术以及引擎的集成.目前主流的图形编程 API 包括 DirectX、Opengl.其他的图形计算平台包括最新的底层 API Vulkan，通用计算平台 CUDA.

DirectX 是 Microsoft 公司集成的一系列图形接口，包括 Direct3D、DirectInput、Direct2D 等，主要应用于 Windows 平台下的游戏开发，现版本为 DirectX12.

Opengl 是开放的图形接口，主要应用于跨平台开发，科学可视化，同时衍生出的 WebGL

应用能力也非常强大，现版本为 4.5.

Vulkan 是在游戏开发者大会（GDC）上发布的.可以提供更低的 CPU 开销与更加直观的 GPU 控制同时可以允许 CPU 上的集成显卡与独立显卡共同工作，是次世代的图形 API.

CUDA 是由 NVIDIA 公司推出的用于进行 GPU 通用运算（GPGPU）的集成统一计算架构.通过 CUDA 可以在 GPU 上进行大规模的并行运算，同时 CUDA 可以与传统图形 API：DirectX，OpenGL 交互开发，完成数据的共享通信.

2.2.2 图形 API 与硬件的关系

常见的图形编程接口，包括 OpenGL 和 DirectX 架起了应用程序和 GPU 的通信桥梁，它们把图形资源转换成了 GPU 支持的数据格式.应用程序从 CPU 向图形 API 发起渲染命令，图形 API 接口接着向显卡驱动发起渲染命令.显卡驱动调用 GPU 的接口，真正的和 GPU 进行通信.

将底层图形 API 封装起来的一个个组件就成了渲染引擎，可以说图形接口就是渲染引擎的地基[错误!未找到引用源。]，现在的主流游戏引擎的渲染引擎部分都同时支持 DirectX 和 OpenGL.

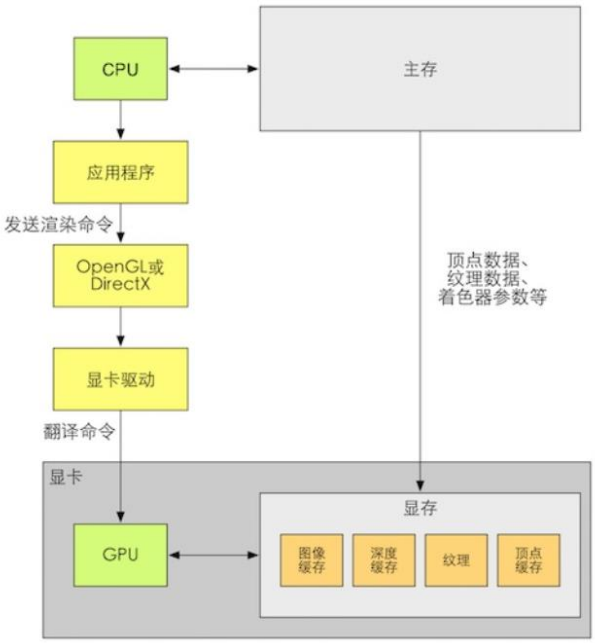


图 2-3 图形 API 与硬件关系图

2.2.3 DirectX11 概述

DirectX11 是在 2009 年 10 月发布的新一代底层图形 API 集合.

DirectX11 是一系列组件的集合，其中主要包括用于加速 2D 图形渲染的 Direct2D、用于加速 3D 渲染的 Direct3D、用于处理声音的 XAudio2、用于访问底层数据资源的 DXGI 框架、用于接收处理 I/O 借口的 XInput 等.

DirectX 系列图形 API 从 DirectX10 开始引入可编程渲染管线的概念，DirectX11 在 DirectX10 的基础上引入了曲面细分(Tessllation)的 3 个阶段，应用于处理曲面的几何处理上例如人脸模型、发质模拟等，在曲面细分阶段中无需手动设置百万个三角形的复杂模型，可以将需要模拟的模型进行勾勒，其余交给曲面细分阶段进行自动镶嵌.

同时 DirectX11 还引入了 GPU 通用计算技术(GPGPU)[错误!未找到引用源。]，可以让 GPU 处理图

形数据以外的其他数据类型.支持物理数据的传递以及运算，顶点数据的传送，在 compute shader 中进行大规模数据的并行运算，也可以在 compute shader 中进行优化算法的构建，例如 KD-Tree，Octree 等.

2.3 渲染方法的分类

渲染方法指的是一个片元在屏幕上进行绘制的方法，分为光线追踪(ray tracing)和光栅化(rasterization)两种.两种方法的实现算法互为逆过程，也导致了他们效率以及准确性的不同.不论是实时渲染或是离线渲染，都离不开这两种最基础的渲染方法.

2.3.1 光线追踪

光线追踪是一种将 3D 模型在屏幕上生成为平面像素的渲染方法，跟踪从眼睛发出的光线，算法如下：

首先对从摄像机开始，对屏幕上的每个像素发射一条射线.然后每条光线逐渐向摄像机发出的方向追踪物体，直到它追踪到物体的表面.如果追踪到了物体的表面，就黑这个像素点着色，若该射线超出了最大追踪范围后仍没有追踪到物体，则没有屏幕上的像素点被着色.

以上是最基本的光线跟踪渲染方法，根据渲染效果的要求，光线追踪还可以进行拓展，例如发射阴影光线，以检测物体的哪部分对光源是可见的、发射折射光线和反射光线来模拟模型物体的不同材质[错误!未找到引用源.].光线追踪的算法图示如下图 2-5：

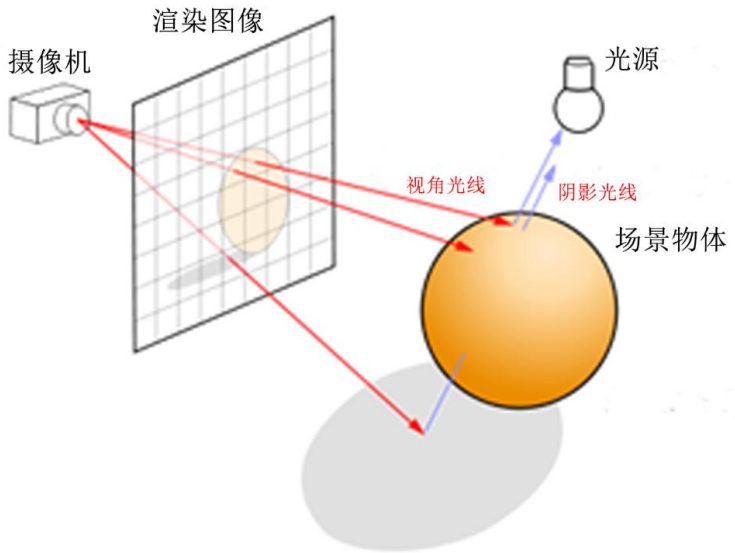


图 2-5 光线追踪算法

光线追踪的优点是精确，例如可以更加准确的模拟反射与折射，所以当追求高质量的效果时经常使用这种方法，例如在动画渲染和电影渲染中.同时算法简单直接，易于实现.

但是由于光线追踪需要对每个像素点进行可见性检测，计算代价非常高，通常在离线渲染中进行应用.最近两年由于硬件的提升，逐渐有游戏引擎尝试在实时渲染中使用光线追踪算法，在 GPU 中进行大量数据的并行运算以实现光线追踪，但是这仍然处于实验性阶段，没有广泛应用.

2.3.2 光栅化

光栅化是将模型的几何图元转变为 2D 图像的过程，为光线追踪的逆过程，主要包括两个过程，片元投影以及像素循环.光栅化算法图示如下图 2-6：

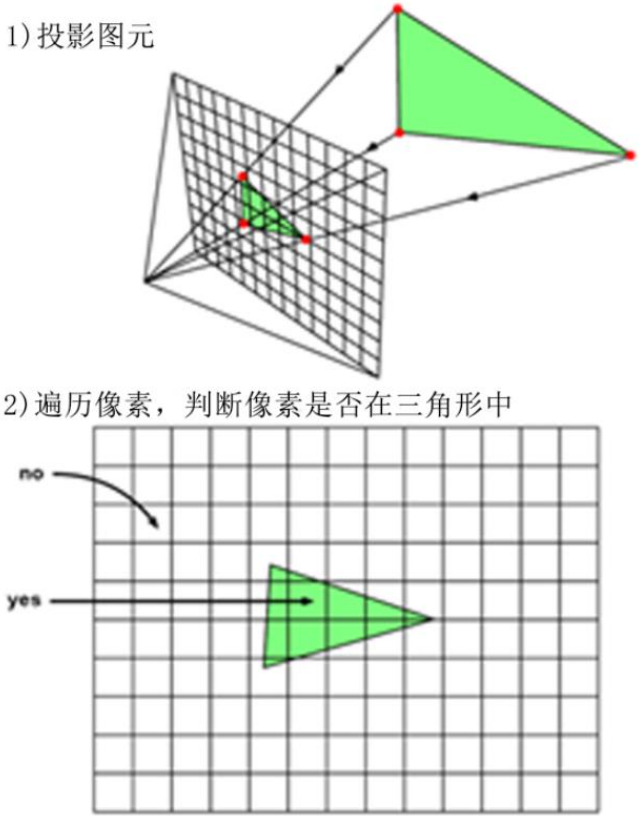


图 2-6 光栅化算法

为了解决可见性问题，它将每个片元用正交投影或透视投影，将片元投影为三角形投影到屏幕上.之后将片元的边界位置以及区域位置确定，确定屏幕上有哪些像素覆盖了这个三角形片元.

光栅化大量应用于实时渲染，是 GPU 渲染管线的基础，几乎所有的图形 API 都集成了光栅化算法.

光栅化虽然效率非常高，但是精确度远远不如光线追踪，同时由于光栅化是 GPU 渲染管线中的不可编程阶段，图形 API 又涉及硬件底层接口，开发者需要熟悉整套渲染管线，开发门槛较高.

