

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得浙江大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名: 阮海锐 签字日期: 2015 年 3 月 27 日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密的学位论文在解密后适用本授权书)

学位论文作者签名: 阮海锐

导师签名:

周君

签字日期: 2015 年 3 月 27 日

签字日期: 2015 年 3 月 27 日



Y2881930

摘要

真实的照片级画面质量一直都是实时渲染所追求的终极目标。近几年，国外次时代游戏已经可以达到非常逼真的画面效果，而国内游戏的画面效果却一直表现平平，难以满足玩家日益增长的高画质游戏需求。虽然国外已经有很多优质的游戏引擎，但这些引擎大都用于家庭主机上的游戏开发，在国内流行的网络游戏上表现得水土不服，这才导致各大游戏公司都在研发自己的游戏引擎。而作为游戏引擎中的核心部分，实时渲染决定着一款游戏画面质量的好坏。国内自研游戏引擎在网络服务、逻辑脚本等其他方面表现都不错，唯独高质量的实时渲染，一直都是软肋。

本论文旨在架构设计一个高质量的实时渲染引擎，能够充分利用现代图形处理器的高级特性，并在此框架上集成目前最先进的实时渲染算法。这些高级的渲染算法能够很大程度上提高游戏的画面质量，是目前国内游戏引擎最需要集成的功能。

本文的研究成果主要表现在以下几方面：

- 1) 设计了一个跨 DirectX 11 和 OpenGL 的渲染引擎，能够充分利用现代 GPU 的所有特性，并设计了灵活的材质特效脚本。
- 2) 完成了纹理空间的次表面散射算法，能够逼真地渲染游戏人物角色的脸部皮肤，提高游戏画面质量。
- 3) 设计了一套水面和流体渲染系统，能够渲染游戏中水面、瀑布、喷泉等特殊场景。
- 4) 提出了一套 Tile Based 的渲染框架，能够高效处理大量动态光源的渲染。
- 5) 集成了目前最先进的动态光影算法，包括阴影贴图、环境遮蔽和快速全局光照。

关键词： 实时渲染，次表面散射，延迟渲染，全局光照

Abstract

Photorealistic image quality has always been the ultimate goal of real-time rendering. In recent years, foreign Next-Gen games already reached a very realistic image effect, while the image effect of domestic games is still mediocre. It is difficult to meet the growing needs of the player's high-quality gaming. Although there are many excellent foreign game engines, most of these game engines is developed for console games, not suitable for the development of domestic online games. This leads to each big game companies develop their own game engine. As a core part of the game engine, real-time rendering determines the image quality of a game. Domestic self-developed game engine works well on network services, logic scripts, etc. with the exception of high-quality real-time rendering, has been weakness.

This paper aims to design a high-quality real-time rendering engine, which can take full advantage of the advanced features of modern graphics processors. In addition, we integrates the most advanced real-time rendering algorithms on this framework. These advanced rendering algorithms that domestic game engine needs most can greatly improve the image quality of the game.

The main contributions of the thesis are listed as follows.

- 1) Design a rendering engine which can be used both on DirectX 11 and OpenGL, takes full advantage of all the features of a modern GPU. Also Design a flexible material effect script.
- 2) Complete the texture space sub-surface scattering algorithm which improves the effect of game characters facial skin.
- 3) Propose a framework for the rendering of water and fluid. It can be use to render waterfall, splash etc.
- 4) Propose a tile based rendering framework for large number of dynamic lights.
- 5) Integration of the cutting-edge rendering algorithm on shadow, ambient occlusion and global illumination.

Keywords: Real-Time Rendering, Sub-surface Scattering, Deferred Shading, Global Illumination

目录

摘要	i
Abstract	ii
第1章 绪论	1
1.1 研究背景与意义	1
1.2 国内外研究现状	1
1.2.1 国内研究现状	1
1.2.2 国外研究现状	2
1.3 本文工作内容概述	3
1.4 本章小结	4
第2章 相关理论与技术	5
2.1 3D 图形编程	5
2.1.1 渲染管线	5
2.1.2 通用计算管线	7
2.1.3 图形资源	7
2.2 基于物理的渲染	9
2.2.1 双向反射分布函数	9
2.2.2 法向量分布函数	10
2.2.3 菲涅尔反射	11
2.2.4 几何遮挡项	12
2.2.5 渲染方程	13
2.3 本章小结	13
第3章 渲染引擎架构实现	14
3.1 跨 DirectX 11 和 OpenGL 的渲染系统	15
3.2 材质和特效系统	16
3.2.1 特效系统	17
3.2.2 材质系统	19
3.3 场景管理	20
3.4 动画系统	22

3.4.1 蒙皮动画	22
3.4.2 动画混合	24
3.5 GUI 系统	24
3.6 素材管线	26
3.6.1 三维网格模型和动画	27
3.6.2 纹理和材质	27
3.7 本章小结	28
第4章 皮肤和流体	29
4.1 人脸皮肤渲染	29
4.1.1 人脸皮肤的光学特性	29
4.1.2 基于纹理空间的次表面散射	30
4.1.3 基于 Kelemen/Szirmay-Kalos BRDF 的反射模型	32
4.1.4 透光效果	35
4.2 水面及流体渲染	37
4.2.1 水面渲染	37
4.2.2 屏幕空间流体渲染	38
4.3 本章小结	41
第5章 Tile Based 渲染	42
5.1 延迟渲染	42
5.2 Tile Based 渲染	43
5.2.1 Tiled Light Culling	44
5.2.2 Tile Based 延迟渲染	46
5.2.3 Tile Based 前向渲染	48
5.3 渲染效率对比	49
5.4 本章小结	50
第6章 阴影环境遮蔽和全局光照	51
6.1 阴影渲染	51
6.1.1 Cascaded Shadow Maps	51
6.1.2 Shadow Maps Filtering	54
6.2 环境遮蔽	56
6.2.1 Alchemy AO	56

6.2.2 HBAO+	58
6.3 基于 Deep G-Buffers 的快速全局光照算法	59
6.3.1 Deep G-Buffers	60
6.3.2 间接光照	61
6.4 本章小结	63
第 7 章 总结与展望	65
7.1 工作总结	65
7.2 未来展望	66
参考文献	67
致谢	71
作者简历	72

图目录

图 2.1 DirectX 11 图形渲染管线.....	6
图 2.2 纹理类型	8
图 3.1 渲染引擎架构图	14
图 3.2 渲染 API 类图	15
图 3.3 特效和渲染管线图	17
图 3.4 特效脚本结构	18
图 3.5 材质脚本结构图	20
图 3.6 场景图结构	21
图 3.7 骨骼动画姿势	23
图 3.8 引擎动画效果	24
图 3.9 GUI 控件效果	25
图 3.10 GUI 系统类图	25
图 3.11 GUI 皮肤和九宫格图	26
图 3.12 素材导出图	28
图 4.1 三层皮肤模型	30
图 4.2 光线扩散能力曲线	31
图 4.3 高斯拟合曲线	31
图 4.4 纹理空间次表面散射算法流程图	32
图 4.5 预计算纹理查找表	34
图 4.6 人脸皮肤纹理空间次表面散射效果图	35
图 4.7 透射原理图	36
图 4.8 透光效果对比图	37
图 4.9 水面渲染效果图	38
图 4.10 屏幕空间流体算法原理图	39
图 4.11 流体渲染效果图	40
图 5.1 Light Cull 的 2D 展示	46
图 5.2 Tile Based Deferred Shading 效果图	47

图 5.3 AMD Leo Demo 画面	48
图 5.4 三种渲染方法的效率对比图	50
图 6.1 阴影贴图算法锯齿	52
图 6.2 阴影贴图算法透视锯齿	52
图 6.3 PSSM frustum 划分图.....	53
图 6.4 CSM 阴影贴图选择.....	55
图 6.5 CSM 阴影算法效果图	55
图 6.6 Alchemy AO 在 Sponza 场景下的效果图	58
图 6.7 HBAO 算法图	58
图 6.8 HBAO+在 Sponza 场景下的效果图	59
图 6.9 裁剪空间三维场景的表示	61
图 6.10 Deep G-Buffer 格式	61
图 6.11 Sponza 场景下的全局光照效果.....	63
图 6.12 室内房间场景全局光照效果	63

第1章 绪论

1.1 研究背景与意义

目前计算机图像学在影视制作、虚拟现实、科学计算可视化、地理信息系统、电子游戏等各个方面都有着广泛的应用。其中实时渲染技术在电子游戏中起着关键作用，由于游戏行业的巨大产值，电子游戏的发展某种程度上也在推动图形硬件的发展。随着图形硬件的迅速发展，欧美游戏制作商在追求真实感、高画质的游戏道路上不断前进。而国内的游戏市场一直停留在大型网络游戏阶段，其游戏画质相比于欧美单机游戏，已经落后很多。此外，近些年国内网络游戏市场的增长速度已经呈现放缓的趋势，但游戏玩家对于高画质游戏的追求热度依旧不减，期待着能有高品质的国产游戏出现。

从技术上讲，决定一个游戏画质优劣的关键是游戏引擎技术^[1]，尤其是实时渲染技术^[2]。我国虽然有着全世界最大的网络游戏市场，但国产游戏引擎在实时渲染上仍然落后日韩、欧美国家很多，要么就是直接引进国外的游戏引擎。在赶超国际水平的过程中，开发高质量的实时渲染引擎仍然是必经之路。而且，目前也急需高质量的实时渲染引擎来刺激国内即将萎缩的网络游戏市场。

此外，今年微软和索尼都在国内发售了家用游戏机制造商，Xbox One 和 Play Station 4。相信不久的将来，国内单机游戏市场也将发展起来，这就更需要高质量的实时渲染引擎。

1.2 国内外研究现状

1.2.1 国内研究现状

相比于国外的游戏引擎发展，国内基本上没有只做游戏引擎服务的公司。各大网络游戏公司都会根据实际项目需求开发自己的游戏引擎。其中渲染效果比较好的有，腾讯公司自主研发的 Quicksilver 引擎，其推出的游戏《天涯明月刀》

刷新了国产 3D 网游图形、技术和画面表现能力的上限。此外还有引擎功能比较完善的《剑网三》游戏引擎，该游戏引擎由金山旗下西山居游戏工作室开发，当年是国家“863”计划支持的项目，在客户端和工具链的集成上做的比较完善。相比于国外的游戏引擎，国内游戏引擎在工具链的完整性上要相对薄弱一点。

此外，还有些个人开发的渲染引擎，其中最知名的当属由微软亚研院龚敏敏开发的 KlayGE 渲染引擎。不同于公司开发的游戏引擎，KlayGE 更加专注于渲染，使用到的技术也紧跟国外引擎的步伐，是目前国内最先进的渲染引擎之一。

国内网络游戏行业发展虽然迅速，但在游戏引擎方面，跟欧美国家的差距还是很巨大。主要原因总结起来有以下几个：

- 1) 国内游戏产业追求快速利润，很少有公司有能力也愿意投入巨大的成本来支持游戏引擎的研发。
- 2) 国内没有家用主机游戏市场，也基本上没有单机游戏。而国外有 XBOX、PS4 等主机游戏，国外很多高品质的游戏都是在主机游戏上发售。个人觉得国外主机游戏市场是驱动引擎发展的一个关键因素。

1.2.2 国外研究现状

经过近 20 年的发展，国外的游戏行业已经非常成熟，并且分工明细，游戏开发中用到的某个特定功能组件都可以单独分离出来提供服务，包括物体引擎、动画系统，GUI 系统等。世界上第一款商用授权游戏引擎是 90 年代 Id Software 公司研发的 Id Tech 1 引擎，由著名的 3D 引擎程序员约翰卡马克主导开发。

目前国外专门提供游戏引擎授权服务的公司比较多。美国 Epic 公司的 Unreal 系列引擎，国外很多主机游戏都是采用 Unreal 引擎开发，如《质量效应》、《战争机器》等。还有德国 Crytek 公司的 CryEngine 引擎，用这款引擎开发的《孤岛危机》曾一度是 PC 游戏最高画质的代表，被玩家称之为“显卡危机”。此外，目前在移动平台最受欢迎的 Unity 游戏引擎，正在以交互式可视化图形开发的方式改变游戏制作的过程，让游戏制作变得更加容易。

除了上面这些提供通用游戏引擎服务的公司外，还有些很多游戏工作室自己

内部使用的游戏引擎，如顽皮狗公司的《最后生还者》、《神秘海域》游戏使用的引擎，美国艺电公司（EA）旗下 DICE 工作室的《战地》系列游戏引擎 Frostbite Engine。这些并不对外公开的游戏引擎在技术上可能更加先进。

同样，国外也有很多开源的渲染引擎，如 OGRE^[3]、irrlicht、OpenSceneGraph 等。相比于上面的游戏引擎，这些开源渲染引擎往往只提供一个渲染框架，开发者可以根据需要集成相应功能的其他组件来开发游戏，如国内搜狐畅游公司的《天龙八部》游戏就是基于 OGRE 渲染引擎开发。

1.3 本文工作内容概述

本文的研究是基于 DirectX 11 和 OpenGL 的渲染引擎设计和实现，并在这个基本框架之上探索了当下最先进的渲染技术，这些渲染算法是目前国内游戏引擎最迫切需要的。本文共分 7 章，章节结构分别如下：

第一章，绪论，介绍本论文的研究背景和意义，以及目前这方面国内外研究现状。

第二章，介绍了基本的 3D 渲染编程，以及目前国外游戏引擎都在使用的基于物理的渲染模型，这是真实感渲染的理论基础。

第三章，整个实时渲染引擎的架构，分别阐述了引擎各个基本模块的功能和实现。

第四章，介绍了人脸皮肤渲染即次表面散射效果，和屏幕空间的流体渲染方法。

第五章，提出了 Tile Based 的渲染框架，包括延迟渲染和前向渲染。

第六章，介绍了在实时渲染中常用的动态光影算法，包括阴影，环境遮蔽和快速全局光照算法。

第七章，总结全文，以及展望未来工作。

本论文的目标是开发一个跨渲染 API 的实时渲染引擎，能够高效的利用现代 GPU 的高级特性探索一些先进的渲染算法。这些渲染算法目前国内的游戏都还未开始使用，如 Tile Based 的延迟渲染。国内游戏引擎基本上都是使用传统的延迟

着色技术，只能支持少量的动态光源。此外，本论文还在延迟渲染的框架上，给出了高质量的环境遮蔽算法和快速全局光照算法。集成这些技术能够很大程度上改善国内网络游戏的画面质量。本文实现的实时渲染引擎已经开源，项目代码托管在 Github 上。

1.4 本章小结

本章节为绪论，首先介绍了本文研究课题的背景和意义，阐述了开发一个高品质的实时渲染引擎的重要性。并对当下国内外游戏引擎和渲染引擎的发展现状做了简要介绍，最后介绍了本文的主要工作和对章节结构安排进行了说明。

第2章 相关理论与技术

2.1 3D 图形编程

目前主流的图形 API 主要有微软领导的 DirectX 和 Khronos Group 领导的 OpenGL。DirectX 只能在微软旗下的操作系统上使用，如 Microsoft Windows、Microsoft XBOX、Windows Phone 上。而 OpenGL 则是平台无关的图形 API，可以运行任何操作系统上，目前在移动平台上使用的是 OpenGL ES。DirectX 在桌面 PC、家用游戏机上的游戏领域占主导地位，而目前移动平台上的游戏开发基本上以 OpenGL ES 为主。

这两种图形 API 都已经有多年的发展历史，各自有好几个版本。市面上主流的 DirectX 版本有 DirectX 9.0c 和 DirectX 11，OpenGL 则是 3.x 版本和 4.x 版本。随着图形处理器性能的增加，新版本图形 API 都会在老版本的基础上增加新的功能，来获取最新的硬件特性。本渲染引擎的定位是新一代实时渲染引擎，希望能充分利用最新的图形硬件特性，所以选择了 DirectX 11 和 OpenGL 4.x 为底层渲染 API，未来还将支持微软即将发布的 DirectX 12。

不管是使用哪种 API，目前最新的 DirectX 和 OpenGL 在功能上基本相同，而且 API 的抽象也差不多，主要包括图形资源，渲染状态、渲染管线、通用计算管线等。下面介绍基本的图形 API 编程。

2.1.1 渲染管线

渲染管线，又称图形管线，是指把三维场景资源转换成二维图像的一系列过程，如图 2.1 所示。整个渲染管线有许多不同的阶段组成。数据经过一个阶段的处理后传递给下一个阶段，虽然现代图形处理器的性能增长迅速，渲染管线中的大部分都是可编程的，但仍旧有固定功能的部分。我们可以把渲染管线分为固定功能部分和可编程部分，分别对应图形 API 中的抽象，以 DirectX 11 为参考，

在 OpenGL 中也是类似的。所谓固定功能是指 GPU 在这个阶段只能按照固定的规则处理数据流，对应到 API 中就是设置对应渲染状态，如深度比较函数、透明混合方式等。而可编程是指 GPU 在这个阶段能以可编程着色器的方式处理数据，如顶点着色器（Vertex Shader）、像素着色器（Pixel Shader）等。

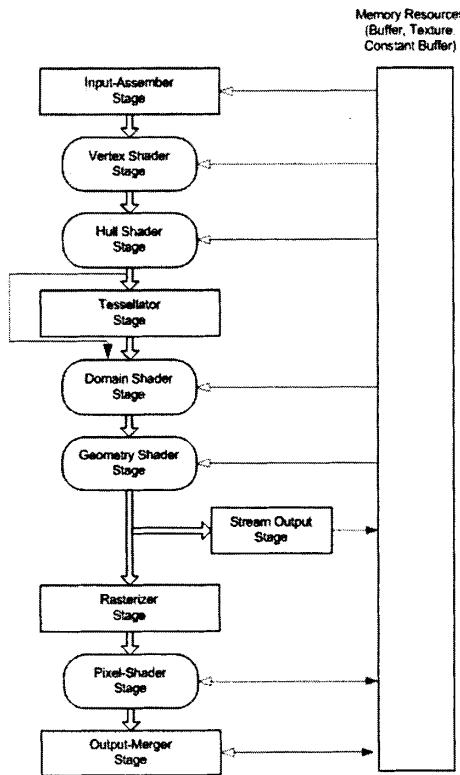


图 2.1 DirectX 11 图形渲染管线¹

其中固定功能阶段有 Input-Assembler、Tessellator、Rasterizer 和 Output-Merger。在 DirectX 11 中固定功能阶段都有提供相应的接口让你设置状态。在 Input-Assembler 阶段，我们需要设置管线的输入，包括设置顶点缓存和索引缓存，并且设置绘制的图元类型，如点、线、三角形等。还需要匹配顶点缓存的数据布局(Layout)，这在 DirectX 11 中通过 InputLayout 设置的，在 OpenGL 中通过 Vertex

¹ Windows DirectX Graphics Documentation

Array Object (VAO) 实现的。Rasterizer 阶段需要设置 RasterizerState，即光栅化规则。Output-Merger 是最后一个阶段，在像素着色器生成像素颜色后，由 DepthStencilState 来决定是否需要输出该像素，BlendState 来决定最终输出目标中的像素值，即合并像素着色器输出的颜色值和已经储存在颜色缓存里的颜色值。

可编程阶段就是着色器，需要使用 GPU 着色代码编写，DirectX 是用 HLSL，OpenGL 使用 GLSL。可编程着色器包括顶点着色器、曲面细分着色器、几何着色器、像素着色器。其中曲面细分着色器在 DirectX 11 中分为 Hull 着色器和 Domain 着色器，分别对应 OpenGL 中的 Tessellation Control 着色器和 Tessellation Evaluation 着色器。

