

# 毕业论文（设计）正文

题目：基于 GPU 的并行光线追踪渲染器设计与实现

## 摘要

图形渲染中的光线追踪技术能够真实地模拟物理光照，呈现一个真假难辨的世界，但也伴随着极大的运算开销。随着硬件水平的提高和先进算法的提出，此前仅能用于离线渲染的光线追踪如今也在迈向实时。本文首先调研了现有的各种光线追踪优化方法以及主流渲染器，总结其中的优秀算法并加以优化，形成了自己的解决方案。然后，本文基于 OpenGL 和 GLSL 编程语言实现了一个 GPU 上并行的光线追踪渲染系统，并设计了一种高效的 CPU 端与 GPU 端的数据传输方案。通过这些算法组件与 GPU 的配合，该系统在几个示例场景下都实现了高质量的实时渲染能力。最后，该系统也提供了基础的交互功能，支持艺术家和 3D 创作者利用该系统搭建 3D 场景并实时渲染。在未来，本系统也有望作为一种新的通用的 GPU 渲染方案接入完整的图形引擎。

**关键词：**计算机图形学，实时渲染，光线追踪，GPU，并行计算

## Abstract

In graphics rendering, ray tracing technique can accurately simulate physical light, presenting a world that blurs the line between reality and fiction, but it also comes with huge computing overhead. With advancements in hardware capabilities and the introduction of advanced algorithms, ray tracing, which previously limited to offline rendering, is now making strides towards real-time applications. This paper first surveys various existing optimization methods for ray tracing and mainstream renderers, summarizes outstanding algorithms, and optimizes them to form its own solution. Then, leveraging the OpenGL and GLSL programming languages, this paper implements a parallel ray tracing rendering system on the GPU and designs an efficient data transfer scheme between the CPU and GPU. Through the coordination of these algorithm components with the GPU, the system achieves high-quality real-time rendering capability in several example scenes. Finally, the system also provides basic interactive features, supporting artists and 3D creators in building 3D scenes and rendering them in real-time. In the future, this system is also expected to serve as a novel, general-purpose GPU rendering solution integrated into a comprehensive graphics engine.

**Key Words:** Computer graphics, Real-time rendering, Ray tracing, GPU, Parallel computing

## 图目录

图 1.1 光栅化渲染管线 . . . . .	3
图 1.2 光栅化和光线追踪方法的对比 . . . . .	4
图 2.1 BVH 划分示意图 . . . . .	12
图 2.2 DynamicBVH 结构示意图 . . . . .	13
图 2.3 逆变换采样示意图 . . . . .	15
图 2.4 NEE + MIS 采样方案示意图 . . . . .	17
图 2.5 各种采样策略的比较 . . . . .	17
图 2.6 A-Trous Wavelet 滤波器 . . . . .	20
图 2.7 高斯滤波与联合双边滤波的对比 . . . . .	20
图 2.8 SVGF 重建器 . . . . .	21
图 3.1 光线追踪部分存储结构示意图 . . . . .	23
图 3.2 RenderPass 组合而成的完整渲染管线 . . . . .	25
图 3.3 一个分层的场景示意图 . . . . .	31
图 3.4 用户界面 . . . . .	32
图 4.1 场景编辑示例 . . . . .	34
图 4.2 几何信息和中间结果可视化 . . . . .	35
图 4.3 大尺度户外场景渲染 . . . . .	35
图 4.4 室内场景渲染 . . . . .	36
图 4.5 水下场景渲染 . . . . .	37
图 4.6 玻璃瓶渲染 . . . . .	37

## 表目录

表 3.1 系统中实现的所有 RenderPass . . . . .	24
-------------------------------------	----