2.4 全局光照算法

2.4.1 离线全局光照算法

由于渲染方程(Rendering Equation)[错误!未找到引用源.]是一个沿全空间方向的积分，公式如式 2-1：

$$I(x,x') = g(x,x')[\xi(x,x') + \int_S \rho(x,x',x'')I(x,x')dx''] \quad (2\text{-}1)$$

计算渲染方程使得渲染非常耗时，在现有的硬件条件下几乎不可能按照原始的渲染方程进行渲染.于是在离线渲染领域，基于原始的渲染方程提出了几种简化模型：

1. 辐射度算法：1984 年由 Cindy M G.提出[错误!未找到引用源.]，辐射度算法通常是基于有

限元分解的，它将一个沿无限空间的积分分解为一个有限纬度的积分进行漫反射计算，辐射度算法需要对场景进行参数化，将其结果保存在纹理中进行预计算全局光照.

2. 路径追踪算法(Path Tracing)：1986 年由 Jamespath K.[错误!未找到引用源。] 提出，与光线追踪方法类似，该算法由摄像机发射光线，光线在场景中反复迭代，比起光线追踪方法采用蒙特卡罗随机采样的方式，路径追踪采用更加物理的过程.之后 Eric P L.又在此基础上提出了双向光线追踪(Bi-directional Path Tracing)，该方法可以有效减少噪声.路径追踪的优点是渲染图像极具真实感，但是速度非常慢.

3. 光子映射算法(Photon Mapping)：于 1996 年由 Henrik W J.提出[错误!未找到引用源。]，该算法模拟光子发射的过程，先生成光子图，再采用路径追踪、光线追踪等基础算法，两个步骤叠加进行渲染，光子映射在渲染质量和渲染速度上都在不断进行优化.

2.4.2 实时全局光照算法

由于实时渲染领域的主要光照渲染思路是根据理论拆分渲染方程，使得光照结果最终由多种效果叠加而成，常使用不同的局部渲染技术进行合成，例如软阴影，直接光照，高光，间接光照，AO [错误!未找到引用源。]，环境光等.这些效果分别使用不同的方法进行计算，也可以对一个场景构建一些特定的数据结构来加速不同效果的运算.

1. 基于体素的全局光照**错误!未找到引用源。**[15]（Voxel-based GI）：基于体素的渲染算法使用体素化的方法对直接光源进行模拟，同时进行 3D 空间内的可见性检测，再在二次渲染时进行光线追踪，此方法在内存消耗上损耗较大，本渲染框架主要对这种全局光照技术进行研究与实现.

2. Light Propagation Volumes **错误!未找到引用源。**（LPV）：LPV 算法用于提供实时的间接漫反射光，并且它只提供一次反弹.这种方法首先使用 Instant Radiosity（IR）技术生成更高密度的虚拟点光源，但并不渲染这些光源，而是使用它们来生成一个 3D 空间更稀疏的具有方向性的间接光表述，这种表述可以用于实时快速插值运算. LPV 算法也是虚幻 4 引擎中应用的全局光照方法.

3. 预计算全局光照：预计算的全局光照一般采取光照贴图的方式，将光源渲染在贴图中，在实时渲染时再将光源采样到图像上，也可以说是一种基于图像的全局光照方式(Image based lighting)，光照贴图一般采用辐射度方法进行计算.Unity5 中的全局光照系统 Enlighten 使用的就是此种方法.

2.5 本章小结

本章介绍了 3D 图形学基础知识以及本课题的背景知识，详细阐述了以下几点：

1. 3D 图形学重要概念基础：渲染管线的概述及其各个阶段的主要工作.
2. 3D 图形学实践编程基础：图形 API 与 DirectX 的概述.
3. 渲染的基础算法：光线追踪算法与光栅化算法的原理.
4. 现阶段的全局光照解决方案：包括实时全局光照以及离线全局光照.

第 3 章 渲染框架的实现架构

3.1 总体架构

本框架基于 DirectX11，在开发后期进行过架构重构，下面以重构后的渲染框架进行说明.

本框架共分为系统接口模块，渲染模块，DirectX 接口模块，I/O 接口模块，模型模块，摄像机模块，渲染状态设置模块，下面进行总体架构以及分模块架构说明.

总体架构分为 CPU 和 GPU 两个部分，分别在 CPU 上和 GPU 上进行运算.

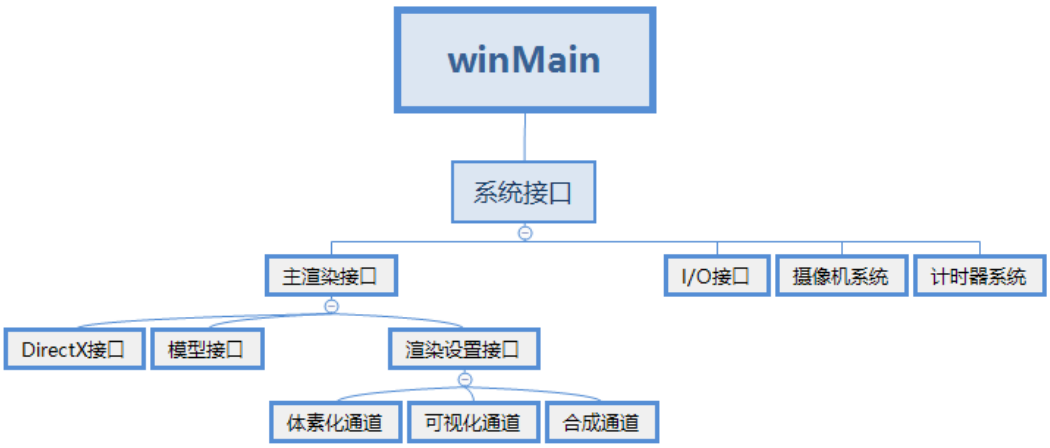


图 3-1 总设计框架图 - CPU 部分

CPU 方面主要由 C++通过 windows 程序调用 Directx11 接口，进行应用数据从 CPU 发起的传递.包括常量缓存的设置，shader 的设置，模型的设置，材质的设置等.同时经由 DirectX 的底层设置，与 GPU 进行交互，这部分包括 shader resource view，depth buffer（深度缓存），模板缓存（stencil buffer）等.

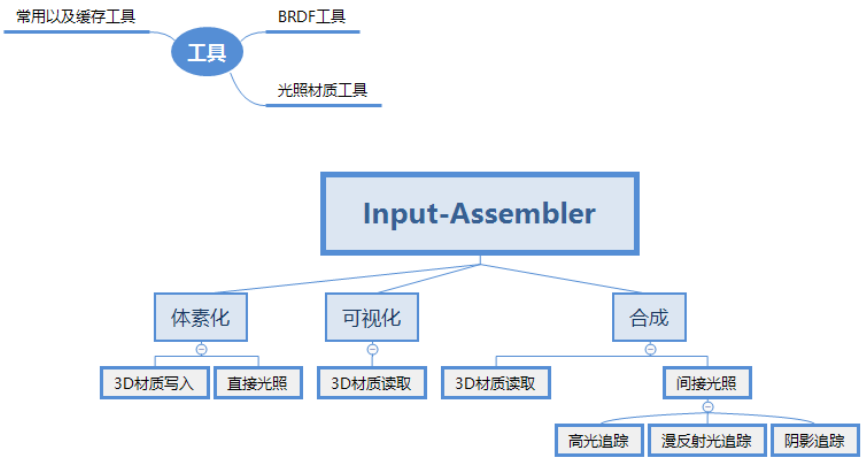


图 3-2 总设计框架图 - GPU

GPU 方面主要进行坐标系变换，体素化，光照模型，材质模型等计算，使图像可以进行实时的渲染运算.是算法的核心部分.

3.2 子功能模块

下面对系统内主要的子功能模块进行实现解释说明.

3.2.1 系统接口与渲染接口

系统接口直接被 winMain 函数调用，主要处理下面几个模块：

1. windows 窗口的初始化设置以及每次接收的事件信息和每帧更新的渲染事件

2. 主渲染接口，负责调用 DirectX 设置信息、加载模型、加载材质以及贴图、设置 3 个 shader 应用的渲染状态.

3. 摄像机接口，建立摄像机的位置、方向、目标、投影矩阵以及视口矩阵.

4. I/O 接口程序，通过导入 DirectInput 库进行与 I/O 设备的交互，处理了摄像机向各个方位的移动以及 I/O 设备对软件发出的消息.

5. 计时器程序，用于计时开始、累计、每帧计时计算.

3.2.2 DirectX 接口

DirectX11 接口负责初始化 D3D 设备，D3D 上下文设备，设置视窗，渲染设置等.初始化 DirectX11 需要引入包括核心库<d3d11.lib>，shader 编译库<d3dcompiler.lib>，错误库<dxerr.lib>，GUI 库<dxguid.lib>，这些库都在 C++的基础链接输入(Linker Input)中引用.

初始化 DirectX 需要设置以下几个 DirectX 属性：

1. 初始化 DirectX 设备(device)，DirectX 通过交换链来显示图形，用于检测适配器功能和分配资源，DirectX 设备用于创建交换链(swap chain)，句柄，限制最大帧率，缓存设置等.

2. 初始化 DirectX 设备上下文(device context)， 用于创建资源、计算显示接口性能，生成渲染命令.设备以及上下文是最重要的接口，我们通过这个接口与硬件进行交互.

3. 设置 4X 多重采样质量等级(MSAA).

4. 为交换链的后台缓冲(back buffer)渲染目标视图(render target view)，渲染目标视图用于将资源绑定到不同管线阶段.

5. 创建深度缓冲(depth buffer)，模板缓冲(stencil buffer).

6. 将深度缓冲以及模板缓冲绑定到 GPU 渲染管线的输出阶段.

7. 创建目标程序视口，用于设置场景渲染到后台缓冲的子矩形区域.