2.1.2 通用计算管线

除了上面介绍的渲染管线外，DirectX 11 和 OpenGL 4.3 还引入了 Compute Shader。Compute Shader 的出现把图形渲染 API 带入了 GPU 通用计算 (GPGPU) 的世界，类似于 Nvidia 的 GPU 编程语言 CUDA，Compute Shader 不仅处理普通的图形渲染任务，还能运用在物理模拟上，为游戏提供更加自然真实的体验，如流体模拟、碰撞检测等。

2.1.3 图形资源

图形资源包括缓存 (Buffers) 和纹理 (Textures)，资源定义了渲染场景的内容来源，包括场景模型的顶点、片面数据，场景材质的纹理贴图等。

1. 缓存

缓存储存了渲染模型所需的几何数据和索引数据，这两类是最常用的缓存，称为顶点缓存 (Vertex Buffer) 和索引缓存 (Index Buffer)。除此之外，另外一种缓存是用来向 GPU 着色器 (Shader) 传递参数的，在着色器代码中定义的全局变量都需要在执行着色器前得到参数值，并且在执行的过程中不能被改变。给这类全局变量提供数据的缓存在 DirectX 中称为常量缓存 (Constant Buffer)，在

OpenGL 中是统一缓存 (Uniform Buffer)。

上面的缓存都是由 CPU 提供数据, GPU 使用数据。GPU 只能是读取这些数据并不能写数据。下面两个缓存是可读写的缓存, 纹理缓存 (Texture Buffer) 和结构体缓存 (Structured Buffer)。结构体缓存在 OpenGL 中称之为 Shader Storage Buffer。相比于常量缓存, 纹理缓存和结构体缓存支持更大的缓存容量。纹理缓存只能储存纹理格式的数据, 而结构体缓存允许自定义数据结构。此外, 在着色器中写数据是比较高级的一种特性, 在 DirectX 11 需要为这类资源创建 Unordered Resource View (UAV)。

2. 纹理

纹理只要用来储存模型的材质贴图, 也可以保存渲染的中间结果, 即渲染到纹理。纹理类型可以分为一维纹理、二维纹理、三维纹理和立方体纹理。每种纹理都支持 Mipmap, 一维、二维纹理还支持纹理数组, 如图 2.2 所示。

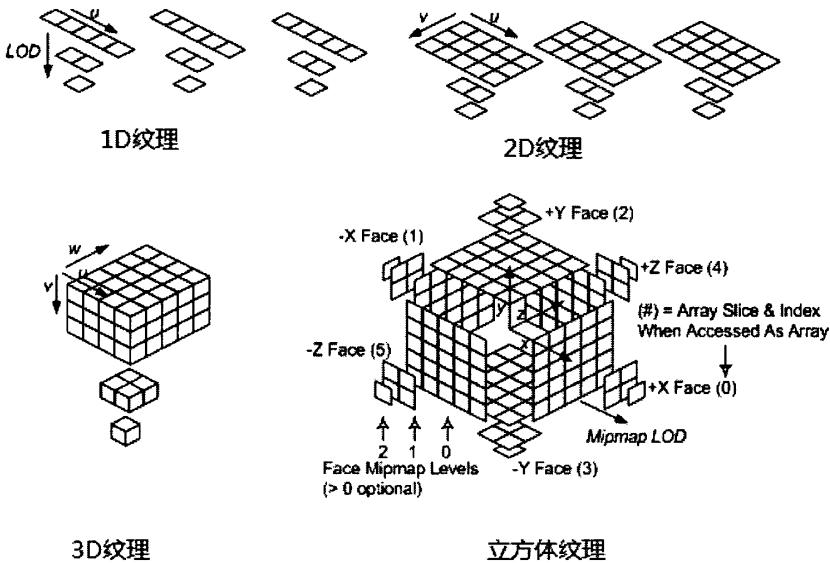


图 2.2 纹理类型²

² Windows DirectX Graphics Documentation

资源只是储存了数据，如何使用、解析这些数据则是通过视图来描述的，即 DirectX 11 中的资源视图，分为 DepthStencilView、RenderTargetView、ShaderResourceView、UnorderedAccessView。虽然 OpenGL 没有这些概念，但在设计跨渲染 API 的渲染引擎时，在资源之上抽象出视图的概念非常重要，况且新版本 OpenGL 也增加了 TextureView 来支持视图的概念。

2.2 基于物理的渲染

基于物理的渲染理论虽然很多年前就已经提出，但近些年才开始在实时渲染中使用。相比于过去的经验模型，基于物理、能量守恒的渲染模型能够渲染出更加真实的图片，而且在材质参数的调节上面也更加容易，美术人员可以花更少的时间完成材质参数的调节。

2.2.1 双向反射分布函数

双向反射分布函数（BRDF）是一种用来描述物体材质的方法，定义了材质在不同光照、观察方向下的对光照的反射函数。在实时渲染的早期阶段，高光的计算都是基于经验模型，如 OpenGL、DirectX 内建的 Phong 模型^[4]、Blinn-Phong 模型^[5]等，这些经验模型渲染出来的物体看上去都具有塑料感，无法渲染出真实感的材质如金属质感。而离线渲染却能渲染出非常逼真的图片，这其中一部分原因是实时渲染没有全局光照，另外更重要的是因为没有使用正确的 BRDF 模型。近几年，实时渲染也开始使用基于物理的渲染，每年的 SIGGRAPH 课程上都有基于物理的渲染课程。基于物理的 BRDF 模型至少需要满足如下几个性质：

- 1) 互逆性，即互换入射和出射光线方向不会改变 BRDF 的值。
- 2) 能量守恒，即要求反射总能量不应该大于入射总能量。

目前绝大部分基于物理的 BRDF 的理论基础是微表面（microfacet）理论^{[6][7]}，其基本思想用无数个很小的微平面去模拟粗糙的物体表面，光线在这些微平面上可以当作是完美的镜面反射，并且只有朝向为半角向量 h （入射和出射方向的一半）的微表面才是有效的。基于微表面理论的 BRDF 可以用如下公式表达

$$f(l, v) = \frac{D(h)F(v, h)G(l, v, h)}{4(n \cdot l)(n \cdot v)} \quad \text{公式 (2.1)}$$

其中

$D(h)$ 是法向量分布项，用来描述朝向为半角向量 h 的微平面分布。

$F(v, h)$ 是菲涅尔项，计算从光线在平滑表面上的反射比例。

$G(l, v, h)$ 是几何遮挡项，用于计算没有被遮挡住的微表面比例，即朝向为 h 并且在 l 和 v 方向上都可见的微表面。

2.2.2 法向量分布函数

物体的微表面并不是均匀分布的，其中大部分微表面的朝向靠近物体表面法向量 n ，少部分偏离法向量。这跟材质的粗糙程度 α 有关系，越是光滑的表面，靠近法向量的微表面越多。极端情况就是镜子，微表面基本上朝向都是法向量。所以 $D(m)$ 其实是个统计意义上的分布函数，计算朝向为 h 的微表面的比重。正确的法向量分布函数应该至少满足下面几个性质：

- 1) 取值范围应该是 0 到无穷大， $0 < D(m) < \infty$
- 2) 对于任何方向 v ，所有微表面的投影面积和应该等于整个表面的投影面积

$$(v \cdot n) = \int D(m)(v \cdot h)d\omega_m,$$

$$\text{对于特殊情况 } v = n, 1 = \int D(m)(n \cdot h)d\omega_m.$$

所以基本上所有法向量分布函数（NDF）都由一个归一化项 $\frac{1}{\pi\alpha^2}$ 。

1. Blinn-Phong^[5]

经典的 Blinn-Phong 高光模型，但我们可以推导更加基于物理的 Blinn-Phong 模型。

$$D_{Blin}(h) = \frac{\alpha_p + 2}{2\pi} (n \cdot h)^{\alpha_p} \quad \text{公式 (2.2)}$$

其中 α_p 是材质的高光指数，跟材质粗糙程度 α 之间的转换见公式 2.3。

$$\alpha = \sqrt{\frac{2}{\alpha_p + 2}} \quad \text{公式 (2.3)}$$

2. Beckmann^[8]

另外一种经常在离线渲染中使用的 NDF 是 Beckmann，Beckmann 分布适用于现实生活中大部分不同粗糙程度的材质。但由于其计算开销昂贵，在实时渲染中不常使用。在实时渲染中可以通过预算算 Beckmann 分布到纹理查找表的方式加速计算，本论文第四章人脸皮肤渲染中的 BRDF 模型使用到了 Beckmann 法向量分布。

$$D_{Beckmann} = \frac{1}{\pi \alpha^2 (n \cdot h)^4} \cdot e^{\left(\frac{(n \cdot h)^2 - 1}{\alpha^2 (n \cdot h)^2}\right)} \quad \text{公式 (2.4)}$$

3. GGX^[9]

近些年各大游戏引擎公司在都追求基于物理的渲染，其中基于 GGX 的法向量分布函数迅速占据了统治地位，如 Unreal 4 游戏引擎就使用该分布函数。GGX 最早是由 Walter 等人提出，可以通过如下公式计算

$$D_{GGX} = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \quad \text{公式 (2.5)}$$

2.2.3 菲涅尔反射

根据物理原理，当光线从折射率为 n_1 的介质进入折射率为 n_2 的介质时，会在物体表面发生反射和折射现象，这种现象称为菲涅尔现象。我们可以通过菲涅尔方程来描述光线在不同介质间发生作用时反射与折射量之间的相互关系。计算多少能量被反射，多少能量被折射。

在实时渲染中，我们一般不直接使用介质的折射率，而是使用 F_0 ，即当入射角为 0 时的反射率 (normal incidence)， F_0 一般就是材质的 Specular 参数。经典的 OpenGL、DirectX 内建光照模型并不考虑菲涅尔现象，我们可以认为这种情况下，菲涅尔项就是 F_0 ，即 $F_{none} = F_0$ 。

大部分游戏引擎使用的是 Schlick^[10]提出的近似估计方法，这种方法基本上在基于物理的实时渲染中占据统治地位，计算公式如下

$$F_{Schlick}(v, h) = F_0 + (1 - F_0)(1 - (v \cdot h))^5 \quad \text{公式 (2.6)}$$

2.2.4 几何遮挡项

几何遮挡项用来描述微表面被其他微表面所遮挡的情况，理想的几何遮挡项应该跟材质的粗糙程度 α 和微表面法向量分布函数 $D(h)$ 相关。在实时渲染中一般有如下几种选择：

1. Implicit

之所以叫 Implicit 是因为使用该计算方法，BRDF 只跟法向量分布函数 D 和菲涅尔 F 有关。计算公式如下：

$$G_{Implicit}(l, v, h) = (n \cdot l)(n \cdot v) \quad \text{公式 (2.7)}$$

2. Cook-Torrance^{[6][7]}

Cook-Torrance 是比较早期的一个遮挡函数，在过去电影渲染中使用了很多年，由于其昂贵的计算开销，在实时渲染中很少使用。

$$G_{Cook-Torrance}(l, v, h) = \min\left(1, \frac{2(n \cdot h)(n \cdot v)}{v \cdot h}, \frac{2(n \cdot h)(n \cdot l)}{v \cdot h}\right) \quad \text{公式 (2.8)}$$

3. Smith^[11]

接下来的几何遮挡项都是用来近似 Smith 的方法，应该跟对应的法向量分布函数 $D(h)$ 一起使用。Smith 把几何遮挡项分成了两个部分，分别对应 v 和 l ，所用的计算公式是一样的。Smith 函数会考虑材质的粗糙程度和对应的法向量分布函数，可以得到比 Cook-Torrance 更加精确的效果。

$$G_{Smith}(l, v, h) = G_1(l) \cdot G_1(v) \quad \text{公式 (2.9)}$$

Schlick^[10]针对 Beckmann 分布近似估计了 Smith 方程。

$$G_{1-Schlick}(v) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \quad k = \alpha \sqrt{\frac{2}{\pi}} \quad \text{公式 (2.10)}$$

同时 Walter^[9] 针对 Beckmann 分布也提出了一个计算更加便宜的近似公式，见公式 2.11。

$$c = \frac{n \cdot v}{a\sqrt{1 - (n \cdot v)^2}}$$

$$G_{1-Walter}(v) = \begin{cases} \frac{3.535c + 2.181c^2}{1 + 2.276c + 2.577c^2}, & c < 1.6 \\ 1, & c \geq 1.6 \end{cases} \quad \text{公式 (2.11)}$$

2.2.5 渲染方程

有了上面介绍了 Specular BRDF 后，我们可以给出在游戏中经常使用的基于物理的染模型。在不考虑次表面散射的情况下，渲染方程如公式 2.12 所示

$$L_o(v) = \int_{\Omega} f(l, v) \otimes L_i(l)(n \cdot l) d\omega_i \quad \text{公式 (2.12)}$$

其中 $f(l, v)$ 是 BRDF， $L_i(l)$ 是光源的贡献。

所谓能量守恒就是指一个表面无法反射比入射光更多的能量。在数学上可以用如下公式表述

$$\forall l, \int_{\Omega} f(l, v)(n \cdot l) d\omega_o \leq 1 \quad \text{公式 (2.13)}$$

本渲染引擎使用的 BRDF 中，D 为 Blinn-Phong 分布，G 为 Implicit 遮挡项，经过能量守恒归一化后，可以得到如下渲染方程。

$$L_o(v) = \left(c_{diff} + \frac{a_p + 2}{8} (n \cdot h)^{a_p} F(c_{spec}, l, h) \right) \otimes c_{light}(n \cdot l) \quad \text{公式 (2.14)}$$

其中 c_{diff} 是材质 Diffuse 参数， c_{spec} 是材质 Specular 参数， a_p 是高光指数。

2.3 本章小结

本章介绍了在实时渲染中使用到的相关技术，包括基本的渲染 API 编程。很抽象的介绍了渲染 API 的基本概念，这些抽象在设计跨渲染 API 的渲染引擎时非常重要，基本上可以把这些抽象概念分别设计成一个类。还介绍了在游戏界被广泛使用的基于物理的渲染模型，这些模型是后面所有渲染算法的基础。

第3章 渲染引擎架构实现

渲染引擎不同于游戏引擎，更加侧重于实时渲染这部分。而游戏引擎则是个更加庞大复杂的系统，游戏引擎除了包含渲染模块外，还要包括网络模块、AI模块、游戏逻辑脚本模块等。本论文实现的实时渲染引擎只负责图形渲染这一部分。本渲染引擎的整体架构如图 3.1 所示

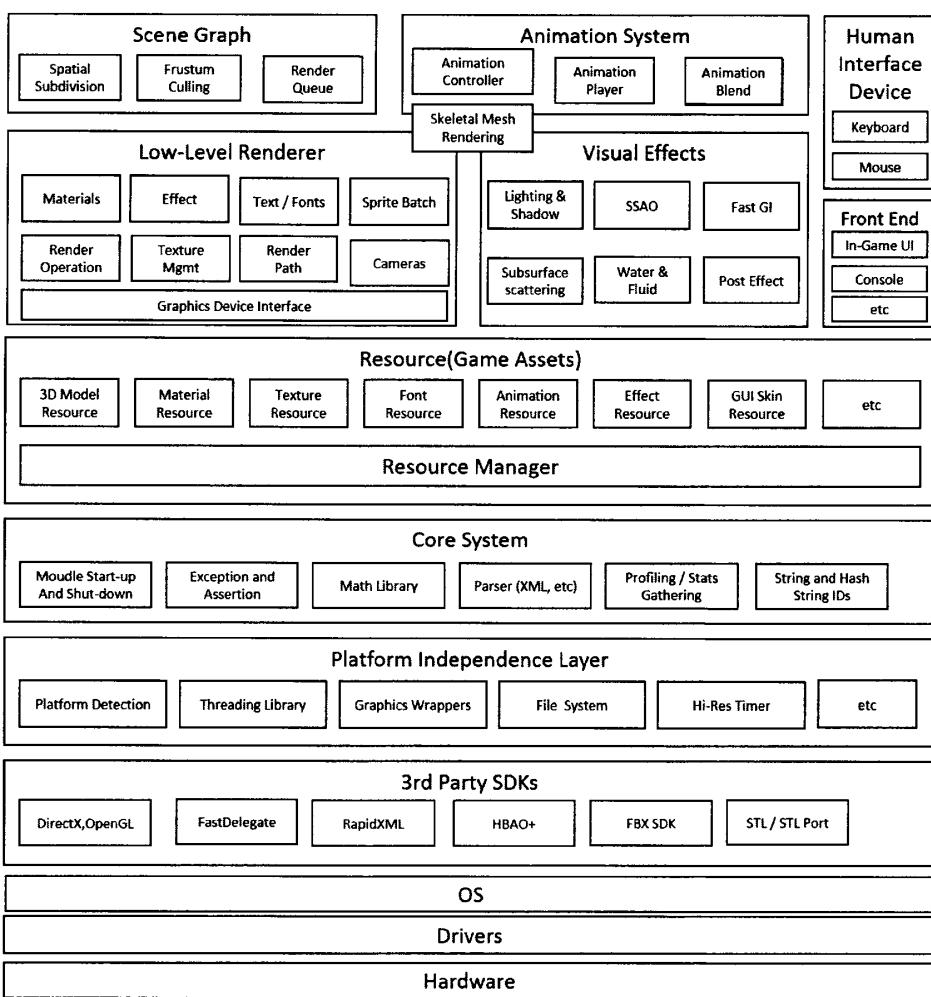


图 3.1 渲染引擎架构图

3.1 跨 DirectX 11 和 OpenGL 的渲染系统

实时渲染引擎的架构在设计之初首先需要考虑的是底层使用哪种渲染 API，目前桌面电脑上的游戏基本上采用微软的 DirectX 进行渲染，而移动平台上则使用 OpenGL ES 进行渲染，OpenGL ES 是从 OpenGL 的基础上发展而来，而且按照目前的趋势，移动平台上的图形硬件性能越来越强，最终 OpenGL ES 也会发展成跟现代 OpenGL 差不多的一个渲染 API。本渲染引擎在设计之初就考虑同时支持 DirectX 11 和 OpenGL 这两种渲染 API，未来还将支持 OpenGL ES 和 DirectX 12。开发者只需编写一次代码，就可以任意选择 DirectX 或者 OpenGL 中一种进行渲染。

设计架构一个同时支持两种渲染 API 的渲染引擎存在一定的技术挑战，首先这两种渲染 API 在功能上有一定差异。为了简化这种差异，本渲染引擎对 OpenGL 版本要求有一定的限制，要求最低 4.0，目前的图形显卡基本上都支持 OpenGL 4.0 版本。下一步就是抽象这两种渲染 API 的相同功能集。渲染 API 最重要的几个概念就是渲染设备、GPU 资源、渲染状态和帧缓存。采用面向对象的编程思想，可以在 DirectX 和 OpenGL 之上再抽象一层渲染 API，把这些相同功能集以抽象接口的方式声明在基类中，而具体用哪个渲染 API 实现封装在子类中，如图 3.2。