## 目 录

摘要.....	I
Abstract.....	II
图目录.....	III
表目录.....	IV
第 1 章 绪论.....	2
1.1 课题背景.....	2
1.1.1 图形渲染技术简介.....	2
1.1.2 传统图形渲染管线.....	2
1.1.3 光线追踪与现代渲染技术的发展.....	3
1.2 相关工作.....	5
1.2.1 光线追踪优化算法.....	5
1.2.2 渲染方案调研.....	6
1.3 课题动机以及意义.....	7
1.4 专有名词解释.....	7
1.5 本章小结以及本文的主要内容.....	9
第 2 章 实时路径追踪算法设计与优化.....	10
2.1 路径追踪.....	10
2.1.1 渲染方程.....	10
2.1.2 蒙特卡洛路径追踪.....	10
2.2 路径追踪加速.....	11
2.2.1 BVH .....	11
2.2.2 DynamicBVH .....	13
2.2.3 GPU 并行追踪 .....	14

2.3 路径追踪降噪.....	15
2.3.1 高质量采样.....	15
2.3.2 时域样本复用.....	17
2.3.3 图像滤波.....	19
2.3.4 SVGF .....	19
2.4 本章小结.....	21
第 3 章 并行光线追踪渲染系统设计与实现.....	22
3.1 系统概述.....	22
3.1.1 开发环境.....	22
3.1.2 需求与挑战.....	22
3.2 GPU 上的数据组织与传输 .....	22
3.2.1 OpenGL 提供的数据接口.....	22
3.2.2 场景数据组织.....	23
3.2.3 RenderPass .....	24
3.3 GPU 端算法实现 .....	26
3.3.1 路径追踪实现.....	26
3.3.2 DynamicBVH 求交实现 .....	26
3.4 材质系统.....	27
3.5 场景系统.....	31
3.6 交互系统.....	31
3.7 本章小结.....	33
第 4 章 系统结果展示与分析.....	34
4.1 场景编辑示例.....	34
4.2 可视化.....	34

4.3 实时渲染效果展示.....	34
4.3.1 大尺度户外场景.....	35
4.3.2 室内场景.....	36
4.3.3 水下场景.....	36
4.3.4 复杂散射场景.....	36
4.4 本章小结.....	37
第 5 章 总结与展望.....	38
5.1 论文总结.....	38
5.2 不足之处.....	38
5.3 未来展望.....	38
参考文献.....	39
致谢.....	40

## 第 1 章 绪论

### 1.1 课题背景

#### 1.1.1 图形渲染技术简介

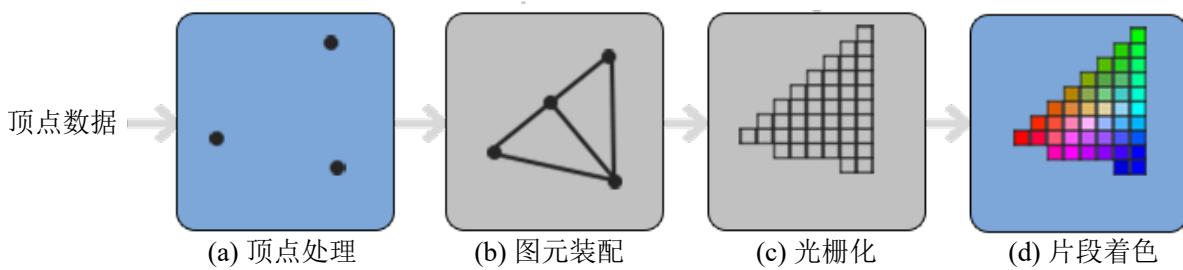
图形渲染技术特指利用计算机中存储的三维模型数据、虚拟的光源、观察者等信息，通过复杂的计算真实地绘制出观察者所能看到的二维图像，实现在计算机中游览虚拟场景。在当今，图形渲染技术已经成为了日常生活和生产中不可或缺的一部分，广泛应用于游戏、影视动画、特效、三维工业软件等领域。