3.2.3 摄像机系统

摄像机系统可以看做第一人称视角，利用摄像机系统在屏幕上渲染 3D 场景的图示如下：

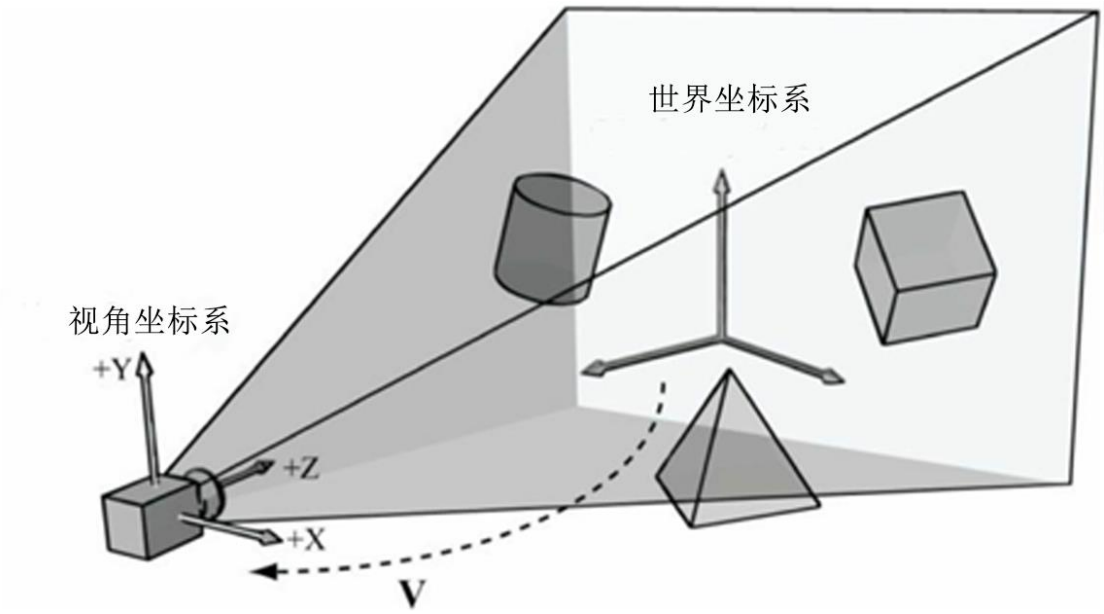


图 3-3 摄像机系统示意图

在摄像机类中构建了用于实现坐标系变换的视角矩阵、投影矩阵并且实现了前后左右上下的平移以及视角旋转.

在一个摄像机系统中，我们需要定义摄像机坐标系或者视口坐标系(view coordinate)，这个摄像机坐标系包括几个主要参数：三个主坐标轴、视角矩阵(view matrix)、投影矩阵(projection matrix)、视角大小(field of view)、摄像机的位置、宽高比、近裁剪平面、远裁剪平面.裁剪平面的定义图示如下：

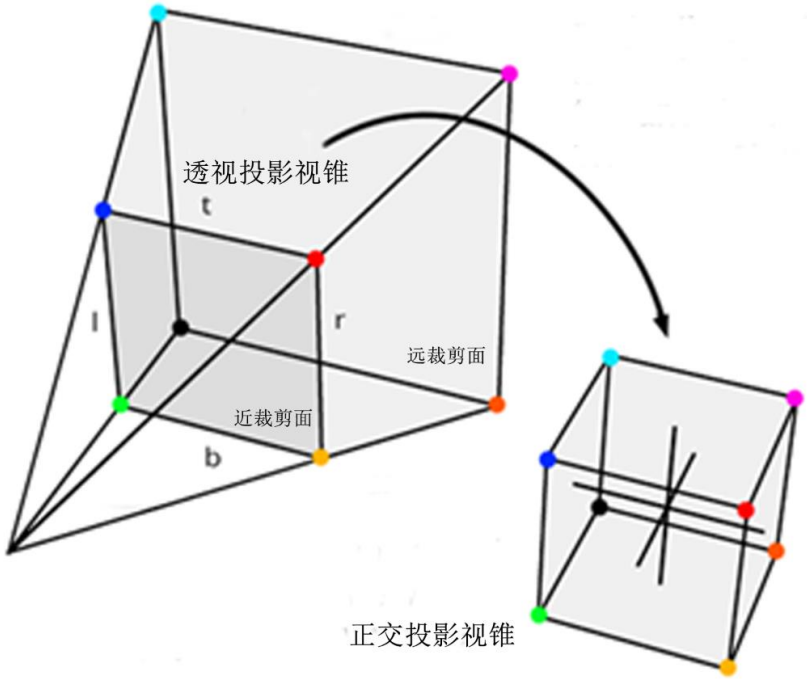


图 3-4 裁剪平面示意图

我们在依据摄像机的四个参数视角大小、宽高比、近裁剪平面、远裁剪平面求解出视角矩阵，用 DirectX 的内置函数 XMMatrixPerspectiveFovLH 求出投影矩阵.视角矩阵如下：

$$\mathrm{v}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_x & v_x & w_x & 0 \\ u_x & v_x & w & 0 \\ -\mathrm{Q}\cdot\mathrm{u} & -\mathrm{Q}\cdot\mathrm{u} & -\mathrm{Q}\cdot\mathrm{u} & 1 \end{bmatrix}$$

其中 u 为摄像机系统的 x 坐标轴、v 为摄像机系统的 y 坐标轴、w 为摄像机系统的坐标轴、Q 为摄像机位置.

3.2.4 模型系统

模型系统主要通过解析 obj 文件得到模型的顶点信息，法线信息，贴图信息，顶点索引，法线索引，贴图坐标索引等信息.

Obj 文件的格式见下表 3-1：

表 3-1 obj 文件解析格式

索引	类型	数据类型
v	顶点位置	3 个浮点数
vt	纹理坐标	2 个浮点数
vn	法线向量	3 个浮点数
f	三角形图元索引	v / vt / vn
f	三角形图元索引	v//vt

通过使用 tinyobj 解析此文件，tinyobj 可以获得 obj 文件的形状(shape)以及图元信息，读取过程分为三步.

1. 将法线、坐标、贴图信息从每个表面(face)读入，存入作为顶点缓存的数据结构.
2. 将他们重新进行映射，存入作为索引缓存的数据结构.
3. 再将读入的信息按照他们的带宽存入 DirectX11 的顶点缓存中.

设置了模型的缓存之后，在 Render 函数中对 D3D 上下文设备每帧传入顶点缓存(vertex buffer)和索引缓存(index buffer).

模型系统还需要获得模型的边界盒子（boundingbox）以计算体素坐标系中的偏移.

3.2.5 渲染状态接口

渲染状态设置接口主要设置了

1. 读取的 shader 以及 shader 的类型，例如 shader model 4.0 或是 shader model 5.0.
2. 常量缓存的设置以及更新.包括矩阵常量结构：世界矩阵(world matrix)，视口矩阵(view matrix)，投影矩阵(projection matrix).普通常量结构：视角位置以及更新的时间.体素常量结构：体素的偏移，维度和大小.材质常量结构：漫反射系数，高光系数.
3. 3D 贴图以及 2D 贴图的设置，包括 mipmap 等级，贴图的大小维度，在 DirectX 图形资源中绑定的类型，例如无序资源视图（unordered access view），shader 资源视图（shader resourse view）等.
4. Shader 中 techinque 以及 pass 的设置和更新.拓扑结构的更新.

5. input-assembley 的数据类型以及带宽.

根据每一个 GPU 上 shader 应用的不同，各自的渲染状态也不尽相同.

3.3 本章小结

本章描述了本框架在 CPU 上的实现与设计原理，描述了渲染器的实现细节和整体框架，本框架由系统接口出发，具体讲述了各个子功能模块的实现原理，包括摄像机系统、模型的导入、CPU 到 GPU 数据的传递等.在 GPU 部分也进行了总体框架的描述，由于 GPU 上的计算为核心算法，将在下一章（第 4 章）进行详细描述.

第 4 章 渲染框架的核心算法实现

4.1 体素化

4.1.1 体素化的意义

体素化(voxelization)也被称为三维扫描转换(3D scan conversion)，是将物体从几何体转换为最接近该物体的体素表现，可以看做将物体从二维的点拓展到三维的立方体单元[错误!未找到引用源。]，其结果是体数据集.体素化有诸多应用，在流体模拟，碰撞检测，细节模拟和实时全局光照上都有广泛的应用前景.

体素化在空间中存储场景数据，与常规的光栅化不同的是体素化数据有深度值，而光栅化（见 2.3 节）只有 2 维平面数据. 体素化数据可以方便的和片元的世界坐标系进行转换，利用体素化存储的信息让一些渲染技术更加可信，例如 shadow mapping [错误!未找到引用源。]，ambient occlusion 等.同时也可以利用体素化信息进行光线跟踪来检测可见性.

4.1.2 体素化算法

在游戏开发中经常使用的体素化算法分为表面体素化算法、实体体素化算法[错误!未找到引用源。]、多值体素化算法和二值体素化算法，按照平台又可分为基于 CPU 的体素化和基于 GPU 的体素化.本框架采用的体素化算法是基于 GPU 的表面体素化方法[错误!未找到引用源。].

体素化方法可以概述为，取一个大小为$a*a*a$的三维贴图，该贴图为无序资源数据 UAV(unordered access view)，在 DirectX11 中的 shader 格式为 RWTexture3D，通过该数据类型，我们可以从 CPU 传入贴图数据，在 GPU 上进行读写，再传回 CPU，实现不同 shader 之间的贴图数据传递.