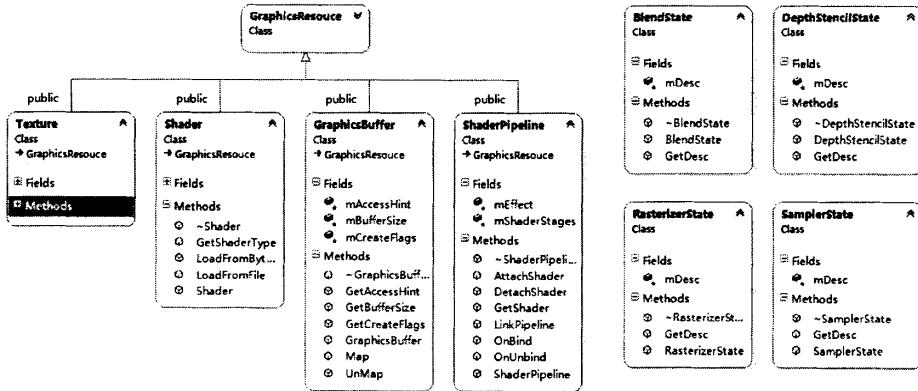


图 3.2 渲染 API 类图

渲染系统的实现如同引擎中的其他系统，被设计成一种插件的体系。这种插件机制最大的好处就是可以根据需求在初始化时候或者动态地载入插件，并且插件新功能增加、Bug 修复都不需要重新编译引擎代码，只需要重新编译生成该插件进行替换就可以了。插件具体是通过动态链接库的方式实现的，所有跟 DirectX、OpenGL 相关的代码都被封装在插件中，编译生成动态链接库。

在渲染系统的设计过程中，最难的部分莫过于对着色器资源（Shader）的抽象。这种障碍主要来自于 DirectX 跟 OpenGL 都有自己的 GPU 着色语言。DirectX 的着色语言是 HLSL，OpenGL 的着色语言是 GLSL，不能相互通用。虽然 Nvidia 显卡公司也推出了可以同时在 DirectX 和 OpenGL 上使用的 CG 着色语言，但 CG 着色语言只能支持 Shader Model 4.0，许多高级的显卡特性都无法使用，如 Compute 着色器。本渲染引擎在这方面最后设计是 HLSL 和 GLSL 都使用，对于同一个特效，分别编写一份 HLSL 和 GLSL 着色器代码。这两个版本的着色器具有相同的文件名和入口函数，在创建着色器时根据使用的渲染 API 去相应的着色器代码库里载入着色器代码。随着引擎的不断开发迭代，发现这并不是一个具有可扩展性的方案，主要有两个缺点：

- 1) 复杂的实时渲染需要大量的着色器代码，同时编写两套代码工作量大。
- 2) 两个版本的着色器代码不容易维护，修改其中一个版本的代码后很容易忘记同步另一个版本代码。

终极的解决方案应该是像 Unreal、Unity 引擎一样，通过交叉编译的方法，把 HLSL 的着色器代码从字节码层面上转换成对应的 GLSL 代码，目前已有开源的交叉编译工具，如 Unreal4 使用的 HLSLCrossCompiler，Unity 使用的 hsl2glsl。通过这个方法，引擎只需要维护一份着色器代码，大大减少了复杂度。

3.2 材质和特效系统

本渲染引擎在设计之初就定位是可编程的渲染管线，完全抛弃古老的固定渲染管线，如 OpenGL 的固定管线。这是因为现代图形显卡已经完全支持可编程管

线，况且新一代的渲染 API 如 DirectX 11 已经不再支持固定管线。在可编程渲染管线中，渲染三维场景中的模型至少需要指定顶点着色器和像素着色器。特效系统就是用来创建和管理这些着色器代码。特效只是用来渲染三维模型的 GPU 着色器组合。要渲染一个模型除了特效，还需要结合模型的材质，即给特效提供输入，如模型的纹理贴图等，如图 3.3。所以在本渲染引擎里，材质的定义是，渲染一个三维模型所需要的特效以及在 GPU 上执行这个特效所需的的各种输入。

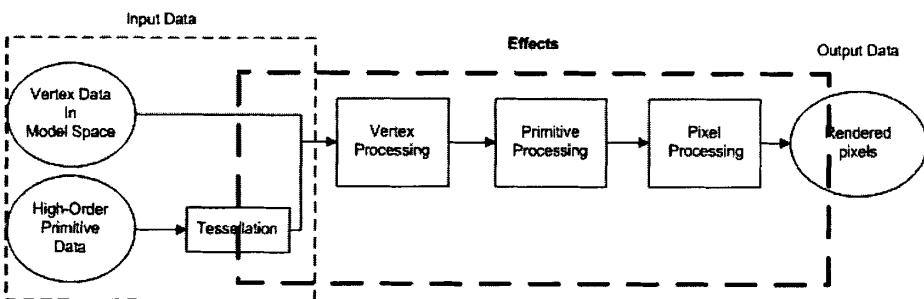


图 3.3 特效和渲染管线图³

3.2.1 特效系统

上面已经介绍，特效其实就是 GPU 着色器的组合，但还包括绘制物体所需的各种渲染状态，如 RasterizerState、DepthStencilState、BlendState 等。每次 GPU 绘制命令都需要提供这样一个着色器组合，最少包括顶点着色器和像素着色器，最多包括顶点着色器、几何着色器、细分(Tessellation)着色器和像素着色器，并且渲染一个三维模型，可能需要好几个着色器组合。如在带动态阴影的实时渲染中，首先需要一个渲染生成阴影贴图的着色器组合，此外还需要一个着色器组合渲染物体到屏幕上。特效不仅需要管理各种着色器代码的组合，还要提供接口让材质系统设置特效参数。本渲染引擎特效系统的设计参考了微软公司 D3DX 的 Effect 框架。Effect 框架使得 GPU 着色器、渲染状态的管理更加方便，不再需要手动设

³ [http://msdn.microsoft.com/en-us/library/windows/desktop/bb173329\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb173329(v=vs.85).aspx)

置着色器的 Uniform 变量、绑定纹理采样器、设置渲染状态等。这不仅减少了出错的可能，还提高了程序的效率，可以把相同特性的物体组合起来一起渲染。本渲染引擎的特效结构如下图所示。

```
Effect OpacityEffect
{
    [ ParamAutoBinding ... ]

    [ SetSamplerGroup ... ]

    Technique TechniqueName
    {
        Pass PassName
        {
            ...
            VertexShader file entry
            PixelShader file entry

            [ SetRenderStateGroup ... ]
            [ SetRenderStateGroup ... ]
        }

        [Pass ... ]
    }

    Technique TechniqueName
    {
        Pass PassName
        {
            ...
            VertexShader file entry
            [ GeometryShader file entry ]
            PixelShader file entry
        }
    }
}
```

图 3.4 特效脚本结构

从图 3.4 的组织结构中可以看出，渲染特效（Effect）可以有一个或多个渲染技术（Technique）组成。一个 Technique 就是以某种渲染技术完成一次渲染操作。Technique 是一个非常有用的概念，可以对硬件能力的不同分别实现 Technique，在硬件配置好的平台上选择效果好同时开销也高的 Technique，在硬件配置相对差的平台选择效果一般同时开销低的 Technique。这在游戏中很常见，就是调节游戏

的画质。一个渲染技术（Technique）包括一个或多个渲染批次（Pass），有些渲染算法可能需要多个渲染批次才能完成。每个 Pass 就是一次 GPU 绘制命令，包括设置各种着色器，渲染状态等。在执行绘制命令时，Pass 首先给 GPU 可编程管线绑定各个着色器，然后更新特效的参数，即每个着色器的 Uniform 变量、Constant Buffer 等，最后还要设置各种渲染状态。有了特效框架后，所有这些繁琐的操作都由引擎帮你完成，这极大地简化了 Shader 编程的复杂度。

上面介绍了特效的作用，而特效系统则是创建和管理这些特效的管理器。特效系统要根据当前的渲染 API 选择对应的特效代码，如果渲染 API 是 DirectX 11，那么特效要从对应的 HLSL 代码库里创建。如果渲染 API 是 OpenGL，则从 GLSL 代码库里里创建。特效还支持额外的宏定义，宏定义可以简化着色器代码的编写。编写一个带有各种宏开关的通用着色器代码，根据不同类型的模型材质输入设置相应的宏开关。如渲染三维网格模型的特效，带有法线贴图、透明贴图、高光贴图等开关。如果材质支持这些贴图，只需要在创建这些特效的时候，提供额外的宏定义即可。相当于只编写一个着色器代码，编译的时候根据宏开关，可以生成不同的效果的着色器版本。

此外，渲染一个特效有时还需要使用引擎的一些参数，如系统时间、摄像机的投影矩阵等，这些参数可以用自动绑定的方式，在特效开始执行时，引擎会自动设置。

3.2.2 材质系统

材质系统是整个引擎中最灵活和富有价值的系统之一。上面已经提到材质是渲染三维模型所需要的特效以及在 GPU 上执行这个特效所需的输入。这些输入包括模型的纹理贴图、材质参数等。材质以脚本的方式组合了渲染一个模型所需的所有素材，如图 3.5 所示。通过材质脚本，你可以不修改任何一行代码就能改变物体的渲染。通过材质脚本，可以绑定特效所需的输入和素材，图 3.5 中，Param 命令就是用来绑定特效参数和素材的。其中 type 是参数的类型，可以是纹理或数值类型。name 对应着色器代码中的变量名，value 是 type 对应类型的值，可以

是材质纹理的图片路径或数值等。在读取材质脚本时，会自动去载入 value 对应的素材。

```
Material name
{
    Param type name [semantic] [sampler] value

    Effect name
    {
        [ Macro name value ]
    }

    RenderQueue [ opaque | transparent | background | overlay ]
}
```

图 3.5 材质脚本结构图

此外，引擎还对三维模型的材质进行了分类，这样可以把相同材质类型的模型归类到一起渲染，提高渲染效率。并且在某些情况下，最终渲染效果的正确与否还跟模型渲染的先后顺序有关。如在有半透明物体的场景中，需要先渲染所有不透明物体，之后再用透明混合的方式渲染半透明物体。在本渲染引擎的实现中，开始渲染所有物体前，会根据物体的材质类型，把可渲染的物体归类到对应的渲染队列中。渲染队列中的每个元素都是一次可执行的渲染操作。我们可以对渲染队列中的这些渲染操作以某种规则进行排序，这样使得执行这些操作时渲染状态的改变最少，从而提高渲染效率切换渲染状态如设置 Shader、DepthStencilState 等是相对比较耗时的过程。半透明物体的渲染，需要按照深度从后往前渲染，所有在排序半透明渲染队列里的操作时，就需要根据深度进行排序，才能保证渲染的正确性。

3.3 场景管理

场景管理在实时渲染引擎中扮演中非常重要的角色，实际应用中不同的场景需要使用不同的空间数据结构和算法，如室外场景一般使用基于四叉树、八叉树的场景空间分割策略。虽然本渲染引擎目前还没有支持这些高级的场景器，但在设计之初已经考虑到将来可能会有这方面的需求，所以场景管理系统也被设计为

插件的方式，方便日后扩展。

目前引擎的场景管理是基于场景图的方式，场景图一般是一种类似多叉树的数据结构。本渲染引擎实现的场景图参考Ogre图形渲染系统，如图3.6所示。场景图只负责维护场景结构，场景节点不关心挂接在其上的任何东西，也不关心其管理方法。引擎抽象出了两种数据结构场景节点(Scene Node)和场景对象(Scene Object)。分离这两个概念非常有好处，这使得场景图可以被方便的扩展、修改和重写。实现其他场景空间分割策略，如要实现八叉树，只需要继承实现场景图的接口，实现相应的分割管理算法，而无需关心具体挂接在场景图上面的内容。同时任何想被放置在三维空间中的物体，都可以通过子类化场景对象，挂接在场景节点中。场景对象可以包括可渲染的对象如三维网格和不可渲染的对象如光源、声源等。

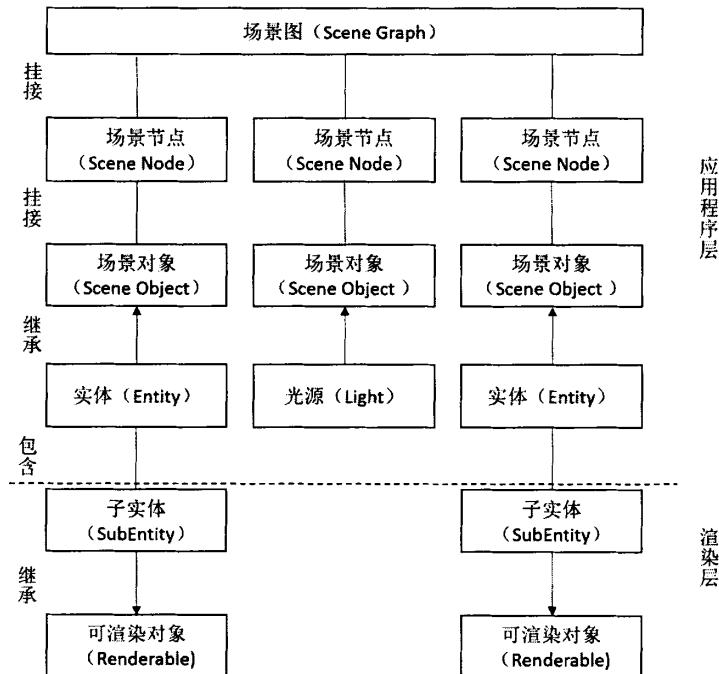


图3.6 场景图结构

场景管理的另一个重要的功能是，执行摄像机视锥剔除，填充渲染队列。在引擎开始渲染前，遍历场景图，收集挂接在场景节点上的场景对象，这种场景对象有些是可渲染的，需要按照材质分别填充到对应的渲染队列里。另外一些不可渲染的对象，可能需要填充到其他队列，如对于光源，引擎也有一个光源队列。有了这种层次式的场景图后，进行摄像机视锥剔除变得非常容易。每个场景节点都有一个接口，获取该节点在三维场景中所覆盖的空间区域即包围盒。如果该包围盒不在当前摄像机的视锥里，在更新渲染队列的时候，就没有必要继续遍历该场景节点下的所有子节点，从而提高渲染的效率。

3.4 动画系统

动画系统是实时渲染引擎不可缺少的一部分，如果没有动画，三维场景中的物体将都是静止不动的。游戏里面的动画大致上可以分为刚体层阶式动画和蒙皮动画。游戏中大部分动画如人物角色、怪物的动画，是蒙皮动画。本渲染引擎也实现了蒙皮动画，可以很方便地在程序里控制、播放美术人员制作的动画。

3.4.1 蒙皮动画

蒙皮动画，又称蒙皮骨骼动画，这种动画有两部分组成，骨骼和蒙皮网格。骨骼是由关节（joint）以层阶的方式连接而成，类似多叉树的父子结构。在动画里并没有真实存在的关节，关节其实只是一个没有实体的空间坐标变换。而蒙皮网格其实跟普通的网格没有上面区别，如果没有骨骼驱动它变形运动，就跟静态网格完全一样。蒙皮动画的关键在于蒙皮的过程，蒙皮把网格的顶点绑定到骨骼上，并赋予不同的控制权重，这样每个顶点的运动会受到多个关节的控制。

关节是坐标空间，骨骼层次其实就是嵌套的坐标空间，父关节的变换将带动其下所有子关节的变换。所谓的骨骼动画，就是通过表示关节的一个空间变换矩阵来影响所绑定顶点的位置实现的。美术人员录制骨骼动画的过程其实是在设置骨骼的空间位置。不管是哪种动画技术，都是随着时间而推移的，骨骼动画也有关键帧序列。骨骼动画的关键帧就是骨骼在时间轴某位置上的一个姿势（Pose），

如图 3.7b。通常情况下每秒需要 30 或 60 个姿势才能播放出比较平滑的动画。

每个骨骼动画都由一个绑定姿势，即把三维网格绑定到骨骼之前的姿势，如图 3.7a 所示的姿势，也叫 T 姿势(T-pose)。这个姿势定义了一个坐标空间变换，可以把顶点从每个关节的局部空间变换到世界空间。我们需要的是这个变换的逆变换，可以把蒙皮网格上的顶点变换到关节空间。骨骼动画的播放过程大致如下：

- 1) 确定当前动画时刻前后两个关键帧，按时间插值计算出当前时刻骨骼的位置，线性插值位置和四元素插值朝向。
- 2) 从根关节开始遍历整个骨骼，计算每个关节的世界空间变换矩阵。这是个递归的过程。
- 3) 计算骨骼的矩阵调色板 (matrix palette)，这个变换矩阵就是把蒙皮网格中的顶点从绑定姿势变换到当前姿势的空间变换矩阵。在每个关节单独作用下，顶点变换后的位置可通过如下公式计算。

顶点的新位置 = 蒙皮网格中的顶点位置 * 绑定姿势的逆矩阵 * 当前骨骼的世界变换矩阵

蒙皮网格中的顶点可以同时受到多个关节的控制，所以最终的位置需要按每个关节的权重线性叠加，所有的权重和应该等于 1。

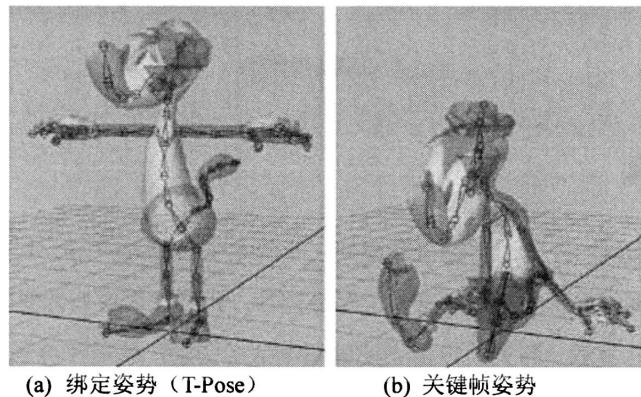


图 3.7 骨骼动画姿势⁴

⁴ 数据来源 Game Engine Architecture^[1]

3.4.2 动画混合

动画混合是一种混合两个或多个姿势来产生最终角色姿势的技术。这在动画系统中非常重要，可以大大减少美术人员需要制作的动画数量。比如人物角色的动画一般分为上半身和下半身动画，可以分开来单独制作。一般战士角色的下半身动画有行走、奔跑、跳跃等，上半身动画有拔剑、技能释放、击打等。通过混合上下半身的动画姿势，可以很方便的创造出边行走边拔剑、边奔跑边击打等动画组合。本渲染引擎也支持动画混合，并提供接口让程序设置每个动画姿势的混合权重，如图 3.8 所示。

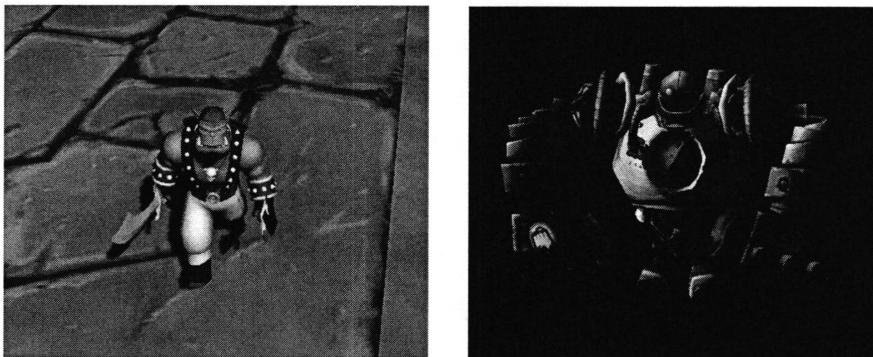


图 3.8 引擎动画效果

3.5 GUI 系统

如果只是实现一个实时渲染引擎框架，可以不支持 GUI 系统。但图形界面对于场景信息的现实很有帮助，而且有了图形控件，我们可以更加方便的调试渲染算法。图形控件本质上跟其他可渲染的物体没什么区别，就是在屏幕上绘制 2D 图形。本渲染引擎用面向对象的方法实现了一个简单的 GUI 系统，实现了一些基本的图形控件，如图 3.9 所示。