图形渲染大致可以按使用情景分为两种门类：离线渲染与实时渲染。离线渲染指设备在后台使用大量时间（从几分钟到几小时不等）绘制一帧画面，最终将许多张绘制好的图像连成视频，通常在电影中使用；而实时渲染则要求渲染必须实时进行，用户能够与系统交互并实时得到反馈，这就对渲染的速度提出了很高的要求，通常来说每秒至少 30 帧画面的渲染速度，才能让用户有一个较为舒适的交互体验。而随着人们生活水平的提升，硬件性能的提高，人们对实时渲染的帧率和画面要求也越来越高。

以往，游戏的画面质量往往是远不如电影的，因为离线渲染可以使用大量的算力资源和时间，来进行更加复杂的计算（例如光线追踪），而实时渲染受限于速率要求，不得不对算法做出大量简化。近年来，实时渲染的画面质量也在变得越来越好，光线追踪等以往只能用于离线的算法也在以各种方式达成实时标准。人们常用“电影级画面”来高度评价一款游戏，是因为它的实时渲染效果已经能够逼近离线渲染，这得益于更加优秀的算法的提出、硬件水平的提升、并行计算理念的兴起和 GPU 编程的发展。

#### 1.1.2 传统图形渲染管线

De Vries 在他的网络教程 Learn OpenGL<sup>[1]</sup> 上介绍了传统的光栅化（Rasterization）渲染管线，图1.1展示了该渲染管线的核心步骤：a) 输入 3D 模型顶点数据，通过一系列线性变换将顶点从局部坐标系变换到以摄像机为中心的观察坐标系下，再通过一个投影矩阵将视锥范围内的顶点映射到一个  $[-1, 1]^3$  的立方体空间（又称裁剪空间）内，该范

图 1.1 光栅化渲染管线，图改编自 Learn OpenGL<sup>[1]</sup>

围外的顶点将被剔除；b) 将剩下的顶点数个一组装配为图元（例如：三角形）；c) 再逐个图元进行光栅化，在屏幕空间将图元拆分为一个个像素（片段）；d) 在每个片段上进行光照计算，得到该像素的颜色值。在一帧的渲染中，场景中的模型逐个进行光栅化，在每个像素中通过深度测试保留距离摄像机最近的片段并最终呈现。

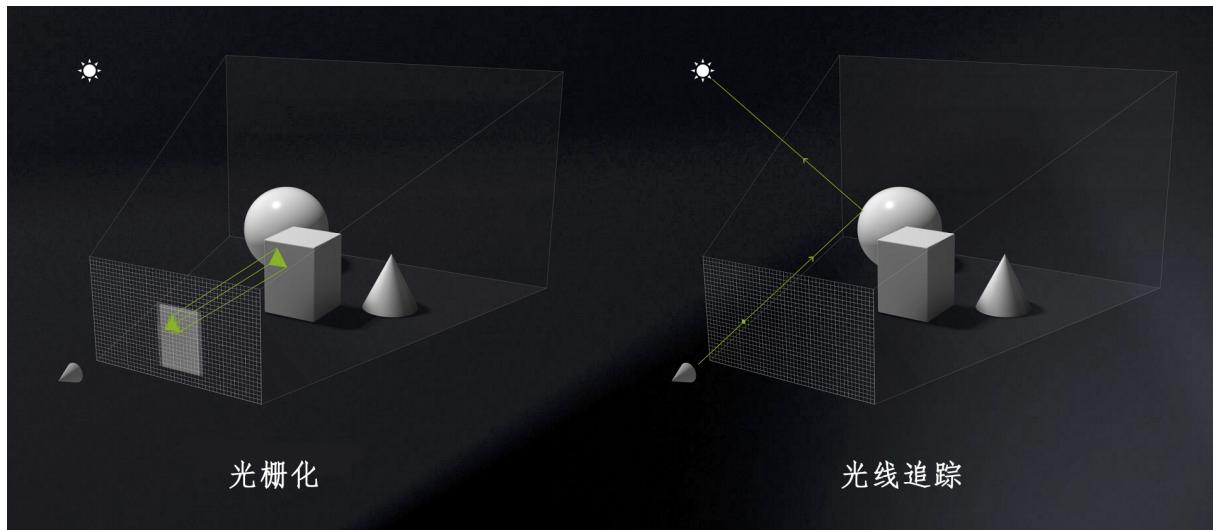
传统的光栅化流程基于局部光照原理，在着色时只有图元本身、摄像机以及光源的信息，只能计算直接光照，无法计算经过多次反射、散射形成的复杂光照效果以及阴影。虽然人们也提出了许多方法来为光栅化添加近似的全局光照效果，例如阴影贴图、屏幕空间反射、环境光遮蔽等等，但这些方法往往不够完善，在许多情景下存在缺陷，并且这些方法的加入会让渲染管线变得更加臃肿，不便于使用者理解和控制。