将此三维贴图细分为 n 个立方体，n 被称为为体素分辨率，在 geometry shader 中将每一个模型正交投影在平面上，在 pixel shader 中将模型世界坐标转换为体素坐标，在体素坐标系中保存三维贴图中的物体的坐标位置.每个物体的连续坐标值被抽象成了一个个在三维空间中离散的点值.下图 4-1 为三维贴图与体素化过程：

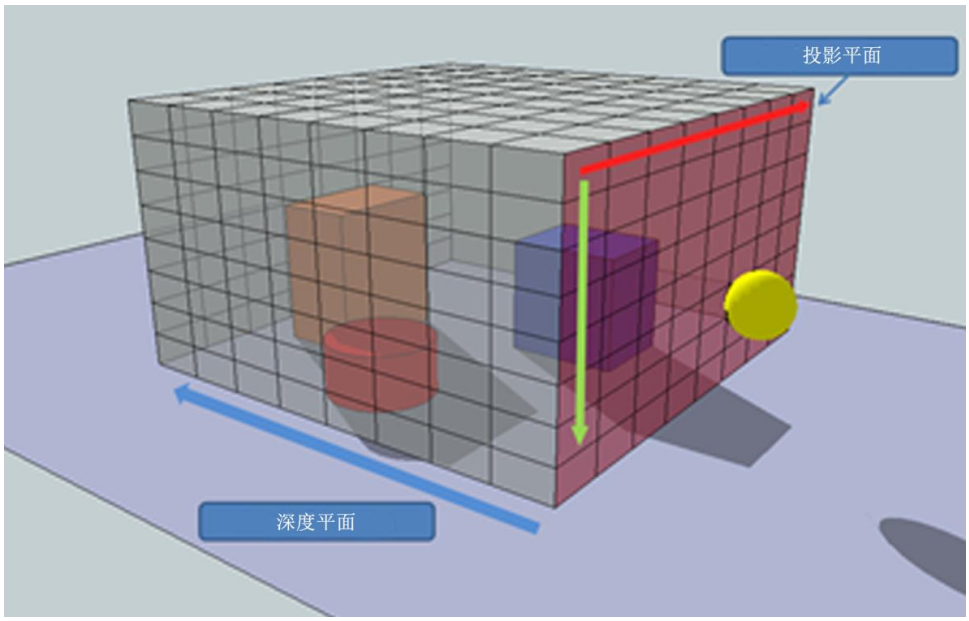


图 4-1 三维贴图与体素化

一个三角形 B 可以被判定为体素 V 的条件为：

1. B 的平面和 V 相交.

2. 三角形 B 沿着 3 条主坐标轴投影最长的法线，这个方向的三角形的 2D 投影面积和 V 的 2D 投影相交.

下图 4-2 与图 4-3 为主坐标系投影判断过程：

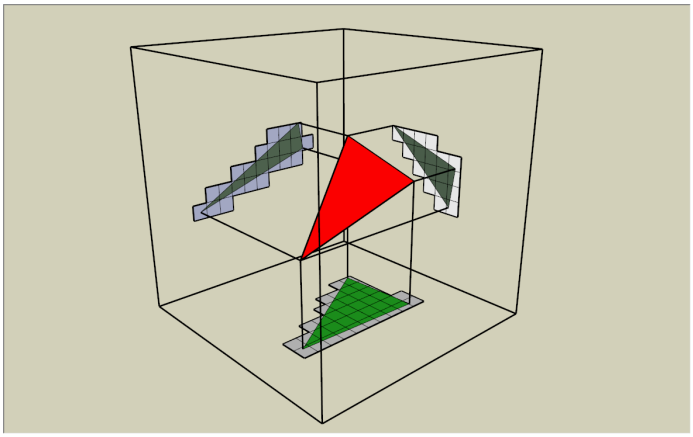


图 4-2 主坐标系判断

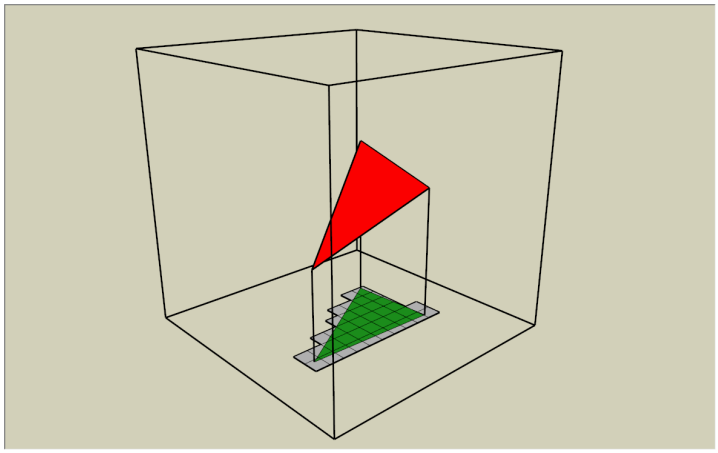


图 4-3 主坐标系投影

基于这个体素化方法，本框架在一个 Draw Call 中进行了 4 步的运算：

1. 在几何着色器中选择每个网格的三角形沿着主要坐标轴的法线 (法线在三个坐标轴平面上的投影距离最长)，这个法线方向是在几何着色器中自动进行判断的.对于每个三角形，有$l_{\{x,y,z\}} = \left|\mathbf{n} \cdot \mathbf{v}_{\{x,y,z\}}\right|$，n 是这个三角形的法线，而V{x, y, z}是三条主坐标轴.

2. 在几何着色器中将每个三角形沿着上一步计算出的主要法线方向进行正交投影，同时为了让整个模型都进入光栅化阶段，需要给模型一个基于原坐标点的偏移，将视角范围设置为 3D 材质的大小.因为 3D 材质的坐标系位置为 f(x,y,z)=(0,width), (0,height) (0,depth)，需要保存模型的世界系坐标位置，再将其转换为体素化位置.

3. 在光栅化阶段需要屏蔽深度测试，深度写入以及混合写入.

4. 在片源着色器中将进行光栅化后的片元进行光照处理，首先判断他们是否在体素坐标系中，如果存在，进行 Blinn-Phong（见 4.2 节）直接光照处理.并将其进行光照

计算后的颜色值写入 3D 材质.

完成了体素化之后我们可以得到一个有着直接光照信息的三维贴图，可以将其应用到间接光照的模拟上.

4.2 直接光照

4.2.1 局部光照模型

常见的光照模型分为局部光照模型和全局光照模型两种.其中局部光照模型也称为经验模型，通过提出经验公式，调整参数来模拟光线的反射与折射. 反射和折射在不同材质上的表现如下图 4-4.下面介绍两种光照模拟模型，Phong [错误!未找到引用源。]反射模型和 Blinn-Phong [错误!未找到引用源。] 反射模型.

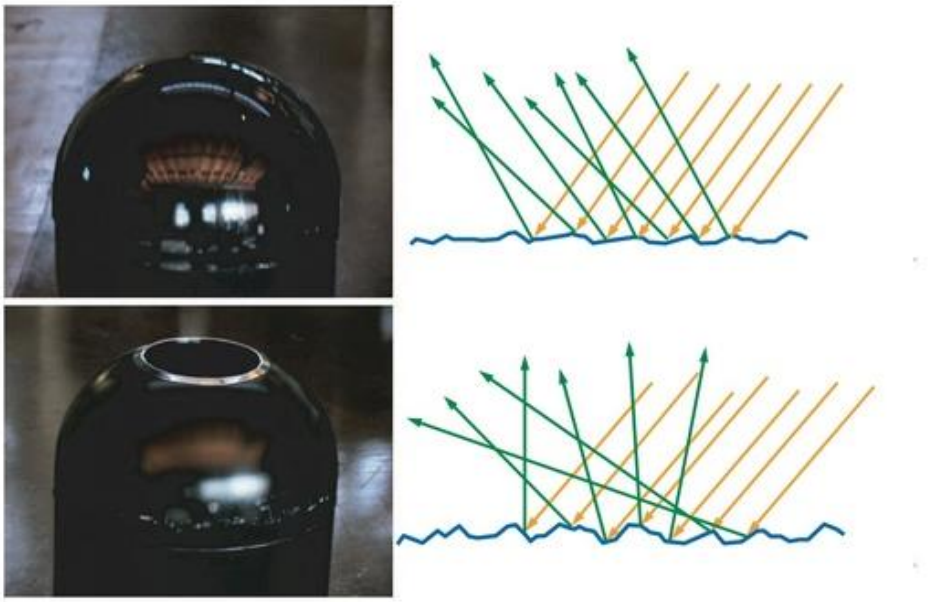


图 4-4 光线的反射与折射

Phong 模型是经典光照模型，它将漫反射光照，高光反射光照，环境光照分别按照经验模型求出系数并相加，公式如下.

$$I_{Phong} = k_a I_a + k_d(n \cdot l) I_d + k_s(r \cdot v)^{\alpha} I_s \tag{4-1}$$

Blinn-Phong 模型用半程向量与法线的乘积替代了反射向量与视角向量的乘积，使得它在 Blinn-Phong 模型在高光部分形状被拉长，更加真实.式(4-2)说明了 Blinn-Phong 的定义：

$$I_{Blinn-Phong} = k_a I_a + k_d(n \cdot l) I_d + k_s(n \cdot h)^{\alpha} I_s \tag{4-2}$$

在第一步的体素化之后，为了下一步的间接光照，我们需要进行光照运算，这里的光照运算不是为了得到准确的光照结果，而是为了模拟让间接光照在 3D 材质中进行采样.所以我们不需要采用高精度的光照模型，而是使用速度快但精度较低的 Blinn-Phong 模型来模拟体素化内的间接光照.

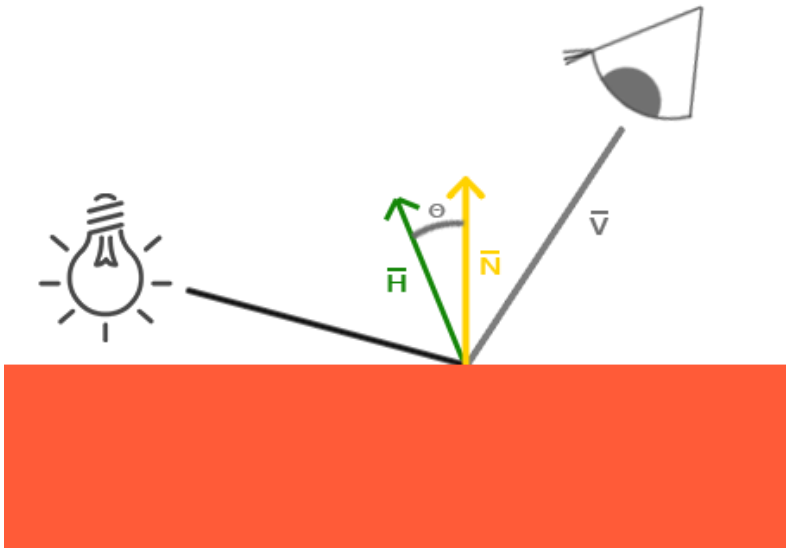


图 4-5 光线与视角的关系

在进行点光源的模拟时，我们需要 3 个分量，漫反射光照，环境光照以及高光(镜面)反射.