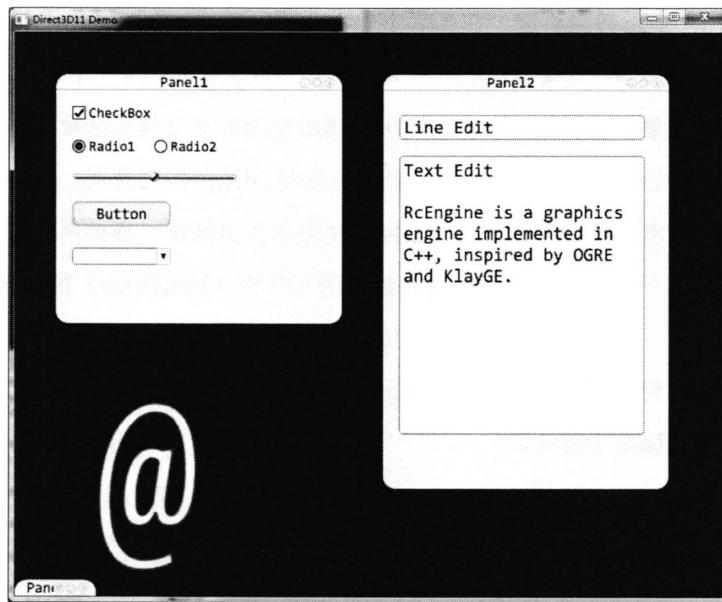


图 3.9 GUI 控件效果

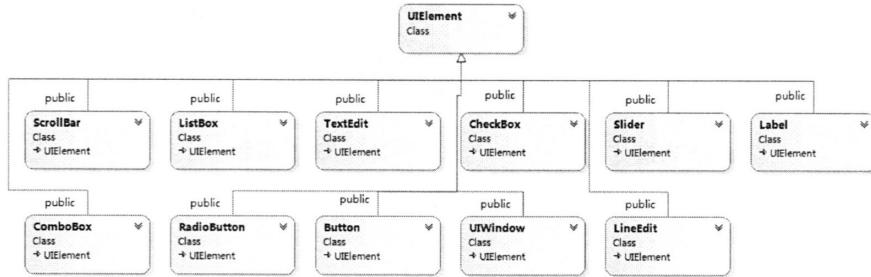


图 3.10 GUI 系统类图

所有的图形控件都继承自基类 `UIElement`，如图 3.10。基类 `UIElement` 定义了一些必须实现的接口，如负责绘制自己的 `Draw` 接口，每一次渲染循环用于更新控件自身状态的 `Update` 接口等，以及一些基本的消息响应函数，包括鼠标、键盘的响应函数。除了这些基本的消息响应函数外，每个子控件都有自己特定的事件响应。如按钮控件的点击事件、滚动条控件的滚动事件等。本渲染引擎采用了

类似于 C#中的事件-委托机制。在具体实现中，使用到了一个非常好用的 C++开源库 fastdelegate。委托其实是像函数一样可调用的对象，可以把具体的消息响应函数委托给该对象，当事件发生时就能自动调用对应的消息响应函数。

此外，我们还需要一种灵活的方式来控制改变 GUI 控件的外观，即控件的皮肤。本渲染引擎也支持自定义皮肤，这样只要美术人员创作一套控件皮肤，就能让程序界面焕然一新。每个控件的皮肤都是用九宫格（NinePatch）的方式定义，如图 3.11a 所示。九宫格可以适应不同尺寸的控件而不产生因皮肤图片放大产生的失真效果。最终不同控件的皮肤打包到一张图片中，如图 3.11b，这样不仅能够节省资源，还能提高绘制效率。

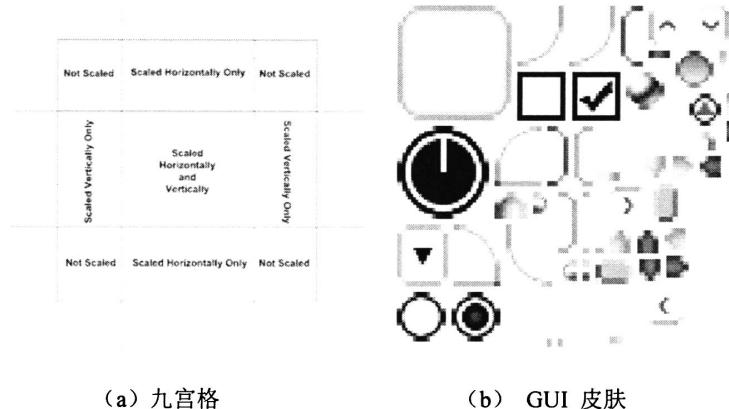


图 3.11 GUI 皮肤和九宫格图

3.6 素材管线

评价一款实时渲染引擎的优劣不仅要看其渲染的效果，还要看引擎的易用性。其中一个功能强大的素材管线对引擎的易用性非常重要。试想一下，如果美术人员在 3dmax 建模软件里创建完模型后，只需要点击一个按钮，就能在引擎里实时地渲染，是不是很方便。这就需要一个强大的素材管线，实时渲染引擎使用到的素材主要有纹理贴图、三维网格模型，动画数据、字体、GUI 皮肤、材质、特效等。

3.6.1 三维网格模型和动画

本渲染引擎目前支持两种模型格式,FBX格式和Ogre渲染引擎定义的格式。对于这两种格式,引擎分别实现了导入器。大部分情况下,FBX模型格式可以满足基本需求,因为一般三维模型都是在Autodesk公司的建模软件里创作,如3dmax、maya。FBX是Autodesk制定的用于旗下建模软件间交换的通用格式,3dmax、maya甚至开源的Blender建模软件都支持导出模型、场景到FBX格式。FBX导入器可以直接把模型转换成引擎自定义的模型格式文件。使用自定义的模型格式是为了加快模型的载入,如模型的顶点数据和索引数据都是以预先组合好的形式按数据块存储的。在创建模型时,可以直接读入内存传递给渲染设备创建顶点和索引缓存。此外,如果模型是带骨骼动画的,在导出网格数据的时候,会计算模型的骨骼和蒙皮,随网格数据一起导出到同一个文件。

动画数据导出后是跟模型数据分开来存储的,这是因为一个蒙皮网格模型可能会附带许多个动画,每个动画称为胶片(AnimationClip)。动画可以在需要播放的时候按需载入,没有必要和模型一起一次性都读入到内存。如图3.8所示的Ogre动画模型,导出后的数据如图3.12所示。

3.6.2 纹理和材质

目前引擎只支持一种图片格式,DDS图片格式。DDS图片基本上可以支持渲染引擎里所有的纹理格式,包括压缩格式DXT1、DXT3和DXT5。此外,DDS还支持Mipmap、立方体贴图和纹理数组,即在一个图片文件里就能存储所有纹理,非常方便。

在导出模型到自定义的格式的过程中,还会生成相应的材质文件和转换材质图片到DDS格式。上面的小节已经介绍过材质和特效脚本,模型导出器在转换模型时也会处理模型的材质,根据模型的材质设置相应的特效宏开关。如果模型带有法线贴图,则打开特效文件的_NormalMap开关。如果是骨骼动画模型,则打开_Skinning开关。模型导出器处理完模型后,一般会生成如图3.12所示的素

材。

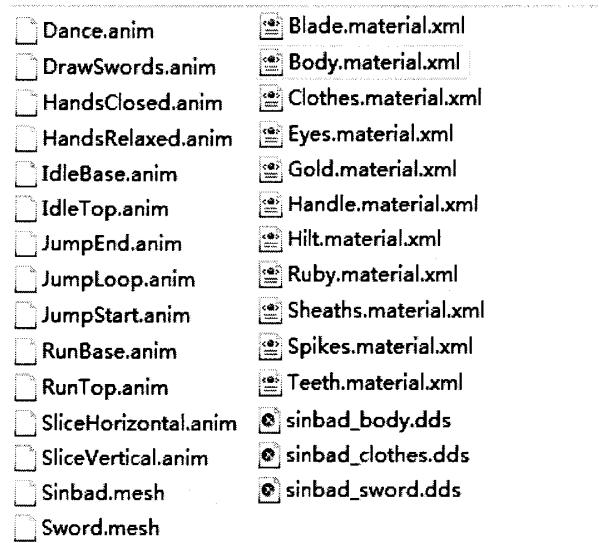


图 3.12 素材导出图

3.7 本章小结

本章详细介绍了整个渲染引擎的架构和实现，包括如何设计一个跨 OpenGL 和 DirectX 的渲染器，灵活的特效和材质系统。在这个框架之上，还介绍了以场景图的方式组织管理三维空间中的对象。结合素材管线，引擎可以直接播放美术人员在建模软件中录制的动画。有了这样一个实时渲染的框架，接下来实现各种高级渲染算法就会变得容易很多。

第4章 皮肤和流体

4.1 人脸皮肤渲染

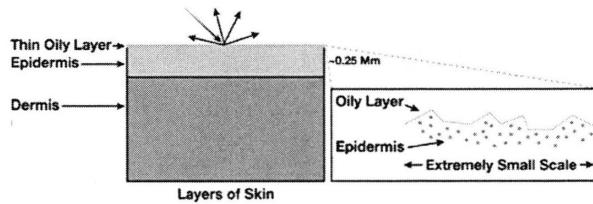
现实生活中有许多物体材质呈现出一种半透明的效果，如皮肤、玉、牛奶、玉等。这种效果是由于半透明材质的物体受到多个光源的透射，光线透过物体表面，在内部发生多次散射，最终在与射入点不同的地方射出表面，这种现象称作次表面散射。具有半透明效果的次表面散射材质是实时渲染中最复杂的材质之一。

皮肤就是其中一种呈现次表面散射效果的重要材质，而且人对于皮肤的外观及其敏感，尤其是脸部。许多电子游戏都是角色驱动的，游戏里存在大量的人物角色，且游戏中许多剧情对话，都需要近距离渲染人物的面部表情。如果能够渲染出逼真的人脸模型，可以让玩家在体验游戏的过程中更加沉浸。

现代三维扫描技术已经可以捕捉到人脸的各种特征，如皱纹、胡渣、疤痕等，这些特征都是渲染逼真的人脸模型所必须的。但是想要实时渲染出逼真的人脸模型还是存在一定的困难，这种障碍主要来自于无法在物理上正确模拟次表面散射过程。本章实现了一种在纹理空间快速模拟次表面散射效果的实时渲染方法，能够渲染出真实的人脸模型。

4.1.1 人脸皮肤的光学特性

人脸皮肤看上去总是显得比较柔和、红润，主要是因为皮肤是一个多层的组织结构，每一层组织的光学特性都不一样，光在皮肤各层组织里发生了不同程度的吸收和次表面散射现象。真实的皮肤组织结构相当复杂，如果完全按照皮肤的医学构造来建模，整个计算过程将非常耗时，无法实时地进行渲染。Donner and Jensen 2005^[13]提出了一种三层的人脸皮肤材质模型，如图 4.1。并且根据这种三层材质模型，可以非常快速地近似估计次表面散射效果，渲染出逼真的人脸皮肤效果。

图 4.1 三层皮肤模型⁵

皮肤的最上面一层是油脂层，当光线照射到皮肤表面时，其中一部分被油脂层直接反射掉，剩余的光才会进入皮肤表层，被内层皮肤组织吸收或者继续散射。由于光和皮肤油脂层之间的 Fresnel 效应，这部分被反射的光颜色也不受皮肤颜色的影响。剩下的光线会透过油脂层进入表皮层，并在表皮层中发生次表面散射与吸收现象。表皮层的主要组成成分是黑色素，黑色素对光的波长并不敏感，即对所有不同波长的光都会产生相同的作用。表皮层会吸收一部分光线，但不会使光的颜色发生巨大改变。没有被吸收的光线会进入真皮层，与各个皮肤组织发生作用。光线会在表皮层和真皮层中无规则地向各个方向发生散射，最后在皮肤表面与入射点不同的位置射出。经过无数次无规则散射后，最终的效果就好像绝大部分光线均匀地向所有方向散射。这种光在皮肤内部的多重散射效应简化了最终的计算模型，使得我们可以使用漫反射扩散剖面（diffusion profile）高斯模糊的方法来近似估计次表面散射。从皮肤的三成材质模型分析可知，真实感皮肤渲染主要包括两个部分：表面高光反射（Specular Surface Reflectance）和次表面散射（Diffuse Subsurface Scattering）。

4.1.2 基于纹理空间的次表面散射^[9]

漫反射扩散剖面（Diffusion Profile）是次表面散射理论中的一个非常重要的概念，对光在半透明材质表面之下的传播和散射提供了一个近似的描述方法。当光线与半透明物体表面发生交互时，会向四周各个方向扩散，并且随着扩散距离

⁵ http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html

的增加，发生散射的光线强度会越来越弱。均匀的半透明材质，光线的扩散跟入射角度无关，会均匀地向四周所有方向扩散。此外，不同颜色的光线扩散能力也不相同，红色光的扩散能力远远大于蓝色光和绿色光，这是由于皮肤真皮层中的组织对不同频率的光吸收能力不同。可以用三条曲线来描述 RGB 颜色光线的扩散能力，如图 4.2 所示。

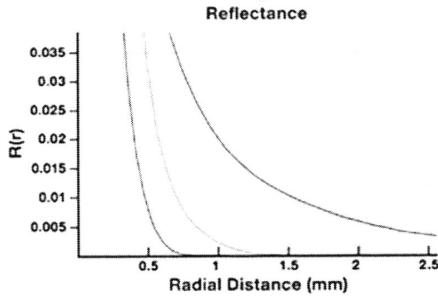


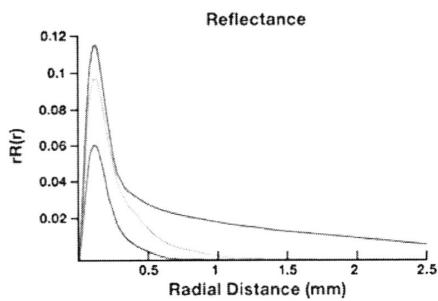
图 4.2 光线扩散能力曲线

4.1.2.1 高斯函数拟合漫反射扩散剖面

Donner 等人^[14]发现三层皮肤模型的漫反射扩散曲线可以通过多个高斯函数的线性叠加来近似模拟，通过同时大量实验论证，使用 6 个带权重的高斯函数内核已经可以逼近皮肤的扩散曲线。图 4.3a 本文使用的 6 个高斯函数内核以及混合权重，图 4.3b 是使用高斯函数拟合的三层皮肤扩散曲线。最终结果表明，使用 6 个高斯项进行拟合，已经达到令人满意的效果。

	Variance (mm ²)	Red	Blur Weights Green	Blue
•	0.0064	0.233	0.455	0.649
•	0.0484	0.100	0.336	0.344
•	0.187	0.118	0.198	0
•	0.567	0.113	0.007	0.007
•	1.99	0.358	0.004	0
•	7.41	0.076	0	0

(a) 高斯内核及混合权重



(b) 高斯拟合的扩散曲线

图 4.3 高斯拟合曲线

4.1.2.2 纹理空间漫反射的实现

由于需要对皮肤表面的光照辐射度进行多次高斯模糊来拟合漫反射扩散剖面，本文采用纹理空间的算法，把模型光照计算得到的辐射度信息，按模型的纹理坐标展开储存在一张二维辐射度纹理中，如图 4.4 所示。再以图 4.3a 中所示的高斯内核进行分别进行高斯模糊，得到 6 张模糊后的辐射度纹理。在最终渲染的时候，按图 4.3a 中所示的混合权重线性叠加这 6 张辐射度纹理就可以得到次表面散射的结果。整个算法流程如图 4.4 所示。

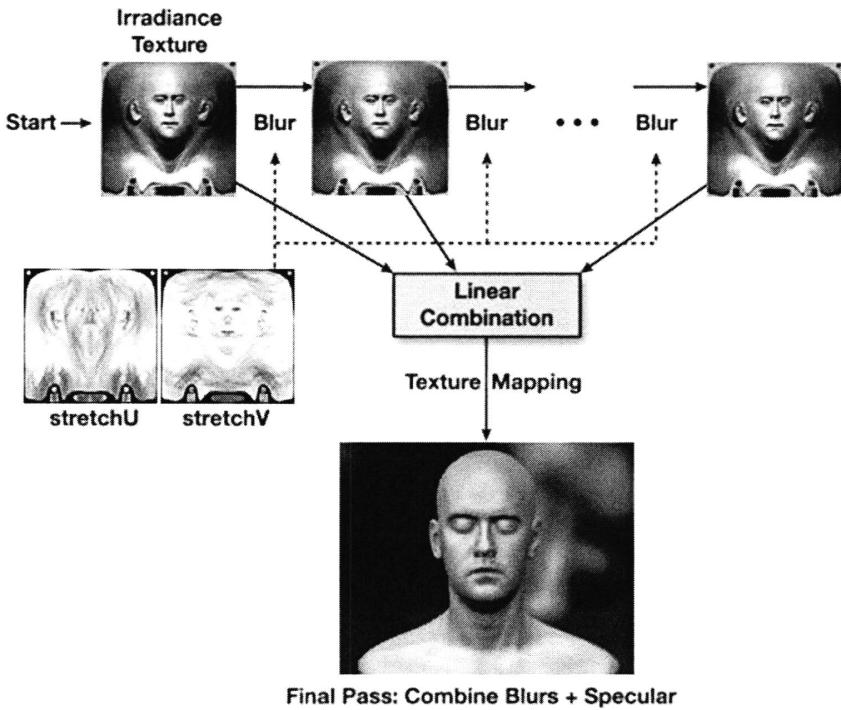


图 4.4 纹理空间次表面散射算法流程图

4.1.3 基于 Kelemen/Szirmay-Kalos BRDF 的反射模型

经典的局部光照模型，如 Phong 模型和 Blinn-Phong 模型，在过去几十年里被广泛应用在实时渲染上。但在人脸皮肤上使用经典光照模型并不能产生逼真的

效果，主要原因是经典光照模型并不是基于物理的，没有考虑能量守恒，往往出射光的能量远远大于入射光。也没有考虑菲涅尔效应，当以接近掠射角（grazing angles）观察时，反射效果会更加明显。为了渲染更加逼真的人脸皮肤，需要一个更具真实感、基于物理的皮肤材质光照模型。

本论文实现的是一个基于 Kelemen and Szirmay. Kalos 2001^[15]的 BRDF，来计算人脸皮肤油脂层的高光反射效果。该模型也是一个基于微表面理论的 BRDF，通过对几何衰减项进行化简，得到了一个更加适合在 GPU 上计算的 DRDF，可以用如下公式描述。

$$f_{r,spec}(\hat{L}, \hat{V}) = P_{\hat{H}}(\hat{H}) \cdot \frac{F(\lambda, \hat{H} \cdot \hat{L})}{\hat{h} \cdot \hat{h}} \quad \text{公式 (4.1)}$$

其中 $P_{\hat{H}}(\hat{H})$ 是法向量分布项，使用第二章介绍的 Beckmann 分布函数，并且通过预计算查找表的方式来加速该模型在 GPU 上的计算。