光栅化方法最大的优势就是效率，在成熟的渲染管线与 RHI (Rendering Hardware Interface) 的加持下，光栅化方法能够在几毫秒内渲染一帧，轻易地满足实时要求。另外，光栅化渲染技术已经有了很久的发展历史，有着大量基于光栅化的全局光照近似方法，前人们也在这项任务上积累了很多经验和宝贵的遗产。因此时至今日，光栅化方法仍然是游戏和各种实时应用中的主流。

随着硬件水平的提升和基于物理渲染（Physically Based Rendering, PBR）观念的兴起，人们不再满足于光栅化渲染的效果，开始追求更加逼真的画面，力求创造一个真假难辨的虚拟世界。

### 1.1.3 光线追踪与现代渲染技术的发展

光线追踪（Ray tracing）基于全局光照模型，它的核心思路就是从相机出发发射光线，计算光线向量与场景中物体的交点；再根据击中物体表面的材质，通过反射或散

图 1.2 光栅化和光线追踪方法的对比，图源网络文章<sup>[2]</sup>

射，递归地生成下一条光线，直到击中发光光源。根据光路的可逆性，可以根据沿途材质，计算这一条光路传递的能量，从而计算一个像素的颜色。光线追踪能够自然地呈现阴影、反射、折射等效果，考虑到了场景中所有物体之间的相互作用，因此有着非常高质量的渲染效果。需要注意的是：光线追踪本身仅仅是一个统称，具体实现方式有很多。最早的 Whitted style 光线追踪<sup>[3]</sup> 虽然也能呈现阴影、反射、折射等效果，但并不是基于物理的；路径追踪（Path tracing）<sup>[4]</sup> 是一种物理上无偏的光线追踪方法，本文将在2.1节详细介绍。

光线追踪原本是仅应用于离线渲染的算法，因为其计算量巨大，每条光线都需要计算它与场景中所有图元的交点，并经过若干次反射或散射后才能得到一条光路，用以计算一个像素的颜色。相比之下，光栅化方法中每帧每个顶点只需要进行一次线性变化，每个图元只需要进行一次光栅化，并且仅考虑直接光照的着色计算也更加简单。图1.2展示了这两种算法的区别。

除了算法本身的复杂性之外，硬件支持不足也是制约光线追踪的一大因素。图形处理单元（GPU）早期是为光栅化管线而设计的，它支持并行地对顶点进行线性变化、光栅化以及片段着色等操作。而尽管光线追踪本身可并行化，但由于没有成熟的硬件支持，早期在离线渲染中，人们还是在使用 CPU 串行或多线程地进行光线追踪。据说当年《阿凡达》用了 40000 颗 CPU，104TB 内存，10G 网络带宽，整整渲染了 1 个多月；