这三个属性分别用如下公式 4-3 进行计算：

$$A = k_a I_a$$
$$D = (n \cdot l) I_d$$
$$S = (n \cdot h)^a I_s \tag{4-3}$$

这部分的光照模型为体素化算法内使用的局部光照模型，用来进行间接光照的模拟，下面将介绍最终渲染合成通道内使用的基于物理的光照模型.

4.2.2 基于物理的光照模型

基于物理的渲染模型虽然已经提出了很多年，也在诸如电影、动画等离线渲染领域进行了广泛的应用，但其在游戏行业的应用是从近年才开始的，相比于上述的经验模型来说，基于物理的光照模型更能够渲染出真实的画面.

光照渲染模拟的是真实世界的光传输，1986 年 James T. Kajiya 在 SIGGRAPH 上提出了著名的渲染方程.图形学的目标之一就是求解该渲染方程.

$$L_0(v) = \int_\Omega f(l,v) \otimes L_i(l)(n \cdot l) \mathrm{d}\,\omega_i \tag{4-4}$$

式 4-4 为渲染公式.

其中的 f(l,v)为 BRDF 的贡献值，Li(l)是光源的贡献值.

对于入射光，表面会将光反射到表面上半球的各个方向，用双向反射分布函数 BRDF 错误!未找到引用源。(Bidirectional Reflectance Distribution Function)来描述入射光与反射光的关系.BRDF 的定义如式 4-5：

$$f(l,v) = \frac{dL_0(v)}{dE(l)} \tag{4-5}$$

其中 f 是 BRDF，l 是入射光方向，v 是视角方向.

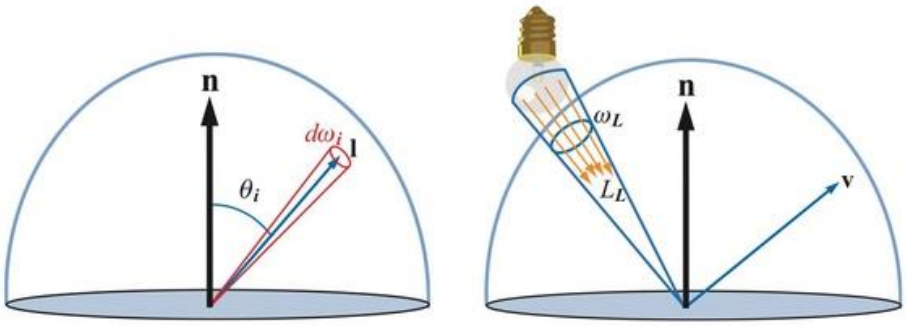


图 4-6 BRDF 中入射光与反射光的关系

BRDF 用来描述物体材质，它定义了材质在观察方向与光照在不同情况下对光源的反射.基于物理的 BRDF 模型满足下面两个特点：

1. 入射光的方向与出射光的方向互换不会对 BRDF 值造成影响.
2. 反射后的总能量与入射总能量满足能量守恒.

BRDF 基于微表面理论，用无数个微平面来模拟物体表面，微表面的法线方向只与半角向量 h 相同，这个微表面才是有效的. 此 BRDF 的表示公式如式 4-6：

$$\mathrm{f(l,v)} = \frac{D(h)F(v,h)G(l,v,h)}{4(n\cdot l)(n\cdot v)} \tag{4-6}$$

其中 D(h)代表了法向量分布，计算物体表面方向量的分布情况，本框架中使用的法向量分布为 GGX [错误!未找到引用源。] 公式如式 4-7：

$$D_{GGX} = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2} \tag{4-7}$$

F(v, h)代表了 Fresnel 项数，描述能量的反射与折射，本框架使用 Schlick **错误!未找到引用源。**的 Fresnel 项数，公式如式 4-8：

$$F_{Schlick}(v,h) = F_0 + (1-F_0)(1-(v\cdot h))^5 \tag{4-8}$$

G(l, v, h)代表了几何遮挡项数，计算微表面被遮挡的百分比，本框架中使用 Cook-Torrance [错误!未找到引用源。] 遮挡项，此项数效果最好，但开销较大，在离线渲染中经常被使用，公式如式 4-9：

$$G_{Cook-Torrance}(l,v,h) = \min(1, \frac{2(n\cdot h)(n\cdot v)}{v\cdot h}, \frac{2(n\cdot h)(n\cdot l)}{v\cdot h}) \tag{4-9}$$

根据上述的 BRDF 各项项数以及不考虑次表面散射的渲染公式，我们可以将基于物理的渲染进行如下应用：

4.3 间接光照

间接光照采用的是基于圆锥体跟踪的方法(cone tracing)，这个方法是在 ray-marching 的基础上进行改进的.下面先介绍 ray-marching 方法.

Ray-marching 的方法与光线追踪的本质是相同的，我们在这里使用 Ray-marching 来获

取间接光照的色彩值.在此方法中，我们从模型的坐标出发，按照法线平面上的 9 个方向来追踪漫反射光照.计算法线、切线和副法线可以得到这 9 个方向.

下图为追踪的 9 个漫反射光照锥体的方向.

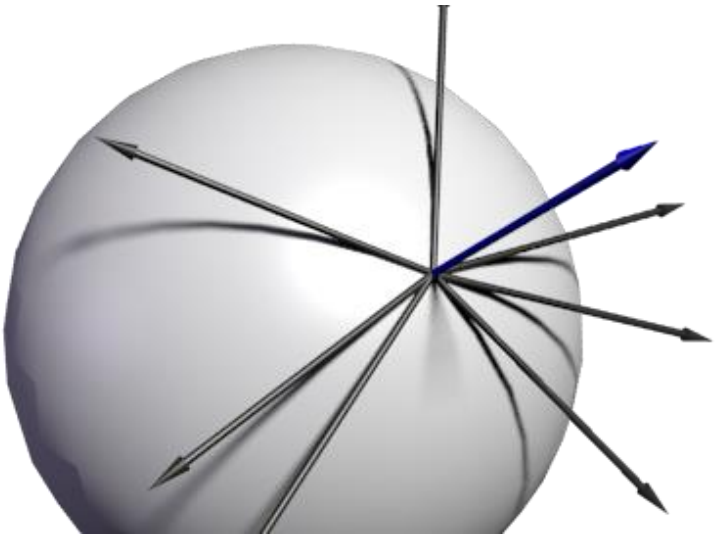


图 4-7 漫反射追踪的方向

我们从模型的坐标出发，沿着这 9 个方向进行光线跟踪，跟踪过程如下图所示，p0 为起始跟踪点，蓝线为单个追踪平面，每一步确定一个追踪距离，向前一点一点的追踪，直到追踪到了模型上的点.如果超出了最大追踪距离还没有追踪到任意一个点，则这个方向没有物体存在.

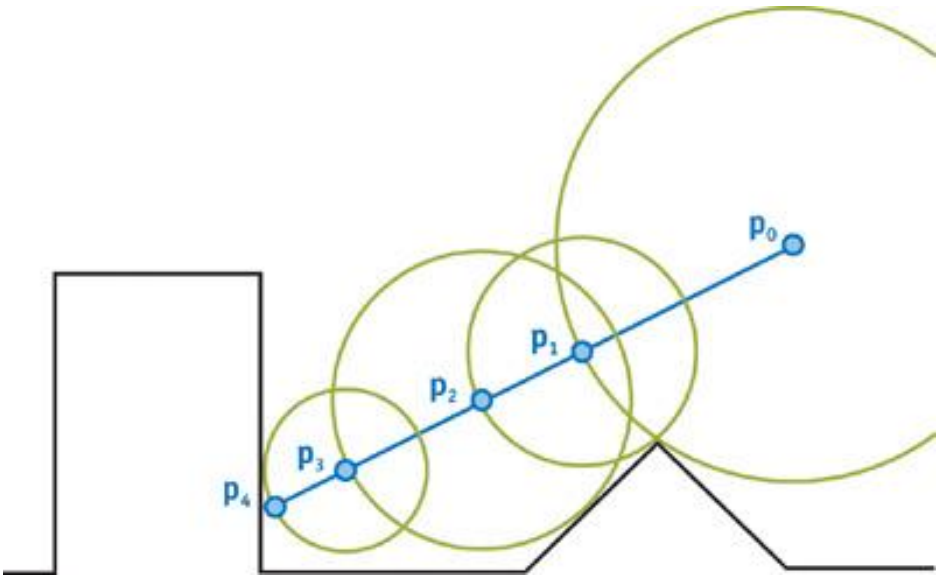


图 4-8 ray-marching 示例

下面按照 ray-marching 的算法拓展到 cone-tracing 方法.锥体跟踪将每一步追踪的距离由球体拓展到了锥体.Cone-tracing 算法如下：

我们确定一个颜色累加值初始为 0，给初始追踪距离一个偏移以免追踪到自身.确定一个锥体的角度，每次将角度(cone ratio)与追踪距离(dist)相乘得到追踪直径(diameter)，为了使间接光照有强弱渐变，设置采样时的 mipmap 等级为 log2(diameter). 发出射线，对每个

点进行采样，将采样得到的像素值加到累加值中，更新追踪距离.