$F(\lambda, \hat{H} \cdot \hat{L})$ 是 Fresnel 项，本论文使用 Schlick 提出的方法快速近似计算方法。

\hat{h} 是没有单位归一化的半角向量， \hat{H} 是归一化的半角向量。

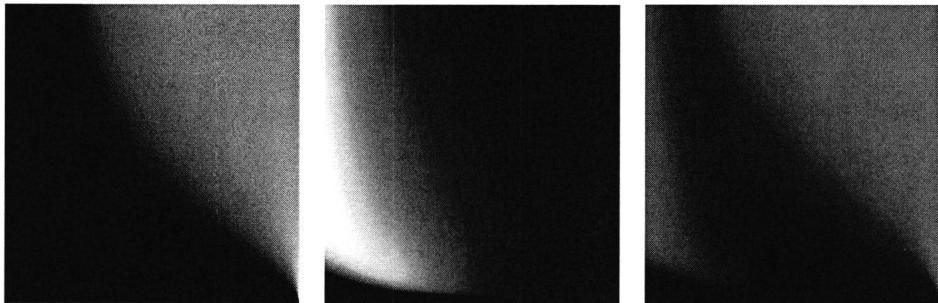
4.1.3.1 预计算 Beckmann 分布和能量守恒

Beckmann 分布模型是一种基于物理的模型，可以精确地表示具有不同粗糙程度的材质。但 Beckmann 法线分布函数的计算相当耗时，不适合在 GPU 上实时计算。通过观察发现，Beckmann 分布函数是一个只跟法向量和半角向量的点乘、材质粗糙参数 Roughness 相关的函数，可以预先将 Beckmann 分布作为一张纹理查找表进行预计算，图 4.5a 是使用公式对 Beckmann 进行预计算的结果。

基于物理的渲染必须要考虑能量守恒，在人脸皮肤光照模型里，进行次表面散射的总能量应该等于没有被皮肤油脂层直接反射掉的能量。被反射掉的能量比例可以在半球空间上根据 BRDF 对所有反射方向进行积分计算，所以剩下可用于次表面散射的能量比 ρ_{dt} 可通过如下公式计算。

$$\rho_{dt}(x, L) = 1 - \int_{2\pi} f_r(x, \omega_o, L) (\omega_o \cdot N) d\omega_o$$

ρ_{dt} 的值跟物体表面的粗糙度 α 和 $N \cdot L$ 有关，也可以采用预计算的方式存到一张二维纹理查找表，如图 4.5b 所示。为了节省资源，可以把预计算 Beckmann 和 ρ_{dt} 结合起来存到一张二维纹理的 RG 通道，见图 4.5c。



(a) Beckmann 预计算表 (b) ρ_{dt} 预计算表 (c) Beckmann 和 ρ_{dt} 纹理

图 4.5 预计算纹理查找表

4.1.3.2 最终的高光反射实现

本程序使用 3 个点光源作为渲染光源，并且分别为每个光源计算阴影，阴影算法我们选择的是 VSM (Variance Shadow Maps)^{[28][29]}。最终在像素着色器上使用公式 4.1 计算 Kelemen and Szirmay BRDF。其中 Beckmann 项已经预计算存在一张纹理贴图里，在渲染时只需要根据 $N \cdot L$ 与粗糙度 α 作为纹理坐标，进行一次纹理访问操作即可代替昂贵的计算开销。结合纹理空间的次表面散射后，最终的效果如图 4.6 所示。

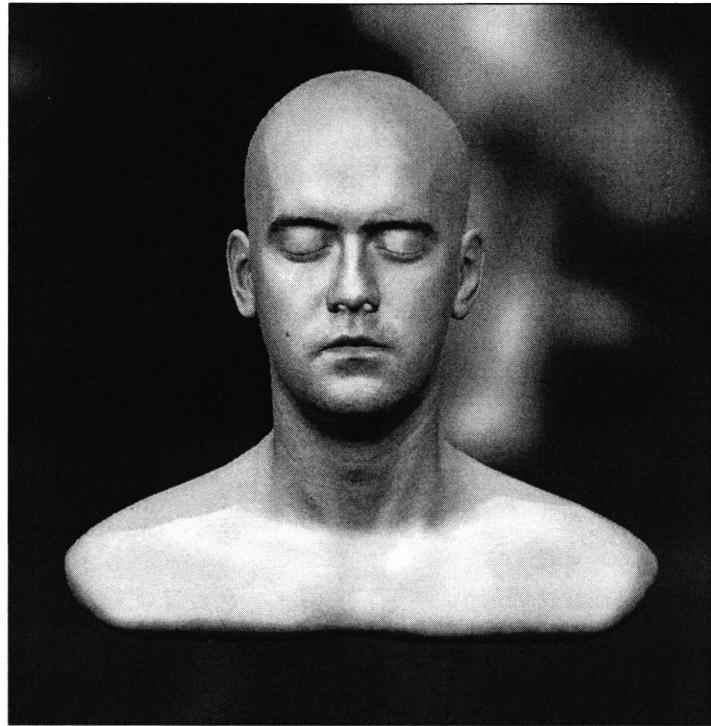
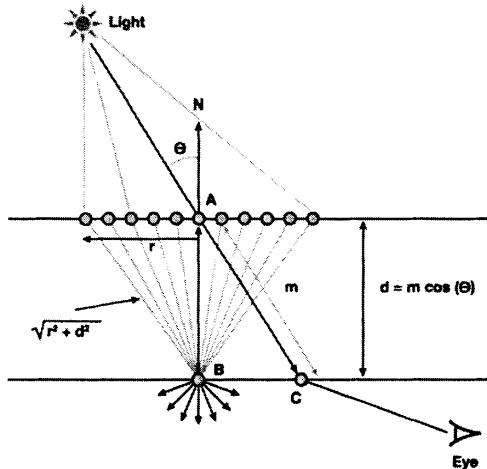


图 4.6 人脸皮肤纹理空间次表面散射效果图

4.1.4 透光效果

透光效果主要发生在人脸上比较薄的区域，如耳朵。常常可以观察到人的耳朵会有透光的现象，这种效果只使用纹理空间的次表面散射并不能捕获，因为在三维空间中距离相近的两个点在纹理空间里可能相差很远，如耳朵的前后面。可以通过使用改进的半透明阴影贴图（Translucent Shadow Maps, TSM）^[16]实现这种效果。在渲染阴影贴图时，除了保存距离光源最近的像素深度值外，还要额外保存这些被光照射到的像素纹理坐标，这样就能计算出光线在物体内部传播的距离 m ，如图 4.7 所示。

图 4.7 透射原理图⁶

当渲染耳朵这种比较薄的部位时，即使耳朵的一面没有被光直接照射到，也有透光的效果。在计算这类背光面如图 4.7 中的 C 点光照时，应该考虑从其对应向光面上 A 点周围透射过来的光线。在物理上计算 C 点的透射光比较复杂，需要根据向光面上每个点到 C 点的距离 m 和透射扩散曲线 $R(m)$ 计算。但计算 B 点的透射光比较容易，根据公式 4.2 计算，通过数学关系，发现 B 点的透射光可以通过前面得到的向光面次表面散射结果计算得到，这些散射结果已经保存在多张辐射度纹理上。只需要根据图 4.3 中的混合权重和 TSM 计算得到的厚度线性叠加模糊后的辐射度结果，即可计算 C 点的透射光照。

$$R(\sqrt{r^2 + d^2}) = \sum_{i=1}^k w_i G(v_i, \sqrt{r^2 + d^2}) = \sum_{i=1}^k w_i e^{-d/v_i} G(v_i, r) \text{ 公式 (4.2)}$$

加上透光效果后，明显可以观察到耳朵等脸部相对比较薄的区域会有通透的效果，如图 4.8 所示，图 4.8a 是没有开启透光的耳朵效果，图 4.8b 是开启后的效果。

⁶ http://http://developer.nvidia.com/GPUGems3/gpugems3_ch14.html

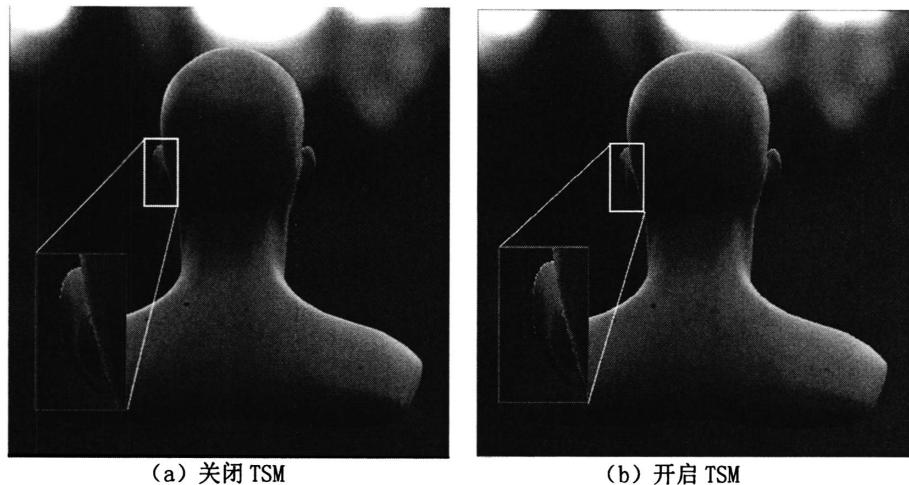


图 4.8 透光效果对比图

4.2 水面及流体渲染

真实感水面实时渲染也是游戏中经常用到的一种渲染场景，如湖面、海洋等。水面的渲染首先需要对水面进行建模，在海洋的模拟上，J.Tessendorf^[17]提出的基于频谱的分析方法，通过大量正弦波的叠加来模拟海面，并通过运行在 GPU 上的快速傅里叶变换来合成海浪谱分布的高度场，能实时的渲染出非常逼真的海洋。本论文的水面模型来自物理模拟，主要是为了模拟船在水面上行驶后的尾迹效果。本论文提出一个整合的水面渲染框架，可以渲染水面，也用屏幕空间流体渲染的方法渲染飞溅出来的水沫、喷泉等。

4.2.1 水面渲染

在给定了水面高度场网格模型后，对水面的渲染主要是需要考虑折射和反射效果。虽然在实时渲染中，基于物理地模拟水面的折射、反射有一定的困难，但我们可以先通过渲染场景到折射、反射纹理，然后根据这两张纹理来近似模拟，最终的效果见图 4.9。折射、反射虽然物理原理不同，但代码实现过程非常类似，所以本文重点说明折射的实现过程。折射的渲染过程如下

- 1) 渲染场景中除水面外的其他物体到折射纹理。
- 2) 最终渲染水面时, 计算没有扰动的折射纹理坐标, 并结合水面法向量和深度计算折射扰动。

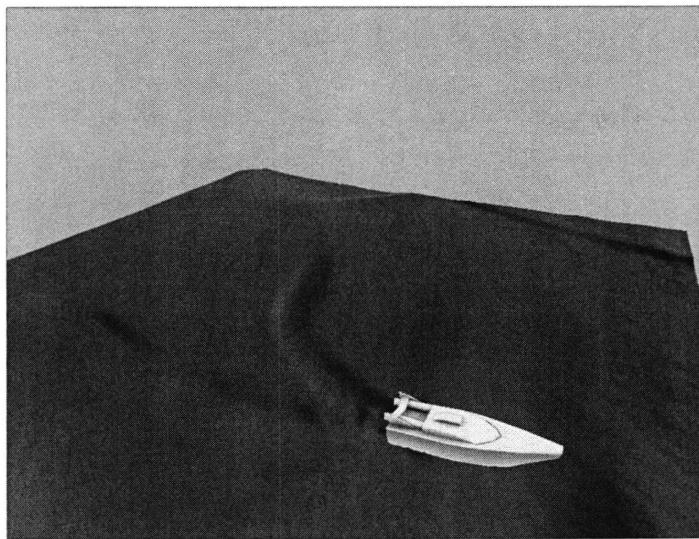


图 4.9 水面渲染效果图

4.2.2 屏幕空间流体渲染

在我们的项目中, 除了高度场水面网格渲染外, 还有喷泉、飞溅出来的水。这些都无法直接使用高度场的方法渲染。在我们的项目中喷泉、飞溅出来的水都是以粒子的方式物理模拟的, 类似基于粒子的流体模拟方法。我们使用 Van der Laan 等人^[18]提出的屏幕空间流体渲染方法来渲染这种粒子。整个算法大致分为以下几个步骤:

- 1) 渲染场景背景, 得到背景深度和颜色纹理, 用于计算折射。
- 2) 渲染流体粒子得到深度图, 用屏幕空间曲率流平滑深度。
- 3) 渲染流体粒子计算厚度图。
- 4) 最终根据平滑后的深度图渲染流体。

4.2.2.1 屏幕空间曲率流平滑

屏幕空间流体渲染方法的优势在于能够快速地从粒子中抽取网格平面，相比于传统的基于 Marching Cubes 算法^[19]抽取网格等势面的方法，屏幕空间的方法更加快速高效。该方法首先以球体的形状一个个渲染粒子得到深度图，然后平滑深度，从平滑后的深度图中可以直接抽取出流体表面并计算出法向量，如图 4.10。

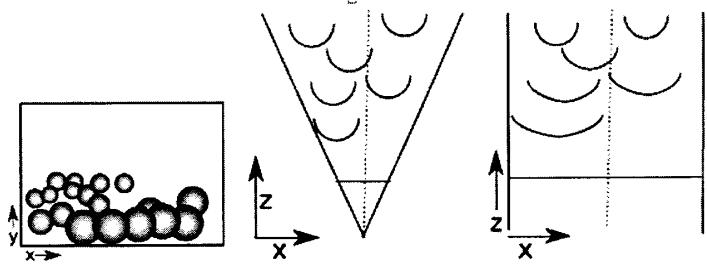


图 4.10 屏幕空间流体算法原理图⁷

深度图是粒子按球的方式一个个渲染得到的，如果不平滑深度，流体表面球的形状清晰可见。Van der Laan 使用了基于屏幕空间表面曲率流（Screen Space Curvature Flow）的平滑方法，通过这种方法流体表面会根据平均曲率沿着法向量方向不断变化，减少曲率突变的部分，最终得到一个平滑的曲面。在我们的应用中，表面是从深度缓存中抽取，只能沿着 Z 方向上移动，因此

$$\frac{\partial z}{\partial t} = H$$

其中 t 是平滑时间间隔步长， H 是平均曲率，对于三维空间中的曲面，

$$2H = \nabla \cdot \hat{n} \quad \text{公式 (4.3)}$$

其中流体表面单位法向量 \hat{n} 可以通过对摄像机空间位置 P 的 X、Y 方向导数做叉积得到。

$$\hat{n}(x, y) = \frac{n(x, y)}{|n(x, y)|} = \frac{(-C_y \frac{\partial z}{\partial x} - C_x \frac{\partial z}{\partial y}, C_y z)^T}{\sqrt{D}} \quad \text{公式 (4.4)}$$

⁷ 数据来源 Screen space fluid rendering with curvature flow^[18]

$$D = C_y^2 \left(\frac{\partial z}{\partial x} \right)^2 + C_x^2 \left(\frac{\partial z}{\partial y} \right)^2 + C_x^2 C_y^2 z^2$$

C_x, C_y 是用于根据深度重建摄像机空间顶点坐标的参数，可以根据视口尺寸和投影矩阵的 fov 计算。

$$C_x = \frac{2}{V_x \tan(\frac{\text{fov}}{2})} \quad C_y = \frac{2}{V_y \tan(\frac{\text{fov}}{2})}$$

把公式 4.4 带入公式 4.3，可以得到

$$2H = \frac{\partial n_x}{\partial x} + \frac{\partial n_y}{\partial y} = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}} \quad \text{公式 (4.5)}$$

其中

$$E_x = \frac{1}{2} \frac{\partial z}{\partial x} \frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2} D \quad E_y = \frac{1}{2} \frac{\partial z}{\partial y} \frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2} D$$

在像素着色器中，就可以使用上面的公式对粒子深度进行平滑，平滑的效果取决于迭代的次数，迭代的次数越多平滑效果越好，最终的效果如图 4.11 所示。

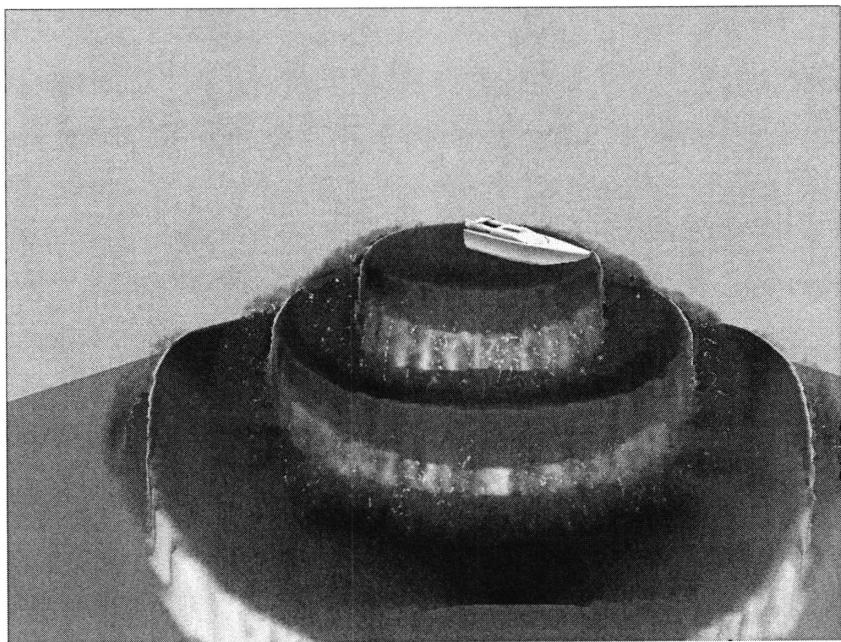


图 4.11 流体渲染效果图，其中喷出来的是喷泉，用屏幕空间的流体方法渲染

4.2.2.2 流体厚度

通常情况下，流体会有半透明的效果，其透明程度不仅跟流体材质属性有关，还跟流体的厚度有关。在最终渲染流体的时候，我们需要用厚度来控制流体的折射效果，即透明程度。基于粒子的流体模型中，流体粒子是以球的形状渲染的，流体的厚度可以通过累积叠加球的个数计算，每个球都有直径。所以在渲染流体厚度贴图时，我们使用 Additive Blending 的混合模式，并且打开深度测试，这样只有在场景其他物体前面的流体粒子被渲染。厚度的计算公式如下

$$T(x, y) = \sum_{i=0}^n d\left(\frac{x - x_i}{\sigma_i}, \frac{y - y_i}{\sigma_i}\right)$$

其中 x, y 是当前像素的屏幕坐标， x_i, y_i 是流体粒子投影到屏幕上的坐标， σ_i 是粒子在屏幕上的大小， d 是厚度内核函数。

4.3 本章小结

本章首先介绍了人脸皮肤的渲染，纹理空间的次表面散射算法的实现。通过多次高斯模糊人脸光照辐射度贴图来模拟次表面散射效果，还结合半透明阴影贴图算法来模拟耳朵等部位的透光效果。接着介绍了屏幕空间的流体渲染方法，该方法适用于用粒子模拟的流体，能够快速地抽取出流体表面，从而提高渲染效率。