Cone-tracing 的图示如下：

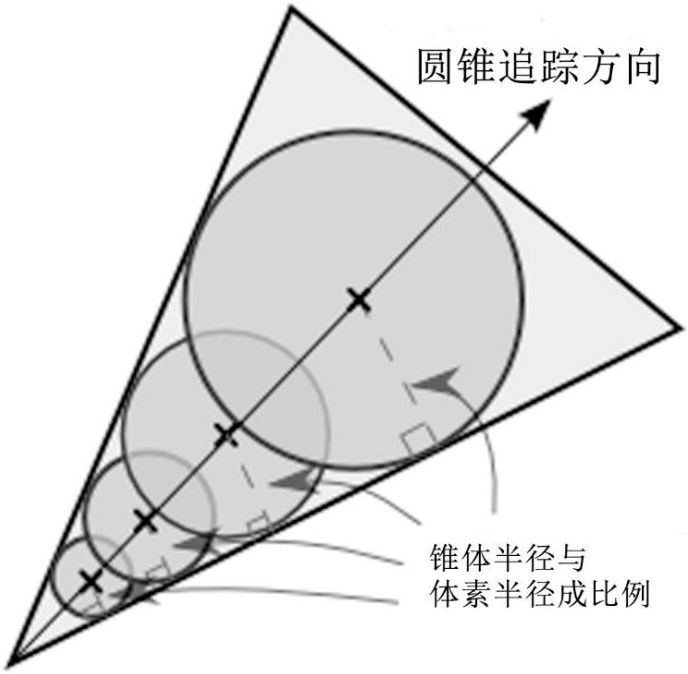


图 4-9 cone-tracing 示例

Cone-tracing 的伪代码如下：

```
float dist = voxel_size;

while(dist<MAX_DIST)
{
   float diameter = cone_ratio*dist;
   float lodLevel = log2(diameter / gVoxelSize);
   float3 ray = startPos + dir*dist;
   float4 color = gVoxelList.SampleLevel(Filter, ray / gDim + 0.5f / gDim, lodLevel);
   float a = 1.0f - accum.a;
   accum += color*a;
   dist += gVoxelSize;
}
return accum;
```

4.4 阴影

追踪阴影的方法与间接光照类似，不同的是我们这次从光源发出光线，追踪到模型的每个坐标，算出光线的最大追踪距离为光源到模型坐标的距离.如果光源在追踪过程中采样到了色彩值，代表这一点有物体存在，也就是说有阴影存在.我们将累计阴影值加上这一点的 alpha 值，如果在迭代追踪过程中 alpha 值大于 1，表示此点完全被阴影覆盖，跳出追踪过程.

下图展示了追踪阴影的过程：

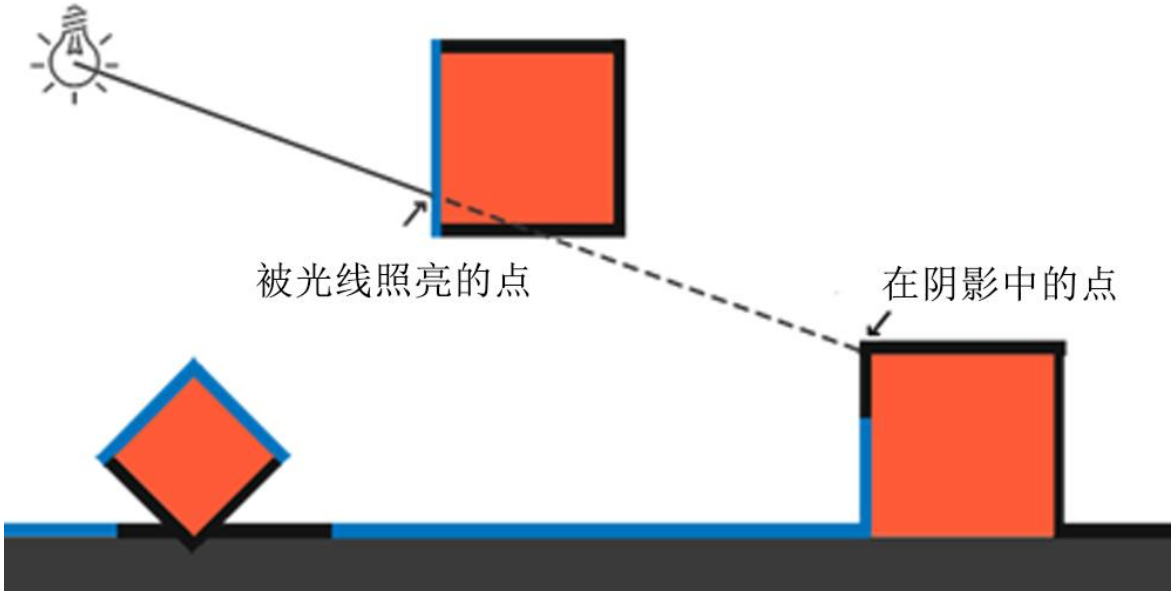


图 4-5 阴影追踪示意图

阴影追踪的伪代码如下：

```
float dist = gVoxelSize;
float STOP = length(endPos-startPos);
while(STOP>=0 && dist <= STOP &&acc<1.0f)
{
   float diameter = cone_ratio*dist;
   float3 ray = startPos + dir*dist;
   float color = gVoxelList.SampleLevel(Filter, ray / gDim + 0.5f / gDim, dist).a;
   acc += (1 - acc)*color;
   dist += gVoxelSize;
}
return (1-acc);
```

4.5 本章小结

本章主要描述了渲染框架集成的核心算法：实时全局光照的具体实现原理和实现步骤，算法在 GPU 渲染管线上进行运算，按功能划分为体素化、间接光照和直接光照，其中直接光照模型为基于物理的光照渲染模型，间接光照由保存体素化后的空间数据得到，并进行光线追踪.本算法可以达到可以交互的渲染帧率.

第 5 章 结果与比较

本文以 SVO 算法为核心搭建出了一套可以进行复用和拓展的渲染框架，在该框架上集成了基于 GPU 的体素化算法、基于体素的全局光照算法等高级渲染技术.用本软件框架渲染的画面达到了一定的画面质量和可交互的实时帧率.

本章主要呈现了本框架软件的运行结果，下面将分别测试三类数据：

1. 体素化效率测试.基于本框架软件实现的基于 GPU 的体素化算法效率测试，分别测试体素化细分网格为大小为 64、128、256 时在本框架上的渲染速度.

2. 直接光照渲染结果对比.基于本框架软件实现的基于物理的直接光照渲染结果横向比较，对比测试平台为 DirectX 的固定管线和 3DSMAX 的 Mental-Ray 渲染器.

3. 合成结果横向比较，对比测试平台为本框架的最终渲染结果与 Unity 的 Enlighten 全局光照渲染器与 3DSMAX 的 Mental-Ray 全局光照渲染器.

本章节的硬件测试环境如表 5-1 所示：

表 5-1 测试硬件平台参数

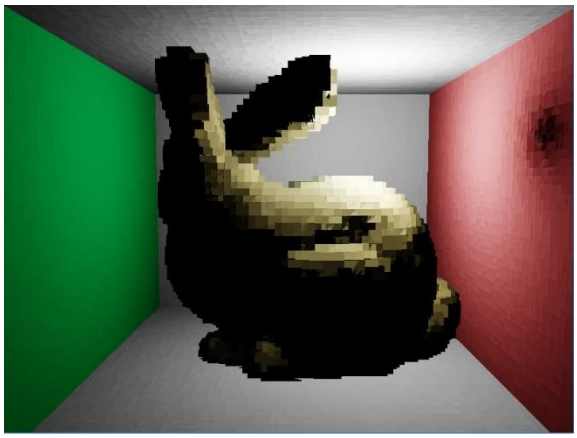
CPU	显卡	内存
INTEL Core I5	NVIDIA GTX1050 Ti	8G

本文的实验数据是 Cornell Box 模型与斯坦福大学的 Bunny 模型，共计 4977 个三角形面片，14934 个顶点，使用了一个动态的点光源，渲染图像大小为 800x600.

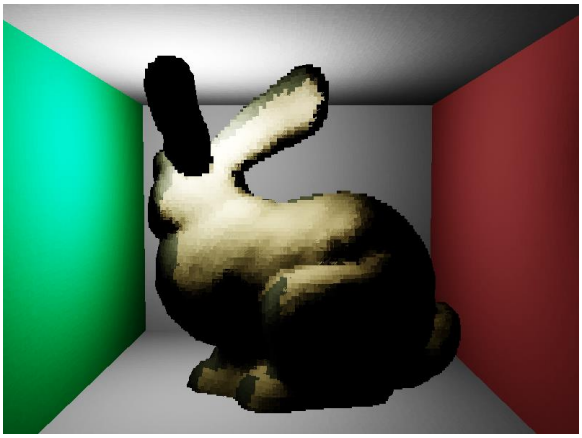
5.1 体素化测试

本框架实现了基于 GPU 的场景体素化，体素化时使用一个动态点光源.并将体素化结果进行可视化.本节展示了基于 GPU 的动态光源体素化结果.

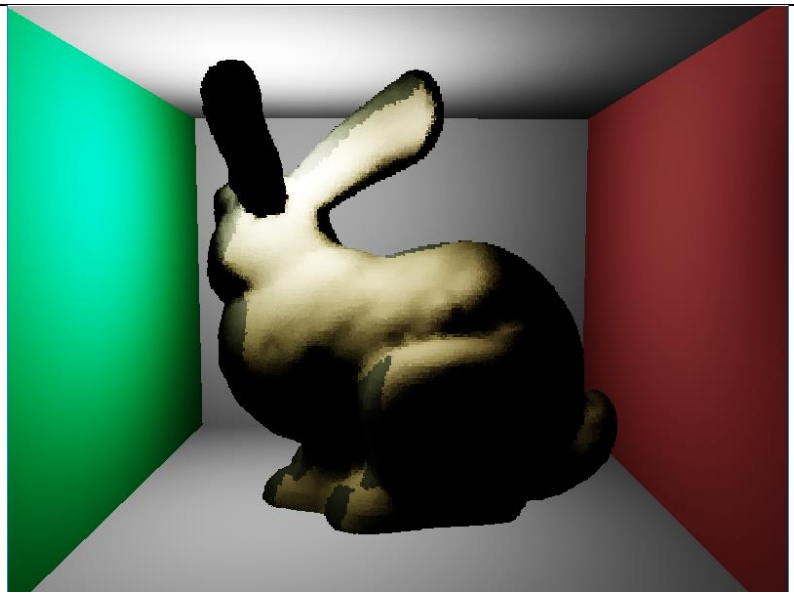
下图 5-1 展示了 bunny 模型在一个动态光源下的体素化结果：



(b) 体素分辨率为 64



(b) 体素分辨率为 128



(c) 体素分辨率为 256

图 5-1 bunny 模型的体素化结果

下表 5-2 展示了本框架实现的体素化算法的效率：

表 5-2 体素化结果

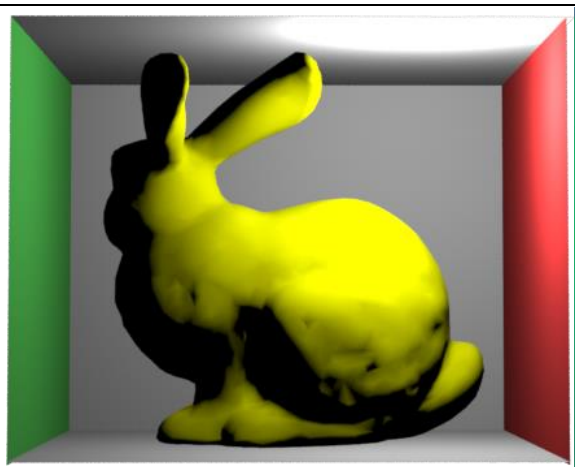
分辨率	所需时间/每帧	FPS
64^3	2 毫秒	602
128^3	9.5 毫秒	125
256^3	58.8 毫秒	17.5

由上表可见在体素化方面体素网格分辨率为 128 最佳.若体素网格分辨率低于 128 则渲染速度足够快，但渲染质量不佳.高于 128 则渲染速度过慢，不能进行流畅的可交互操作.

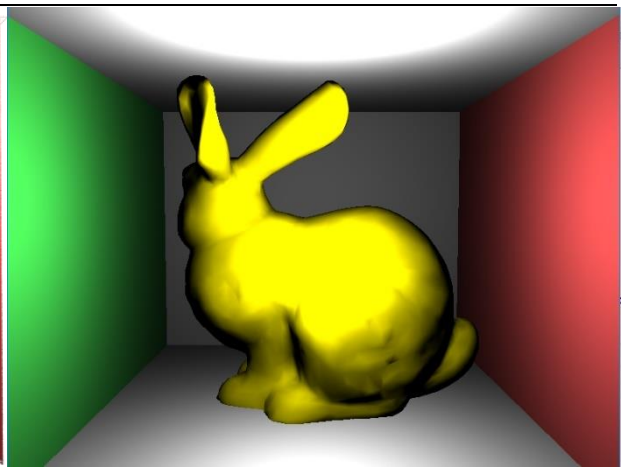
5.2 直接光照结果比较

本框架软件实现了基于物理的直接光照(physically-based lighting) 以及基于物理的材质，双向反射分布函数(BRDF).

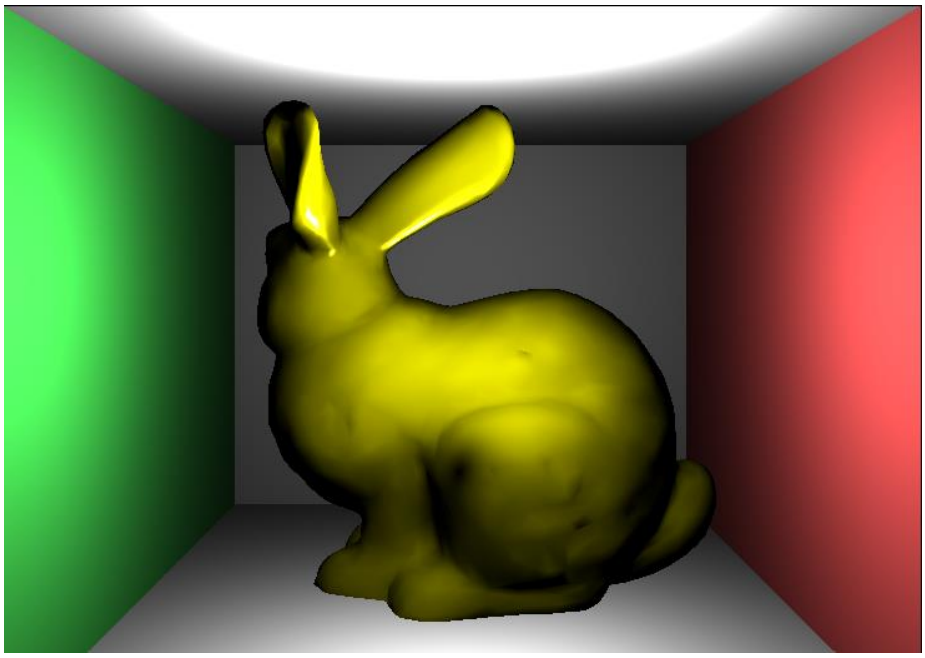
下图 5-3 为本框架实现的直接光照结果与 DirectX 固定管线(局部光照模型)中的直接光照结果、Mental-Ray 中的直接光照结果对比：



(a) Mental-Ray 的直接光照结果



(b) DirectX 固定管线的直接光照结果



(c) 本框架的直接光照结果

图 5-2 直接光照结果

下表 5-3 为本框架实现的直接光照渲染效率与 DirectX 固定管线的直接光照渲染效率对比：

表 5-3 直接光照结果对比

测试程序	所需时间/每帧	FPS
本框架	9.17 毫秒	109
Mental-Ray	2 秒	0.5
DirectX 固定管线	2.6 毫秒	380

由上图和上表分析可得，在直接光照中固定管线与本框架实现的 BRDF 渲染方法皆可以达到实时运行的速率，而 Mental-Ray 由于采取的是光线追踪方法，无法进行实时运算.在渲染质量方面，Mental-Ray 和本框架实现的直接光照皆可以模拟出柔和的光线和有过渡的表面阴影效果相比之下，DirectX 固定管线的过渡较弱.

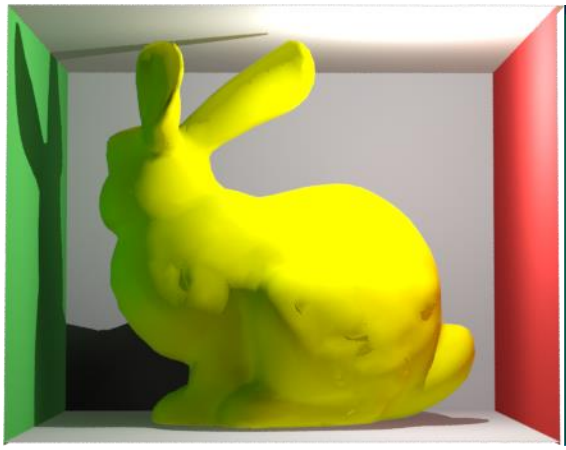
5.3 最终渲染结果比较

本框架实现了基于 SVO 算法的实时全局光照算法，其中有 1 个动态光源，体素化网格分辨率为 128，在直接光照部分采用了基于物理的渲染，在间接光照部分采用了基于体素的锥体跟踪方法.跟踪间接光照时追踪了 9 个用于追踪漫反射的 Diffuse Cone，一个用于追踪阴影的 Shadow Cone.

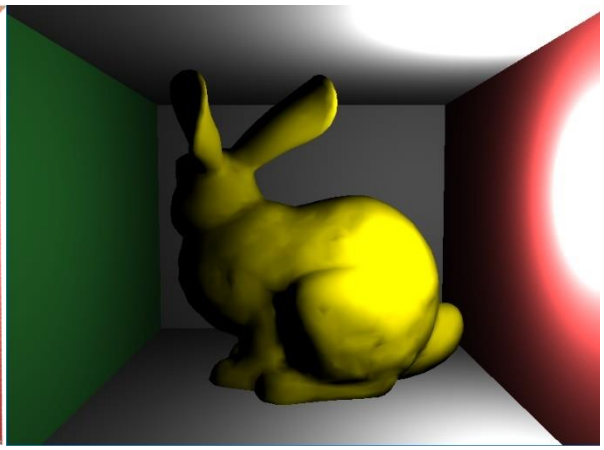
本节展示了在相同场景下，本框架的最终渲染结果与 DirectX 中固定渲染管线、3DSMAX 中 Mental Ray 的渲染质量和渲染速度的比较.图像大小都为 800x600.

为本框架实现的基于物理渲染的直接光照结果与DirectX固定管线中的渲染结果对比：

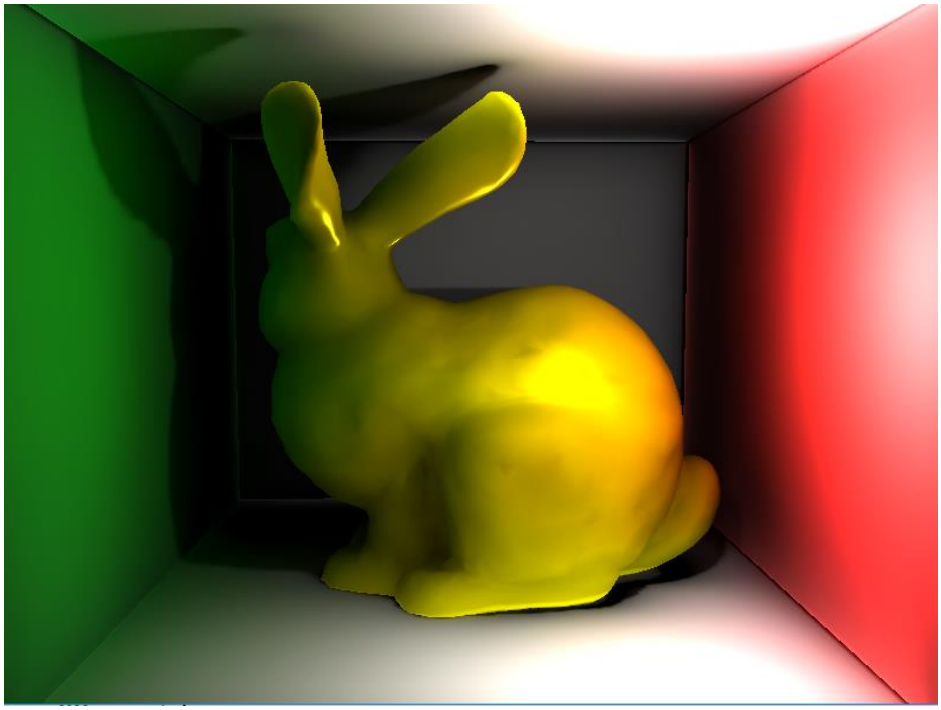
下图 5-3 为三个渲染平台的最终渲染结果画面质量比较：



(a) Mental-ray 渲染结果



(b) DirectX 固定管线渲染结果



(c) 本框架渲染结果

图 5-3 最终渲染结果

下表 5-3 展示了三个渲染平台的渲染效率：

表 5-3 渲染效率结果对比

测试程序	所需时间/每帧	FPS
本框架	31．25 毫秒	32
Mental-Ray	5 秒	0.2
DirectX 固定管线	2.6 毫秒	380

根据上图和上表的结果，可以分析到在渲染速度方面：

DirectX 固定渲染管线由于采用的是不可编程的固定渲染方式，无法添加全局光照效果，速度最快，本框架实现的基于 SVO 的实时渲染算法次之，可以达到实时可交互的全局光照效果，Mental-Ray 渲染引擎最慢，无法实时渲染画面.