第5章 Tile Based 渲染

5.1 延迟渲染

在实时渲染中，想要渲染出逼真的效果必须同时满足三个要素：精细的几何模型、高质量的材质纹理和真实的光照。前两者主要依靠美术人员的艺术创作，而后者依赖于真实的光照模型和相应的光照算法。基于全局光照的渲染技术如光线跟踪可以渲染出非常逼真的图像，但由于其昂贵的计算开销，至今难以达到实时渲染的效果。而传统的局部光照技术如前向渲染（forward rendering）只能支持一定数量的动态光源。前向渲染无法渲染出逼真的室内场景和夜晚效果，真实的场景需要大量的灯光来模拟。并且由于无法在实时渲染上使用全局光照技术，一般在游戏场景中，不能被光源直接照射到的地方，需要时用点光源进行补光。延迟渲染是近几年在实时渲染中被广泛使用的一项技术，支持多动态光源的实时渲染。目前市面上主流的游戏引擎都支持延迟渲染，如 Unreal、Unity3d 等。延迟渲染技术通过把三维场景中计算着色所需的法线、深度和材质等信息渲染到 G-Buffer 中，然后根据 G-Buffer 的信息在屏幕空间进行着色计算。延迟渲染最大的优势是光源的光照着色计算跟场景的复杂度无关，只需要渲染一次场景生成 G-Buffer，后面每个光源的光照着色计算只在被其所影响到的像素上进行，把复杂度从前向渲染的 $O(\text{geometry} * \text{lights})$ 缩减到了 $O(\text{geometry} + \text{lights})$ 。

最早在游戏中使用的延迟渲染技术是延迟着色（Deferred Shading），Deferred Shading 是一个两个批次的渲染过程，第一个批次渲染生成场景的 G-Buffer，第二个批次逐个光源地计算光照和着色，通常是渲染光源对应的包围体，点光源对应球体，聚光灯对应圆锥。Deferred Shading 把光照和着色计算耦合在同一个渲染批次进行，这就要求 G-Buffer 存储所有光照着色阶段所需的信息，包括法线、Diffuse 材质，Specular 材质等，在硬件上必须有同时渲染到多个纹理（MRT）的支持。

延迟光照（Deferred Lighting）对延迟着色做了一些修改，从而减少了 GPU

带宽的占用。延迟光照是一个三个批次的渲染过程。第一个批次渲染场景生成 G-Buffer，不同于 Deferred Shading，只需要输出光照计算所需的信息，如法线和材质的粗糙度，存到一张纹理上。第二个批次逐个光源计算光照，生成光照辐射度纹理。第三个批次需要再渲染一遍场景，结合光照辐射度纹理进行着色计算。Deferred Lighting 甚至不需要同时渲染到多个纹理（MRT）的硬件支持，被广泛使用在低端硬件设备上，如手机、平板等移动设备上。缺点就是需要额外再渲染一遍场景，增加了一定的开销。目前流行的游戏引擎 Unity3d 的延迟渲染就使用了这项技术。但随着移动设备图形硬件的不断提升，Deferred Lighting 渐渐的失去其优势。目前 Unity3d 官方正在发起投票，征求下一版本 Unity 5.0 切换到 Deferred Shading 上。

本渲染引擎在延迟光照的基础上做了一些改进，分离了光照和着色计算，可以一定程度上提高效果。改进的延迟光照也是三个批次的渲染过程，第一次批次渲染场景生成 G-Buffer，需要只输出法线和材质信息，可以打包成两张纹理。第二个批次逐个光源计算光照，生成光照辐射纹理。但只需要访保存法线和材质高光指数的纹理，降低了纹理访问的开销。第三个批次着色计算，不需要再渲染一遍场景，只需渲染一个全屏幕的矩形，计算着色所需的信息都已经存在 G-Buffer 和辐射度纹理里了。此外，在计算光照阶段，还使用了基于 Stencil 的 Light Volume 优化，这个优化类似于 Shadow Volume 算法，可以找出被光源包围体覆盖的区域，真正达到只在光源包围体覆盖的可见物体表面进行光照计算。

5.2 Tile Based 渲染

前面小节提到，传统的前向渲染无法同时支持大量动态光源，而延迟渲染通过先渲染场景生成 G-Buffer，再在屏幕空间进行光照着色计算的方法，可以实时渲染多动态光源的场景。但传统的延迟渲染技术最多只能支持上百个动态光源，无法实时地处理更多数量的光源。庆幸的是，随着计算机图形硬件的不断提升，GPU 的并行计算能力变得更加通用，两个主流的渲染 API 也都增加了对通用并行计算的支持，DirectX 11 从 Shader Model 5.0 开始支持 DirectCompute，OpenGL

从 4.3 版本开始支持 Compute Shader。这给多光源实时动态渲染打开了一扇新的大门，随着而来的是各种基于 Compute Shader 的高级渲染算法孕育而生。其中一个非常流行的思想是把屏幕分块后并行处理，Tile Based 渲染就是基于这样一种思想的算法。

Tile Based 渲染的方法不仅可以用在延迟渲染上，还可以用在前向渲染上，使得前向渲染也能同时支持大量动态光源。前向渲染、延迟渲染和 Tile Based 渲染这些术语会在本章中反复出现，我们先分别对这些术语做出定义。前向渲染是指光照着色计算发生在模型被光栅化渲染的同一个批次里。延迟渲染是两个阶段的过程，第一个阶段模型被光栅化渲染生成 G-Buffer，第二个阶段根据 G-Buffer 进行光照着色计算。而 Tile Based 渲染是一种新的算法框架，既可以用在延迟渲染，也可以用在前向渲染，大致的步骤如下：

- 1) 把屏幕分成 $N \times N$ 的 Tile
- 2) 为每个 Tile 计算最大/最小的深度值
- 3) 把光源分配到每个 Tile
- 4) 对每个像素，根据当前像素所处 Tile 得到的光源计算光照着色

其中第四个步骤，在延迟渲染中，每个像素光照着色的信息可以直接从 G-Buffer 中获取。而在前向渲染中，需要通过再光栅化渲染一遍场景才能得到每个像素。

5.2.1 Tiled Light Culling

Tile Based 渲染中，最重要的一个步骤就是 Tiled Light Culling，也就是上面提到的前三个步骤，主要任务是把光源分配到所影响到的 Tile 中去。这是 Tile 在三维空间中对每个光源包围体进行剔除的过程。在 Compute Shader 出现以前，在 Pixel Shader 实现 Tiled Lighting Culling 十分困难。近几年图形硬件发展迅速，加上 Compute Shader 的支持，Tile Based 渲染才渐渐被应用于实时渲染上。

5.2.1.1 屏幕划分

屏幕划分把屏幕分成 $N \times N$ 的格子，每个格子称之为 Tile。这个过程在 Compute Shader 里很容易实现，可以给每个 Tile 分配一个二维的 Thread Group。Thread Group 里每个线程对应一个像素。划分屏幕时，Tile 的大小需要合理选择。如果 Tile 划分的过大，会导致很多光源被分配到同一个 Tile 里，而实际上 Tile 的大多数像素并没有被这些光源照射到。如果 Tile 划分的过小，导致 Tile 的数量过多，整个 Light Culling 的计算开销提高，需要更多的 GPU 显存。本渲染引擎的实现是采用 32×32 的 Tile。

5.2.1.2 计算最大/最小深度

为 Tile 计算 Z 方向上的深度范围，可以得到一个更加紧凑的包围盒，在光源分配阶段剔除更多的光源包围体，从而减少影响该 Tile 的光源数量。在 GPU Compute 编程模型里，属于同一个 Thread Group 里的线程可以使用共享变量并在 Group 内同步线程。计算 Tile 的深度范围很容易在 Compute Shader 里实现，Tile 中每个像素都会分配一个 GPU 线程，每个线程计算完自己的深度后，使用 GPU 原子操作同步最大/最小深度共享变量。

5.2.1.3 光源分配

每个 Tile 在三维空间中的包围体类似于摄像机的视锥，在二维平面上的简化如图 5.1 所示，其中图 5.1a 中绿颜色是每个 Tile 的边界，图 5.1b 是结合 Tile 的最大/最小深度计算得到的视锥。类似于摄像机视锥剔除，需要计算 Tile 视锥六个面的平面方程。有两种方法计算 Tile 视锥的平面方程：1) 分别计算出视锥四个角点的三维坐标位置，根据平面方程计算。2) 根据 Tile 的投影矩阵，直接推导平面方程，这种方法计算量最少。每个 Tile 的投影矩阵其实就是 Off-Center 透视投影，可以根据摄像机的投影矩阵推导过来。

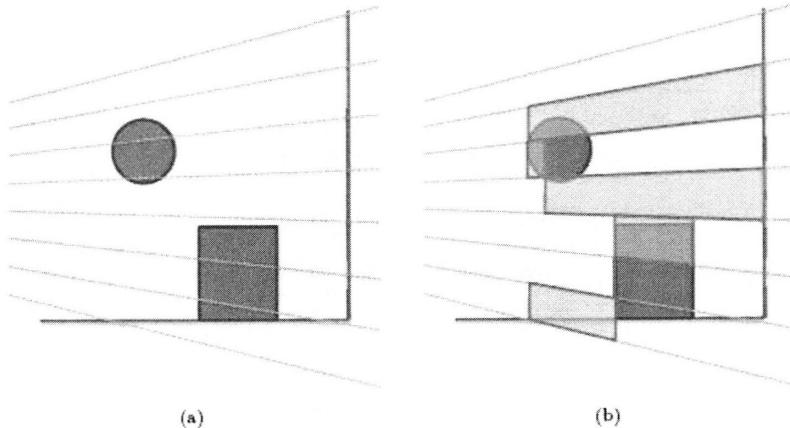


图 5.1 Light Cull 的 2D 展示⁸, (a) 摄像机在左侧, 绿色是 Tile 的边框, (b) 根据 Tile 的深度范围, 可以为每个 Tile 创建 Frustum.

得到了 Tile 的视锥后, 光源分配就是对每个 Tile, 遍历所有光源, 进行视锥剔除, 把没有被剔除掉的光源加到该 Tile 对应的光源列表中。但在 GPU 上, 这也是个并行计算的模型, 本渲染引擎采用的是 32×32 的 Thread Group, 也就是说可以同时并行处理 256 个光源的剔除。

未被剔除掉的光源需要添加到 Tile 的光源列表, 可以使用 Thread Group 的共享存储, 每个 Tile 在共享储存中都有一个光源列表。这个过程需要用到 GPU 的原子操作, 每个线程往队列里添加光源时, 首先需要用原子操作 atomicAdd 获得对应的队列索引, 再按照索引更新光源队列。

5.2.2 Tile Based 延迟渲染^[20]

如前面介绍的, Tile Based 渲染很容易应用在延迟渲染上。Tile Based 延迟渲染也是一个两个批次的渲染过程, 第一个批次同传统的延迟渲染一样, 渲染一遍场景生成 G-Buffer。第二个批次, 完全在 Compute Shader 上执行, 先进行上节介

⁸ GPU PRO 4 [22]

绍的 Tiled Light Culling，得到每个 Tile 的光源列表。所有能影响到 Tile 中像素的光源都已经收集在光源列表中了，最终的光照计算只需要遍历列表中的所有光源进行着色计算即可。图 5.2 是 1024 个点光源在 Sponza 场景下的光照效果。

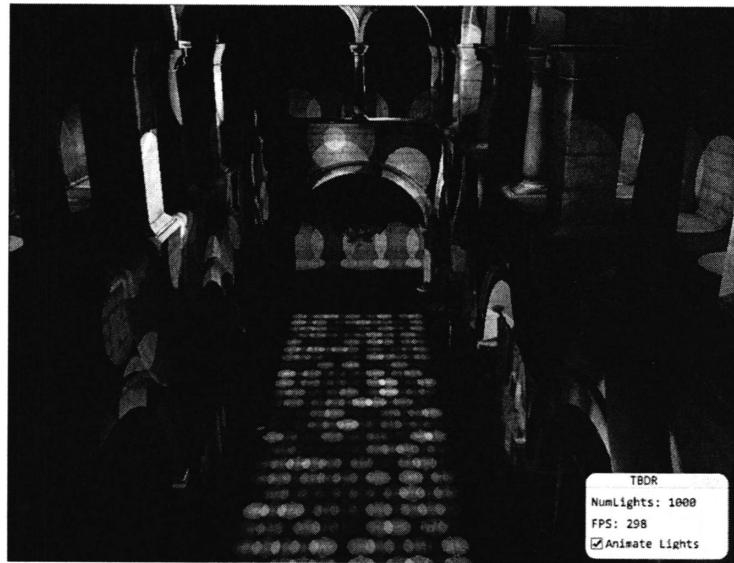


图 5.2 Tile Based Deferred Shading 效果图

Tile Based 延迟渲染相比于传统的延迟渲染，有许多优点：

- 1) 减少了带宽和 G-Buffer 纹理读取的次数和写像素的次数。传统的延迟渲染如果光源之间有相互重合的话，在计算光照着色时，G-Buffer 纹理读取和像素颜色值输出都要被重复。而 Tile Based 延迟渲染只需要读 G-Buffer 一次，写像素一次。
- 2) 只需一次 Compute Shader 渲染命令，而传统的延迟渲染需要逐个渲染光源的几何包围体。

当然 Tile Based 延迟也有跟传统的延迟渲染一样的缺点：

- 1) 无法渲染半透明物体，因为 G-Buffer 只能储存场景中的一层物体属性。
- 2) 对硬件 MSAA 的支持不是很友好，G-Buffer 的存储需求和计算带宽都要成倍增长。

5.2.3 Tile Based 前向渲染^[21,22]

随着玩家对游戏画质的要求不断提高，传统的前向渲染已经很难继续使用在必须支持多动态光源的游戏渲染中。过去十年，大部分游戏都在使用延迟渲染，延迟渲染已经成为次时代游戏的唯一选择。但随着图形硬件的发展，Tile Base 渲染的出现再一次把前向渲染推向了新的高度。AMD 显卡公司为其 AMD Radeon 系列显卡推出示例程序 Leo Demo，使用了 Tile Based 前向渲染，画面效果非常华丽，见图 5.3。



图 5.3 AMD Leo Demo 画面

AMD 把这项技术称作 Forward+，其实质是把 Tile Based 算法应用到了前向渲染中。在前向渲染中使用 Tile Based 算法框架需要一定的修改。

首先在 Tiled Light Culling 阶段，需要计算每个 Tile 的最大最小深度，这就需要提供场景的深度缓存。Tile Based 延迟渲染会在 G-Buffer 阶段附带输出深度缓存，而前向渲染则没有这个过程。所以需要一个 Pre-Z 渲染批次，预先渲染一次场景生成深度缓存。虽然 Pre-Z 渲染会有一定的开销，但由于只输出深度缓存，不需要像素着色，所以 Pre-Z 的开销很低。并且增加 Pre-Z 批次还有额外的好处，

在最终前向渲染的时候，可以使用 Pre-Z 生成的深度缓存做 early-Z 深度测试，这会大大减少最终需要计算光照着色的 Sample 个数。

其次在 Tiled Light Culling 阶段，需要把每个 Tile 的光源列表保存到一个全局的列表中。这需要使用 DirectX 11 的高级特性，UnorderedAccessViews (UAV)。通过 UAV 可以用数组的方式线性存储 StructureBuffer 或 TextureBuffer。我们可以把每个 Tile 的光源列表即光源的索引，保存到全局的 UAV 缓存中。在最后前向渲染的时候，结合材质进行光照计算着色计算。所以 Tile Based 前向渲染也是一个三个批次的渲染过程。

Tile Based 前向着色最主要的优点是，可以像传统的前向着色一样使用透明混合，支持半透明物体的渲染。并且不需要延迟渲染的 G-Buffer，显存带宽占用也比较低，可以很好的支持硬件 MSAA。

5.3 渲染效率对比

本渲染引擎集成了上面提到的三种渲染方法，在引擎中抽象出 RenderPath 来对应三维场景的渲染过程，分别有 DeferredPath、TileDeferredPath 和 ForwardPlus 三个渲染过程。在实际应用时，开发者根据实际需求，选择其中一种 RenderPath 进行渲染。

图 5.4 是这三种渲染方法在 Sponza 场景下的性能对比图，可以发现，当光源数量不是很多的情况下，传统的延迟渲染方法相比于 Tile Based 方法还是具有优势的。但当光源数量很多时，传统的延迟渲染算法效率下降就很明显，这种情况下 Tile Based 方法的优势才体现出来。此外，也可以发现 Tile Based 前向渲染方法在本引擎的实现中效率更加高，未来也可以作为延迟渲染的替代。

通过对比，我们可以得出结论，在场景光源数量不是很多，大约 100 个以下的应用中，可以选择传统的延迟渲染性能更好。而当光源数量很多，达到成千上万时，Tile Based 方法相比于传统的延迟渲染就会有巨大的优势。

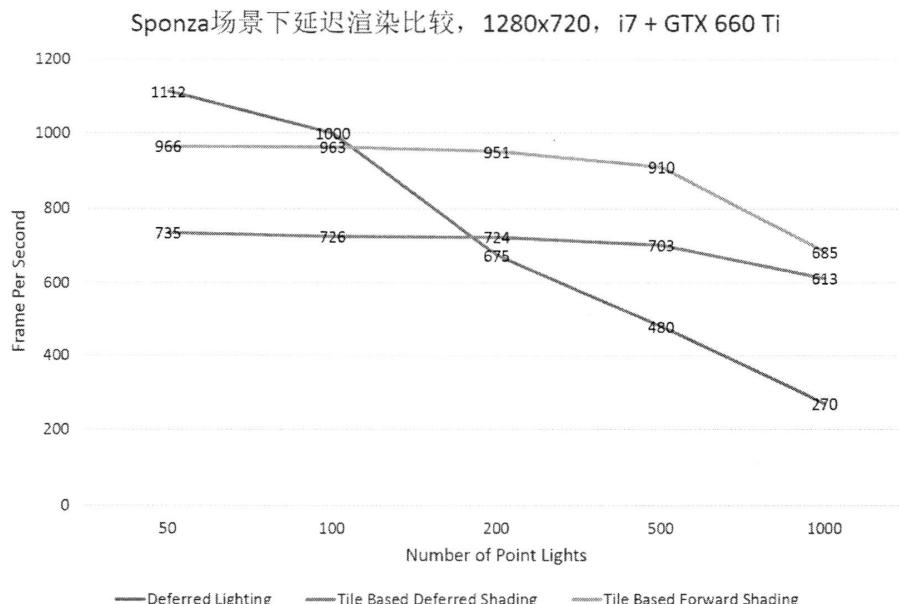


图 5.4 三种渲染方法的效率对比图

5.4 本章小结

本章介绍了几种实时渲染多动态光源场景的方法。传统的延迟渲染可以应用在光源数量不是很多的游戏场景里，或者图形硬件相对落后的移动设备上。Tile Based 延迟渲染则可以使用在次时代的三维游戏里，可渲染出逼真的光影效果。Tile Based 前向渲染虽然还没有被广泛使用，但其优势还是不可忽略的，未来有可能可以代替延迟渲染。本渲染引擎对这三种渲染方法都有实现，而且开发者可以在不修改其他代码的情况下，根据项目需求任意选取一种方法渲染。

第6章 阴影环境遮蔽和全局光照

第五章介绍了大量多动态光源的实时渲染方法，但只考虑了局部光照模型，也没有考虑阴影、环境遮蔽等光照效果。而这些效果是真实感渲染所必须的，不可或缺。本章将在第五章介绍的渲染方法上继续探索，在追求真实感渲染的道路上再迈进一步。

6.1 阴影渲染

高效稳定的阴影渲染一直都是实时渲染领域里面比较热门的一个方向。虽然实时阴影渲染已经有几十年的研究历史，但到目前为止还没有一个统治性的方法来解决真实感实时渲染里的阴影问题。早期游戏中的阴影渲染都是使用 Shadow Volumes 算法，该算法最早由 Franklin C.Crow^[25]在 1977 年写的一篇论文里提出。随着 id Software 公司的知名游戏 DOOM3 的发布，Shadow Volumes 算法成为了 FPS 游戏玩家和图形学爱好者争论的对象。而目前游戏中的阴影渲染都是以阴影贴图（Shadow Maps）^[26]的方法实现。