在渲染质量方面：

DirectX 固定渲染管线由于使用的是局部光照模型，渲染质量最差，没有阴影以及间接光照.本框架的实现算法在间接反射方面有着不俗的表现，但在阴影和材质方面表现仍然有待改进.Mental-Ray 的全部渲染算法都采用基于物理的渲染，渲染质量最好.

5.4 本章小结

本章首先分析了本框架的体素化渲染结果，测试了分辨率不同时本框架的体素化结果。其次比较了不同平台上直接光照的渲染质量和渲染效率，最后与 3dsmax 的 Mental-Ray 渲染器中、DirectX 固定管线的最终渲染效果进行对比，并且分析了每个平台渲染效果的优缺点.

第 6 章 结论与展望

6.1 结论

全局光照算法一直是计算机图形的重要研究方向，由于硬件限制，全局光照算法虽然已经在动画产业进行了广泛应用，在游戏领域以及其他需要实时帧率的领域仍然属于实验性阶段.

全局光照算法有两个研究方向，第一个是在动画领域等不需要实时渲染的领域，每渲染一帧可能需要几个小时，此领域的目标是让基于物理的渲染进行更加精确的模拟.第二个领域是在游戏等需要实时高帧率的领域，在这个方向上的目标是如何平衡画面与速度达到可接受的实时渲染帧率.同时由于产业中游戏场景的庞大，一个游戏场景中还包括动画，UI 等占用内存的模块，使得实时全局光照的广泛应用更加困难.

本文介绍的基于体素化的实时全局光照算法完全在 GPU 上实现，可以达到可交互的渲染帧率，与传统的 CPU 渲染器相比，性能得到了大幅度提升.

文章还介绍了此渲染系统的设计与实现原理，渲染系统是实现各种高级渲染技术的必不可少的平台，现在的主流开源游戏引擎由于工程量巨大并且搭载了物理引擎，粒子系统等完整的引擎模块，初学者难以借鉴，本渲染框架为实验性平台，具有一定的参考意义.本文还介绍总结了渲染技术的基础核心知识，包括 3D 流水线，基于物理的渲染等.

6.2 不足之处及未来展望

渲染引擎是一个非常庞大并且复杂的架构，主流的游戏引擎中的渲染引擎有巨大的结构，需要非常强的鲁棒性以及可用性和前沿性.本渲染框架为实验性平台，主要实现了实时全局光照算法，并且搭建了一套基于 DirectX11 的可拓展渲染引擎.

由于时间周期和复杂度的原因，本框架还有下面这些改进点：

1. 改进贴图系统和材质系统以改进材质系统，以自动读入 obj 材质格式.
2. 改进基于 GPU 的体素化算法，使得模型的体素化可以更完整.
3. 完善前沿性的渲染功能，集成更多的高级渲染算法.
4. 增加粒子系统以及动画系统.
5. 完善底层数据传送，使用通用计算平台以加快 CPU 与 GPU 的通信.

根据本课题的研究，基于体素的实时全局光照算法可以满足一定质量的画面表现，有着不俗的间接光照效果，同时在渲染上可以达到可交互的帧率，但速度仍然需要加快，在未来本课题会继续拓展实时渲染引擎的框架，让它集成更多的渲染模块以及高级算法. 同时在实时全局光照算法方面会在本课题的基础上继续探索基于体素的实时渲染算法和基于 GPU 的光线追踪算法.

参考文献

[1] 中国音数协游戏工委. 2016 年中国游戏产业报告：摘要版[M]. 北京：中国书籍出版社，2016：11-15.

[2] Jason G. Game Engine Architecture[M]. A K Peters/CRC Press, 2009: 100- 125.

[3] Yibing J. The Process of Creating Volumetric-based Materials in Uncharted 4[R]. SIGGRAPH 2016 Course, ACM SIGGRAPH, 2016: 16-36.

[4] Carlos G O. Rendering Rapids in Uncharted 4[R]. SIGGRAPH 2016 Course: Advances in Real-Time Rendering in Games, ACM SIGGRAPH, 2016: 13-22.

[5] Matt P, Wenzel J, Greg H. Physically based Rendering: From theory to implementation[M]. Morgan Kaufmann, 2016: 240-248.

[6] Tomas A M, Eric H, Naty H. Real-Time Rendering[M]. A K Peters/CRC Press, 2008: 101-111.

[7] 冯乐乐. Unity Shader 入门精要[M].北京：人民邮电出版社，2016：30-40.

[8] Frank L. Introduction to 3D Game Programming with DirectX 11[M]. Mercury Learning & Information, 2012: 300-325.

[9] Peter S. Ray Tracing in One Weekend[M]. Amazon Digital Services, 2016:20-22.

[10] James T K. The rendering equation[C]. Computer Graphics(Proceedings of SIGGRAPH ' 86)20,4, 1986:143-150.

[11] Alexander K. Instant radiosity[C]. SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques,1997.

[12] Eric S C. progressive multi-pass method for global illumination[C]. A progressive multi-pass method for global illumination. Proceedings of SIGGRAPH 91, ACM 1991.

[13] Henrik W J. Global illumination using photon maps[C]. Rendering Techniques'96, 1996: 21-30.

[14] Cook L., Torrancek E, A reflectance model forcomputer graphics[C]. In SIGGRAPH 81 ACM, 1981: 307–316.

[15] Crassin, C, Neyret F. Interactive Indirect Illumination Using Voxel Cone Tracing[J]. Computer Graphics Forum (Proc. of Pacific Graphics 2011), 2011.

[16] Kaplanyan A, Dachsbacher C. Cascaded light propagation volumes for real-time indirect illumination[C] ACM SIGGRAPH Symposium on Interactive 3d Graphics and Games. ACM, 2010:99-107.

[17] Hasselgren J, Akenine-Möller T, Ohlsson L. Conservative rasterization[J]. Gpu Gems, 2005.

[18] Brabec S, Annen T, Seidel H P. Practical Shadow Mapping[J]. Journal of Graphics Tools, 2002, 7(4):9-18.

[19] Schwarz M, Seidel H P. Fast parallel surface and solid voxelization on GPUs[J]. Acm Transactions on Graphics, 2010, 29(6):179.

[20] Crassin C, Neyret F, Sainz M, et al. Interactive indirect illumination using voxel-based cone tracing: an insight[J]. Computer Graphics Forum, 2011, 30(7):1921–1930.

[21] Phong B T. Illumination for computer generated pictures[M] Seminal graphics. ACM, 1998.

[22] Blinn J F. Models of light reflection for computer synthesized pictures[J]. Acm

Siggraph Computer Graphics, 1977, 11(2):192-198.

[23] Nicodemus F E. Directional Reflectance and Emissivity of an Opaque Surface[J]. Applied Optics, 1965, 4(7):767-773.

[24] Walter B, Marschner S R, Li H, et al. Microfacet Models for Refraction through Rough Surfaces.[C] Eurographics Symposium on Rendering Techniques, Grenoble, France. DBLP, 2007:195-206.

[25] Schlick C. An Inexpensive BRDF Model for Physically - based Rendering[J]. Computer Graphics Forum, 1994, 13(3):233-246.

[26] Torrance K E, Sparrow E M. Theory for off-specular reflection from roughened surfaces[M] Radiometry. Jones and Bartlett Publishers, Inc. 1992:1105-1114.

致　谢

四年的大学生活即将结束，回想这四年我收获的不仅是知识，更是认真对待生活、学习的态度.在这期间我得到了很多人热心的帮助，尤其是在做毕业设计的这段时间，非常感谢各位老师和同学在这段时间的热情帮助.

首先要感谢我的导师张军老师，本文是在张军副教授的指导下完成的，张老师在毕设的每个阶段都给了我他的指导建议，在我有问题的时候都悉心指导，提出宝贵的参考意见.在大学四年里，张老师不仅是我的毕设指导老师，也是我的编程入门老师，从他身上我不止学习到了编程的思维，最重要的是学习到了对于技术真诚、执着的态度.张老师是我在学习和工作上的榜样，是一个对专业知识有着深刻理解和追求的良师益友，每次与张老师的交流都能使我获益匪浅.

除了我的指导老师之外，我还要感谢赵燕老师在艺术方面给我的参考意见以及对我生活上的关心，赵燕老师为学生着想的态度让我十分触动.感谢毕设期间曹钰同学对我的帮助，在这段时间里我们互相帮助和鼓励，感谢你这段时间给我的真诚意见.感谢杜文婷学姐、官兵兵学长、艺术系与我交流过的同学在我有技术细节问题的时候给我的悉心解答.感谢我的好友顾佳枫以及我的父母，你们是我生活和精神上的坚实后盾.

最后特别要感谢的是数字媒体学院的每一位老师们，感谢你们的言传身教和无私给予.大学四年，每一天都离不开你们在课上课下辛勤的授业和回答问题的身影，你们在这四年里教给了我许许多多专业知识和对待科学的方法，这都是我累积下来的财富，在这里向各位老师表达深深地谢意.