阴影贴图算法可以看作是一个信号重建的过程，类似于纹理贴图，只是这张纹理是实时地从三维场景中采样生成的。对 Shadow Maps 算法的改进主要也从两个方面入手，一方面就是阴影贴图的生成，采样的过程。另一方面是阴影贴图的滤波（Filtering），重构的过程。

本渲染引擎对阴影的渲染算法做了一定的实验研究，最后选择了一些效果相对较好的方法集成到了引擎中。

6.1.1 Cascaded Shadow Maps^[23]

在讨论阴影贴图算法之前，我们先来分析一下 Shadow Maps 锯齿的来源。通过图 6.1 中的空间几何关系，我们可以得到阴影贴图中大小为 ds 的一个纹素和投影平面上大小为 dp 的一个像素之间的对应关系，见公式 6.1。我们可以通过这个

比例来定性的分析 Shadow-Maps Aliasing，可以分为两个部分，透视锯齿（Perspective Aliasing）和投影锯齿（Projection Aliasing）。

$$\frac{dp}{ds} = \frac{1}{\tan \phi} \frac{dz}{zds} \frac{\cos \phi}{\cos \theta} \quad \text{公式 (6.1)}$$

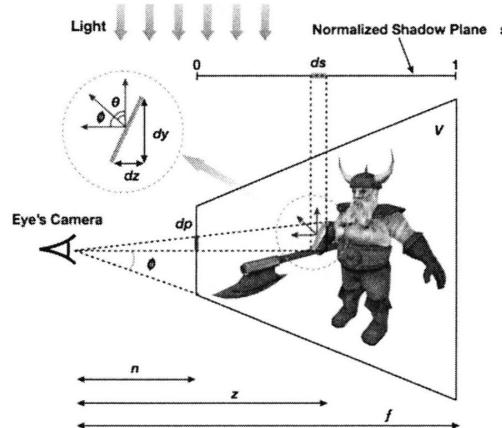


图 6.1 阴影贴图算法锯齿⁹

其中投影锯齿 $\cos \phi / \cos \theta$ 主要跟模型的几何细节有关，减少这种锯齿需要分析场景的几何细节。而透视锯齿 dz / zds 则跟摄像机近大远小的透视投影有关系，跟场景无关，离摄像机越近的像素需要更高的阴影贴图分辨率。如图 6.2 中的透视阴影锯齿，是因为屏幕上好几个像素对应到了阴影贴图中同一个纹素。

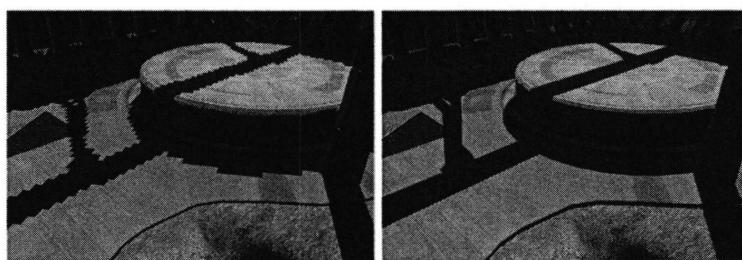


图 6.2 阴影贴图算法透视锯齿¹⁰

⁹ http://http.developer.nvidia.com/GPUGems3/gpugems3_ch10.html

¹⁰ [http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx)

目前解决透视锯齿最有效的方式是级联阴影贴图（Cascaded Shadow Maps, CSM）^[23]，这种方法在游戏中已经占据统治地位。CSM 的思想很简单，摄像机视锥的不同区域需要不同分辨率的阴影贴图，离摄像机近的物体相比于远的物体需要更高的阴影贴图分辨率。通过把视锥按深度划分为多个平截头体（frusta），分别为每个平截头体生成一张阴影贴图，在最终渲染场景的时候从中选择一张最合适的计算阴影，如图 6.3 所示。

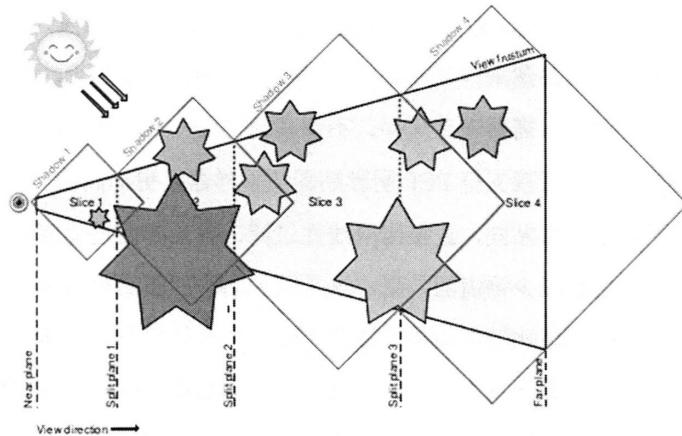


图 6.3 PSSM frustum 划分图¹¹

CSM 是基于这类思想的一种算法框架，而具体的算法实现即平截头体划分策略有多种方法。其中由 Fan Zhang 等人提出的 Parallel-Split Shadow Maps^[24]在业界得到了广泛的使用，本渲染引擎的集成的阴影渲染算法也是基于 PSSM 的。PSSM 按照如下公式划分平截头体。

$$C_i = \lambda C_i^{\log} + (1 - \lambda) C_i^{uni} \quad 0 < \lambda < 1$$

$$C_i^{\log} = n \left(\frac{f}{n}\right)^{i/m} \quad C_i^{uni} = n + (f - n) \frac{i}{m}$$

其中 C_i^{\log} 对数划分策略是理论上最优的方法，可以保证投影锯齿在这个深度范围不变，维持在一个常量上。但这种划分策略会导致前几个平截头体非常小，没有实用价值。 C_i^{uni} 是均匀划分策略，即在摄像机整个深度范围内等距离划分。

¹¹ Shadows & Decals: D3D10 Techniques in Frostbite (GDC'09)

PSSM 结合这两种方法，并且可以按照实际场景需求调整划分的权重 λ ，在实际应用中取得了不错的效果。

6.1.2 Shadow Maps Filtering

另一类 Shadow Maps 算法是从阴影贴图采样方面改进的，这类算法主要有 PCF(Percentage-Closer Filtering)^[27]、VSM(Variance Shadow Maps)^[28,29]、ESM(Exponential Shadow Maps)^[30]、EVSM(Exponential Variance Shadow Maps) 等。本渲染引擎实现的阴影贴图采样算法是 Poisson Disc PCF。PCF 算法相比于其他方法，更加节省带宽，只需要深度缓存。不像其他方法一样，需要输出多个分量到纹理。目前显卡硬件直接支持 PCF 阴影贴图的采样器，更加高效。

阴影贴图不同于纹理贴图，直接用双线性过滤、各向异性过滤或者预采样生成 Mipmap 都不能起到减少锯齿的效果。这是因为阴影贴图储存的是深度信息，阴影的计算实际上是深度比较的结果。这个比较过程还依赖每个像素在光源坐标系下的深度，这个值是无法预先知道的。PCF 的思想是不直接过滤深度信息，而是过滤深度比较的结果。PCF 阴影贴图在游戏界已经被广泛使用，如 Crytek 公司的 CryEngine 3 使用的就是 Poisson Disk PCF。

PCF 阴影贴图配合 Possion Disk 采样可以取得非常好的阴影效果，甚至还能模拟软影效果。使用 Possion Disk 采样可以用更少的采样点取得规则采样的效果。具体实现的时候，我们可以预先用 Possion Disk 分布生成采样点，保存在查找表纹理或者常量缓存(Constant Buffer)中。PCF 阴影贴图的高效实现还得益于硬件双线性 PCF 过滤器的支持，DirectX 11 支持比较过滤器(Comparison Filters)，这使得可以直接在硬件上实现 PCF 双线性过滤、各向异性过滤。

本渲染引擎最终也选择了 Possion Disk PCF 算法作为阴影贴图的过滤方法，后期还将考虑转换到效果更加好的 EVSM，但 EVSM 算法的带宽占用要比 PCF 高很多。图 6.4 是在渲染场景时 CSM 阴影贴图的选择，可以看到场景不同的区域选择了不同的阴影贴图。图 6.5 是结合了 PSSM 和 PCF 的阴影渲染效果图。



图 6.4 CSM 阴影贴图选择

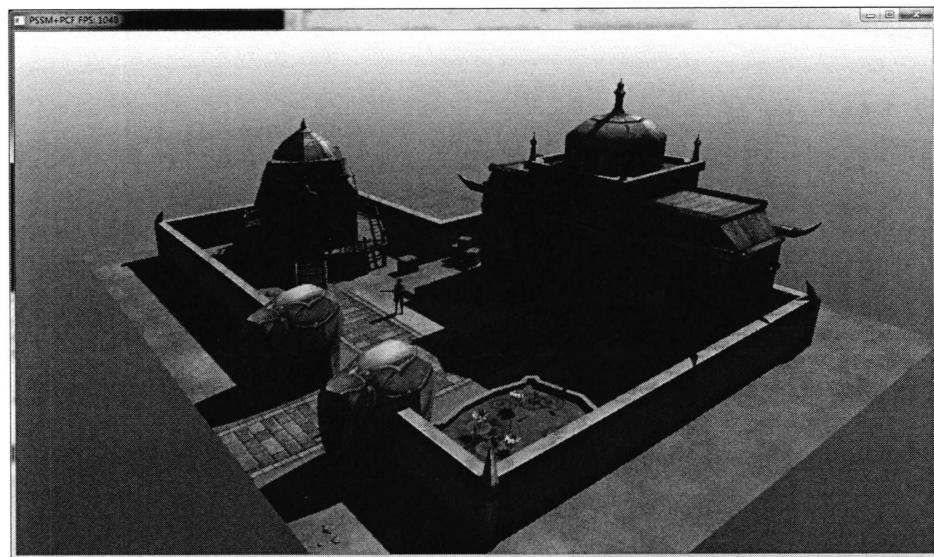


图 6.5 CSM 阴影算法效果图

6.2 环境遮蔽

环境遮蔽是真实感实时渲染中非常重要的一种视觉效果，用于描述物体间的相互遮挡关系，用较低的计算代价来获得类似全局光照的效果，最早由 Crytek 在 2007 年底开发并推出的游戏《孤岛危机》中使用。Crytek 提出的屏幕空间环境遮蔽(SSAO)算法^[31]与场景的复杂度无关，无需预处理，可以适用于动态场景。屏幕空间环境遮蔽通过使用后处理渲染中的一些常用信息如深度、法线，将传统基于三维空间的 AO 计算转换到屏幕空间，极大地提高了计算效率，从而使得这项技术在游戏中的应用变得实际可行。由于其在《孤岛危机》中的惊艳表现，环境遮蔽已经成为次世代高画质游戏必不可缺的一项技术。屏幕空间环境遮蔽实现效果的优劣主要取决于 AO 计算方法和采样策略，不同的游戏引擎在 SSAO 的实现细节上不尽相同。本渲染引擎集成了两种 SSAO 算法，其中一种是由 Morgan McGuire 等人提出的 Alchemy AO^[32,33]，另一种是目前在国外大制作游戏中广泛使用的 HBAO+^[34]。

6.2.1 Alchemy AO

Crytek 最早提出的 AO 是指 Ambient Occlusion，其 AO 可用公式 6-1 计算。

$$AO(p) = \frac{1}{\pi} \int_{\Omega} V(p, \omega)(n \cdot \omega) d\omega \quad \text{公式 (6.2)}$$

其中 V 是可见性函数，如果光线被场景中其他物体遮挡，值为 0，否则为 1。但公式 6-1 并没有对光线的范围做出约束，光线可以无限延伸。这并不是一个实际可行的计算方法，因为远距离的物体的遮挡不应该起很大的作用。为了更好的区分近距离遮挡和远距离遮挡，Zhukov 等人提出了 Ambient Obscurance^[35]。通过一个跟距离相关的衰减函数代替了公式 6.2 中的 V 。

$$AO^*(p) = \frac{1}{\pi} \int_{\Omega} \rho(|p\omega - p|)(n \cdot \omega) d\omega \quad \text{公式(6.3)}$$

其中衰减函数 ρ 应该满足如下两个性质：

- 1) ρ 应该随着距离 d 单调递增。

2) 存在一个上界 d_{max} , 对于任何大于 d_{max} 的距离, ρ 都为 1

· Alchemy AO 通过巧妙地选择了一个衰减函数来消减公式 6.3 中的计算项, 从而提高计算效率。McGuire 选择的衰减函数 P 如下。

$$\rho(d) = \frac{u \cdot d}{\max(u, d)^2} \quad \text{公式 (6.4)}$$

McGuire 选择的衰减函数 ρ 随着 d 的增大, ρ 趋近于 0 不是 1, 所以带入衰减函数 ρ 后, 真正的 AO 等于公式 6.5。

$$AO = 1 - \frac{u}{\pi} \int_{\Omega} \frac{d \cdot (n \cdot \omega)}{\max(u, d)^2} d\omega \quad \text{公式 (6.5)}$$

定义 $v = d \cdot \omega$, 根据 $v \cdot v = |v|^2 = d^2$, 公式 6.5 可以化简为

$$AO = 1 - \frac{u}{\pi} \int_{\Omega} \frac{(v \cdot n)}{\max(u^2, v \cdot v)} d\omega \quad \text{公式 (6.6)}$$

剩下的工作就是用蒙特卡洛积分的方法对公式 6.6 进行计算, McGuire 又对公式 6.6 进一步化简得到公式 6.7。

$$AO = 1 - \frac{1}{N} \sum_{n=1}^N \frac{\max(v_n \cdot n + \beta, 0)}{v_n \cdot v_n + \epsilon} \quad \text{公式 (6.7)}$$

具体实现时, Alchemy AO 使用的是螺旋扰乱式采样, 即在当前像素的半球空间中, 以螺旋的方式从内往外选择采样点, 并为每个像素增加随机旋转。此外, McGuire 还使用了 Hierarchical Z-Buffer 来提高采样效率, 增加一个 Hierarchical Z-Pass 计算摄像机空间深度值, 并生成 Mipmap。这是因为在螺旋扰乱式采样的过程中, 需要多次采样深度并计算摄像机空间深度值, 通过预先创建 Mipmap 并在采样时直接使用粗糙层的深度值, 可以提高纹理采样的 Cache 使用率, 从而提高效率。图 6.6 是 Alchemy AO 算法在 Sponza 场景下的效果图, 没有进行模糊, 但也可以发现 Alchemy AO 能够捕捉到场景中的某些模型细节, 如狮子头模型, 有非常好的明暗效果。

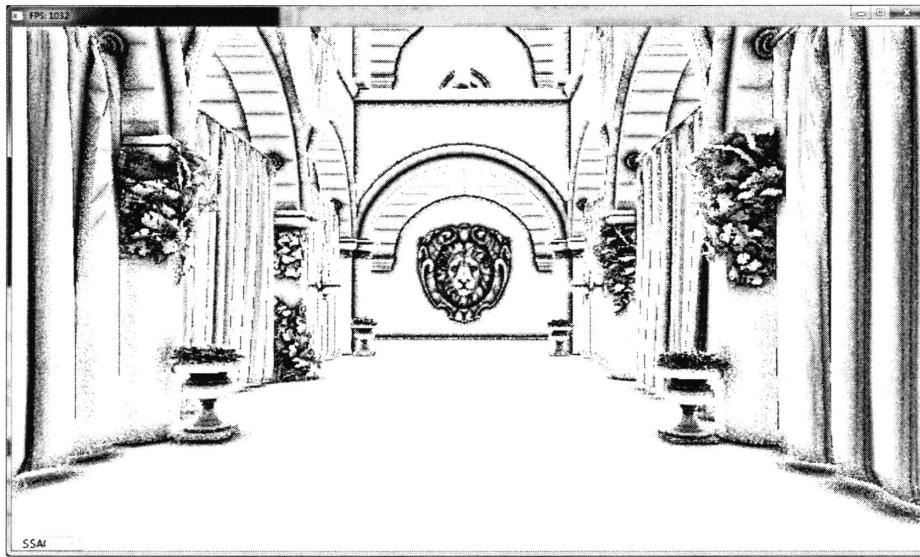


图 6.6 Alchemy AO 在 Sponza 场景下的效果图，未平滑

6.2.2 HBAO+

另外一种高质量的环境遮蔽算法是 HBAO+，这是 Nvidia 对其在 2008 年的 SIGGRAPH 会议上提出的 HBAO 算法^[34]的改良，HBAO 通过计算如图 6.7a 所示的水平方向可见角度来近似估计 AO，即没有被周围物体遮挡住的角度。HBAO 假定 P 点周围都是连续的高度场，所有不在水平可见角度内的光线都被周围高度场给遮挡了。

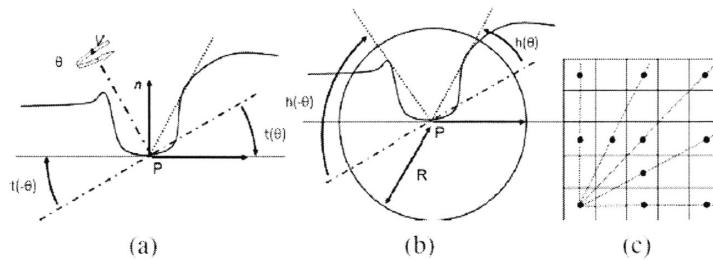


图 6.7 HBAO 算法图^[34]，其中 (a) θ 是绕 V 的旋转角， $t(\theta)$ 是高度场切平面上的水平仰角。
(b) 所有小于 $h(\theta)$ 的区域都是被遮挡住的。(c) 采样的方法。

有了连续高度场的假定，我们知道所有水平角度低于 $h(\theta)$ 的光线都被遮挡住，所有角度高于 $h(\theta)$ 的光线都是可见的。所以我们可以通过如下公式计算 AO。

$$AO = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} (\sin(h(\theta)) - \sin(t(\theta))) W(\theta) d\theta \quad \text{公式 (6.8)}$$

其中 $W(\theta)$ 是衰减函数，使用 $W(\theta) = \max(0, 1 - d/R)$ 。

HBAO+是 HBAO 算法的改进，Nvidia 没有公开其具体的实现细节，但发布了 HBAO+ SDK，开发者可以很方便的集成到自己的应用中。目前使用 HBAO+的游戏有《细胞分裂：黑名单》、《刺客信条：团结》等。本渲染引擎也集成了 Nvidia 的 HBAO+ SDK，在 Sponza 场景中的效果如图 6.8 所示。



图 6.8 HBAO+在 Sponza 场景下的效果图

6.3 基于 Deep G-Buffers 的快速全局光照算法

全局光照是真实感绘制中不可缺少的一部分，目前大部分游戏都只是在用直接光照，或者使用预算算的全局光照算法如光照贴图等。实时地支持动态场景动态光源的全局光照一直都研究的热点。随着 GPU 计算能力的不断增强，在 GPU 上实时光线跟踪的呼声也越来越高，Cyril Crassin^[36]提出了 Sparse Voxel Octree (SVO)，通过 SVO 来储存场景，并进行 Voxel Cone Tracing。该算法虽然能取得

不错的全局光照效果，但计算代价依然很高，不太适合当前的硬件设备，同时也不能很好的跟目前游戏引擎延迟光照的渲染光线结合。可以在游戏中实际应用的全局光照算法应该满足如下几个特性：

- 1) 能够快速计算，最好跟场景和光源复杂度无关。
- 2) 各种类型的光源都可以使用，而不仅仅是点光源。
- 3) 全局光照的结果可能只是近似估计，但不应该有明显的锯齿。
- 4) 能够很方便的集成到现有的渲染引擎中。

本渲染引擎实现了一种快速全局光照算法^[39]，通过对延迟渲染的 G-Buffer 做些修改，可以很快的计算出 One-Bounce 和 Multiple-Bounce 的间接光照，并且这种快速全局光照算法能够很好的跟引擎的延迟渲染管线相结合。该算法本质上也是基于图像空间光照辐射度收集(Gathering)的方法，我们通过创建两层 Deep G-Buffers 来弥补单层情况下部分场景信息缺失的问题，使得全局光照更加可信。

6. 3. 1 Deep G-Buffers

很多渲染算法通过使用多层场景信息来改进算法质量，如 Depth Peeling。同样屏幕空间的全局光照算法也可以通过利用多层场景信息来改进。在延迟渲染中，G-Buffer 其实就是场景信息的单层描述，描述的只是离摄像机最近的物体。单层 G-Buffer 只能储存观察者所看到的物体，在计算全局光照时，那些看不见的部分对光照的计算也有贡献。而整个场景的储存方法如 clip-space voxelization^[37,38]又需要耗费巨大的存储开销。双层 Deep G-Buffers 是对这两种方法的折中，如图 6.9 所示，可以一定程度上弥补单层 G-Buffer 场景信息上的缺失。

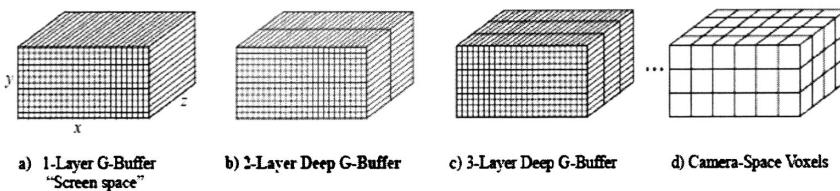


图 6.9 裁剪空间三维场景的表示, (a) 传统的 G-Buffer, 在分辨率上, xy 最高, z 最低, 只储存离摄像机最近的表面。(d) xyz 分辨率都中等, 每个 voxel 储存了表面的平均信息。(b) 就是本论文全局光照算法使用的双层 Deep G-Buffer。

多层 Deep G-Buffers 已经在延迟渲染中有所应用, 如用多层 Deep G-Buffers 来渲染半透明物体。本渲染引擎使用了双层 Deep G-Buffers, 并对每层 G-Buffer 之间的间距有所限制。这是因为现实生活中物体几何模型都非常复杂, 有很多细节。传统的 Depth Peeling 捕捉到的第二层 G-Buffer, 仍然会有当前物体的一些模型细节, 这些细节会遮挡真正对全局光照计算有用的物体, 所以我们对每层 G-Buffer 之间的间距设定了一个最小间隔 Δz 。每层 G-Buffer 的格式如图 6.10 所示。

Texture Format	Contents	
RG16	Screen-Space Velocity	
RG16	Normal	
RGBA8	Diffuse RGB	Unused
RGBA8	Glossy RGB	Glossy Exponent
Depth32F	Depth	
32 bits		

图 6.10 Deep G-Buffer 格式

6.3.2 间接光照

场景中物体表面上的点 X 受到的光照辐射度等于半球空间中所有入射光线强度的积分, 见公式 6.9。

$$E(X) = \int_{\Omega} \frac{B(Y)}{\pi} \max(\hat{n}_X \cdot \hat{\omega}, 0) d\hat{\omega} \quad \text{公式 (6.9)}$$

其中 X 是当前需要计算全局光照的点。Y 是 $\hat{\omega}$ 方向上离 X 最近的其他点， $\hat{\omega} = \frac{Y-X}{\|Y-X\|}$ 。

公式 6.9 的积分可以像环境遮蔽小节中使用到的屏幕空间蒙特卡洛积分的方法计算，见公式 6.10。

$$E(X) \approx \frac{2\pi}{M} \sum_{\text{samples}} B(Y) \max(\hat{\omega} \cdot \hat{n}_X, 0) \quad \text{公式 (6.10)}$$

在计算全局光照时，我们分别在两层 Deep G-Buffers 中取采样点，但只使用同时满足 $(\hat{\omega} \cdot \hat{n}_X) > 0$ 和 $(\hat{\omega} \cdot \hat{n}_Y) < 0$ 的采样点 M。因为只有这样的采样点 M 才朝向 X，会对 X 点接受的光照产生直接影响。

有了 X 点的入射光照辐射度，我们可以计算 X 点的出射辐射度 B(X)。

$$B(X) = E(X) \cdot \rho_X \cdot \text{boost}(\rho_X) \quad \text{公式 (6.11)}$$

其中 ρ_X 是物体材质的反射率， $\text{boost}(\rho_X)$ 是一个用来缩放反射率的函数， $\text{boost}(\rho_X) = 1$ 能量守恒，通过 $\text{boost}(\rho_X)$ 可以从美学上调节全局光照的 Color Bleeding 效果。

初始阶段的 B(Y) 可以同时直接光照计算。通过上面介绍的全局光照算法的辐射度收集(Gathering)方法，我们已经可以快速计算 One-Bounce 的间接光照。Multiple-Bounce 间接光照可以通过反复的应用公式 6.9 和 6.11 模拟得到。为了提高计算的效率，我们把 Multiple-Bounce 的间接光照分摊到多个连续帧里。每一帧在前一帧光照辐射度的基础上再计算一次间接光照，这样本来需要同一帧里计算的 N 次迭代被分摊到了连续的 N 帧里，从而提高了计算效率。图 6.11 是在 Sponza 场景中使用我们的全局光照算法的效果。图 6.12 是另外一个室内房间的效果图。

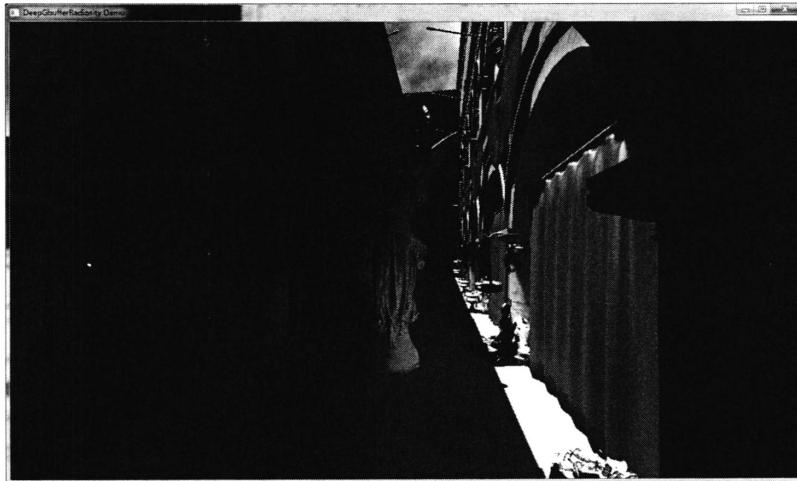


图 6.11 Sponza 场景下的全局光照效果



图 6.12 室内房间场景全局光照效果

6.4 本章小结

本章介绍了渲染引擎的动态光影系统，首先介绍了 CSM 阴影贴图算法框架，并采用 PSSM 作为具体实现。CSM 是目前室外大范围场景阴影渲染的标配。接着探索了高质量稳定的环境遮蔽实现，本渲染引擎集成了两种 SSAO，其中 Nvidia

的 HBAO+更加稳定，开发者可以根据需要选择一种。最后，我们在实时渲染中尝试了全局光照，基于 Deep G-Buffer 的快速全局光照算法可以很方便的集成到延迟渲染框架里。

第7章 总结与展望

7.1 工作总结

游戏的画面质量是给玩家的第一印象，画质好的游戏更加能吸引玩家，带来更好的游戏体验。近些年，国内玩家已经厌倦了画质一般的国产游戏，开始去体验欧美、日韩等外国的高品质游戏。国产游戏急需提高画面质量来重新赢得玩家的信任。所以自主研发先进的实时渲染引擎仍然是很有必要的。

本文的研究工作就是希望能够开发下一代实时渲染引擎，充分利用现代图形处理器的高级特性，探索试验先进的渲染算法，希望最终可以应用到国内网络游戏上去。本论文研究了基于 DirectX 11 和 OpenGL 4.x 的实时渲染引擎设计，并在这框架之上实现了各种高级渲染算法。

本论文主要的研究内容和成果主要体现在以下几个方面：

- 1) 完成了跨图形 API 的实时渲染引擎架构设计，包括灵活方便的特效系统和材质脚本，骨骼动画系统和素材管线等。
- 2) 实现了纹理空间的次表面散射算法，使用该算法可以逼真地渲染游戏人物角色的脸部皮肤，提高游戏画面的真实感。
- 3) 设计了一个水面、瀑布渲染系统，其中屏幕空间的流体渲染方法可以快速地渲染物理引擎模拟的流体。
- 4) 整合了实时渲染中大量多动态光源的渲染问题，提出了 Tile Based 的渲染框架，利用最新的硬件特性 Compute Shader，不论是延迟渲染还是前向渲染，都能解决大量多动态光源的渲染。
- 5) 提供了一整套实时光影解决方案，包括大范围场景的阴影渲染，高质量稳定的环境遮蔽算法和终极的全局光照算法。

本论文尝试的这些渲染算法是目前国内游戏引擎还没有集成的高级特性，但国外游戏早已在广泛使用这些算法。这才导致国内外游戏画面质量的巨大差距，希望对这些先进渲染算法的研究最终能应用到国产游戏上，提高游戏的画质，给

玩家带来更好的游戏体验。

7.2 未来展望

渲染引擎是个庞大复杂的系统，设计一款优质的实时渲染引擎，不仅需要前沿的图形学算法，还需要软件工程的设计思想。一个好的渲染引擎必须是开放的，可以很方便的集成其他系统，如物理引擎，游戏脚本等。实时渲染引擎设计的另外一个难点在于如何把各种先进的渲染算法高效的集成到框架里。由于开发的周期有限，目前本渲染引擎还有很多功能没有实现，主要包括如下方面：

- 1) 增加其他平台的支持，如移动平台、Linux 系统等。
- 2) 重构底层框架，切换到基于对象组件的架构设计，这种设计更加方便增加新的扩展，如物理引擎等。
- 3) 尝试更加高效的次表面散射算法，屏幕空间的次表面散射算法。
- 4) 继续完善引擎的渲染功能，还有很多渲染算法没有集成到引擎里，如 Image Based Lighting 等。
- 5) 结合实际游戏开发需求，开发引擎的工具链，包括场景编辑器、动画编辑器等。

参考文献

- [1] Gregory J. Game Engine Architecture[M]. CRC Press, 2009.
- [2] Akenine-Moller T, Haines E, Hoffman N. Real-Time Rendering, Third Edition[M]. AK, 2008.
- [3] Junker G. Pro OGRE 3D programming[M]. Apress, 2006.
- [4] Phong, Tuong B. Illumination for computer generated pictures[J]. Communications of the ACM , 1975, 18(6): 311-317.
- [5] Blinn, James F. Models of light reflection for computer synthesized pictures[J]. ACM SIGGRAPH Computer Graphics, 1977, 11(2):192-198.
- [6] Cook, Robert L., Kenneth E. A reflectance model for computer graphics[J]. ACM Transactions on Graphics (TOG), 1982, 1(1): 7-24.
- [7] Torrance, Sparrow. Theory for Off-Specular Reflection From Roughened Surfaces[J]. Journal of the Optical Society of America, 1967, 57(9): 1105-1112.
- [8] Petr Beckmann, Andre Spizzichino. The scattering of electromagnetic waves from rough surfaces[M]. Norwood:Artech House, 1987.
- [9] Walter Bruce, Stephen R. Marschner, Hongsong Li, Kenneth E. Torrance. Microfacet models for refraction through rough surfaces[C] //Proceedings of the 18th Eurographics conference on Rendering Techniques, 2007:195-206.
- [10] Schlick, Christophe. An Inexpensive BDRF Model for Physically based Rendering[J]. Computer Graphics Forum, 1994, 13(3): 149-162.
- [11] Smith B. Geometrical shadowing of a random rough surface[J]. Antennas and Propagation, IEEE Transactions on, 1967, 15:668-671.
- [12] Eugene d'Eon and David Luebke. Advanced Techniques for Realistic Real-Time Skin Rendering[M]. Hubert Nguyen. GPU Gems 3. Addison Wesley, 2007: 293-347.
- [13] Donner, Craig, Henrik Wann Jensen. Light Diffusion in Multi-Layered Translucent Materials[J]. ACM Transactions on Graphics (TOG), 2005, 24(3): 1032-1039.
- [14] Donner, Craig, Jensen H W. Light diffusion in multi-layered translucent

- materials[J]. ACM Transactions on Graphics (TOG), 2005, 24(3):1032-1039.
- [15] Csaba Kelemen, László Szirmay-Kalos. A Microfacet Based Coupled Specular-Matte BRDF Model with Importance Sampling[R]. Manchester, England: Eurographics, 2001.
- [16] Dachsbaecher Carsten, Marc Stamminger. Translucent shadow maps[C] // Proceedings of the 14th Eurographics symposium on Rendering, Leuven, 2003:197-201.
- [17] Jerry Tessendorf. Simulating Ocean Waves. SIGGRAPH '99 Course Notes & SIGGRAPH'2000: Course Notes 25: Simulating Nature: From Theory to Practice, 3.1-3.18,1999.
- [18] van der Laan, Vladimir J., Simon Green, Miguel Sainz. Screen space fluid rendering with curvature flow[C] //Proceedings of the 2009 symposium on Interactive 3D graphics and games. Boston, 2009: 91-98.
- [19] Lorensen, William E., Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm[J]. ACM Siggraph Computer Graphics. 1987, 24(4): 163-169.
- [20] Lauritzen Andrew. Deferred rendering for current and future rendering pipelines[C] // Proceedings of the ACM SIGGRAPH 2010 Course: Beyond Programmable Shading. Los Angeles, California, USA, 2010.
- [21] Markus Billeter, Ola Olsson, and Ulf Assarsson. Tiled Forward Shading[M] . Wolfgang Engel. GPU Pro 4: Advanced Rendering Techniques. A K Peters/CRC Press, 2013: 99- 112.
- [22] Takahiro Harada, Jay McKee, Jason C. Yang. Forward+: A Step Toward Film-Style Shading in Real Time[M]. Wolfgang Engel. GPU Pro 4: Advanced Rendering Techniques. A K Peters/CRC Press, 2013: 115- 135.
- [23] Rouslan Dimitrov. Cascaded Shadow Maps[R]. Santa Clara: Nvidia Corporation, 2007.
- [24] Fan Zhang, Hanqiu Sun, Oskari Nyman. Parallel-Split Shadow Maps on Programmable GPUs[M]. Nguyen H. GPU Gems 3. Boston: Addison-Wesley,

- 2007: 203-238.
- [25] Crow F C. Shadow algorithms for computer graphics[J]. ACM SIGGRAPH Computer Graphics, 1977, 11(2):242-248.
 - [26] Williams L. Casting curved shadows on curved surfaces[J]. ACM Siggraph Computer Graphics, 1978, 12(3):270-274.
 - [27] Cook R L, Reeves W T, Salesin D. Rendering antialiased shadows with depth maps[J]. ACM SIGGRAPH Computer Graphics, 1987, 21(4):283-291.
 - [28] Donnelly W, Lauritzen A. Variance shadow maps[C] //Proceedings of Symposium on Interactive 3D Graphics and Games, Redwood Shores, 2006: 161-165.
 - [29] Lauritzen A, McCool M. Layered variance shadow maps[C] //Proceedings of Graphics Interface, Windsor, 2008:139-146.
 - [30] Annen T, Mertens T, Seidel HP, et al. Exponential shadow maps [C] //Proceedings of graphics interface, Windsor, 2008:155-161.
 - [31] Vladimir Kajalin. Screen Space Ambient Occlusion[M]. Wolfgang Engel. ShaderX7: Advanced Rendering Techniques. Cengage Learning, 2009: 413- 436.
 - [32] McGuire Morgan, et al. The alchemy screen-space ambient obscurrence algorithm // Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. Vancouver, British Columbia, Canada, 2011: 25-32.
 - [33] McGuire Morgan, Michael Mara, David Luebke. Scalable ambient obscurrence[C]// Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics, Eurographics Association, 2012:97-103.
 - [34] Bavoil Louis, Miguel Sainz, Rouslan Dimitrov. Image-space horizon-based ambient occlusion[C] // Proceedings of the ACM SIGGRAPH 2008 talks. Los Angeles, California, USA, 2008:22.
 - [35] Zhukov Sergey, Andrei Iones, Grigorij Kronin. An ambient light illumination model[C] //Drettakis G, Max N eds. Rendering Techniques' 98. Springer Vienna, 1998:45-55.
 - [36] Crassin Cyril, et al. Interactive indirect illumination using voxel cone tracing[J]. Computer Graphics Forum, 2011, 30(7): 1921-1930.

- [37] Michael Schwarz. Practical Binary Surface and Solid Voxelization with Direct3D 11[M]. Wolfgang Engel. GPU Pro 3: Advanced Rendering Techniques. A K Peters/CRC Press, 2012: 337- 352.
- [38] Crassin Cyril, Simon Green. Octree-based sparse voxelization using the GPU hardware rasterizer[M]. Patrick Cozzi, Christophe Riccio. OpenGL Insights. A K Peters/CRC Press, 2012: 31-40.
- [39] Mara M, McGuire M, Nowrouzezahrai D, et al. Fast global illumination approximations on deep G-buffers[R]. Tech. Rep. NVR-2014-001, NVIDIA Corporation, June. URL: <http://graphics.cs.williams.edu/papers/DeepGBuffer14>.

致谢

时光匆匆如流水，转眼两年半的硕士生涯即将结束。在这两年半中，我度过了快乐而又充满挑战的硕士研究生涯，学到了宝贵的专业知识，也结实了很多志同道合的朋友。值此毕业论文完成之际，我谨向所有关心和帮助过我的老师、亲友呈上我最诚挚的感谢与美好的祝愿。

首先，衷心感谢我的导师周昆教授、任重副教授、侯启明副教授对我的细心指导。他们深厚的学术造诣、严谨的治学精神和孜孜不倦追求高标准科研成果的精神深深地感染了我。他们在我整个硕士生涯中对我的指导和帮助才使得我在求学的道路上满载而归。我还要感谢杭州师范大学许威威教授对我的科研指导，让我学到了很多做学术研究的方法。

我还要感谢我的亲人，是你们鼓励帮助让我一次次地战胜困难，没有你们的支持，我不可能在学习的道路上走的如此坚定。另外同样要感谢 GAPS 实验室的兄弟姐妹们，是你们让我两年半的硕士生活充满乐趣。

最后，感谢评阅论文和答辩委员会的各位老师，感谢你们百忙之中抽出时间给予我指导。

阮蒙铠

2015 年 1 月

作者简历

阮濛铠，男，生于 1990 年 5 月，浙江省宁波市人。2012 年 6 月毕业于华中科技大学软件工程专业，获学士学位。2012 年至今在浙江大学攻读硕士学位，现于浙江大学 CAD&CG 国家重点实验室从事科研工作，研究方向为计算机图形学。