2016 年上映的动画电影《小门神》声称所有设备的总渲染时长达 8000 万小时。

2006 年，NVIDIA 推出通用并行计算架构 Cuda，使用户可以用 C 语言为 GPU 编程，大大降低了 GPU 开发的难度，2009 年，NVIDIA 推出基于 GPU 的光线追踪渲染器 Optix，并在后续稳定地随着 GPU 的升级而加速。2018 年起，在 NVIDIA、AMD 等各大厂商与众多研究者的大力推动下，以 NVIDIA GeForce RTX 20 系显卡为代表的一系列支持硬件光线追踪的 GPU 出现了。这代显卡具备两个核心：1. 专用光线追踪硬件 RT-core，它集成到了流式多处理器（SM）中，支持 BVH 的遍历以及光线与三角形的求交测试；2. 用于高性能 AI 处理的 Tensor-core，它加速深度学习超采样（DLSS）技术，使用较低的渲染分辨率得到高质量的大分辨率渲染结果，将视觉保真度和帧率提升到了一个新的层次。

目前的光线追踪还没有成为一项通用技术，一方面，它需要在专用硬件上运行，并且效率仍然远不如传统光栅化，在实时应用中使用中必然会压缩其他数据处理的时间；另一方面，实时光线追踪仍然需要大量的近似策略和降噪手段，它们会一定程度上牺牲画面质量。需要注意的是，目前应用于游戏中的光线追踪大多是光线追踪和光栅化的混合管线，它们利用光线追踪改进了一些问题，但成效距离真正的光线追踪仍有一段距离。因此，实时光线追踪优化相关的理论研究和工程化落地尚未完成，仍然需要继续推进。当实时光线追踪可以在每台设备上轻易运行且不成为性能瓶颈时，它才能成为图形渲染的标配。

## 1.2 相关工作

### 1.2.1 光线追踪优化算法

光线追踪的优化主要有两个出发点：加速和降噪。

加速方面，一种思路是使用空间数据结构管理图元以加速求交，包括八叉树、Kd-tree、BVH 等。其中八叉树、Kd-tree 都是基于空间划分的方法，图元被划分到多个空间时会产生冗余计算，BVH 基于物体划分，结构简单且没有图元冗余（但有空间冗余），通常用于硬件加速，闫润等人<sup>[5]</sup>综述了硬件加速 BVH 的相关算法。另一种思路是

基于 LOD (Level of detail)，为模型生成简化后的低精度版本，渲染时依据观察距离等因素动态加载。UE5 的 Nanite 系统<sup>[6]</sup> 是一套非常全面的 LOD 系统，基于三角形簇划分实现了平滑的层级切换。

降噪方面，一种思路是提高采样质量，Tokdar 等人<sup>[7]</sup> 综述了重要性采样相关理论，本文也将在2.3.1节详细描述。近年来兴起的 ReSTIR 技术<sup>[8]</sup> 基于重采样<sup>[9]</sup> 和蓄水池采样<sup>[10]</sup>，提高了样本的有效性。路径引导（Path guiding）<sup>[11]</sup> 使用空间数据结构存储先前计算的辐射值，实现对整个辐射场的近似采样，近年来，深度学习方法应用于路径引导<sup>[12]</sup>，实现了辐射场的连续存储，也取得了良好的效果。

另一种思路是时空样本复用，时域样本复用策略（如 TAA）利用重投影<sup>[13]</sup> 找到像素在上一帧的对应位置，实现历史样本复用。空间样本复用（或者说图像空间滤波）方面，联合双边滤波<sup>[14]</sup> 在高斯滤波核的基础上，利用额外的几何信息对滤波加以指导，适用于渲染任务。Schied 等人提出的 SVGF<sup>[15]</sup> 是一种广为使用的时空综合滤波器，除了上述方法之外，还估计了渲染的方差用于衡量噪声。本文在2.3节详细描述了这些降噪方法。除了这些方法之外，神经网络也可以直接用于图像滤波。NVIDIA 使用专用硬件 Tensor-core 加速深度学习超采样（DLSS），展现了强大的性能。

### 1.2.2 渲染方案调研

硬件光线追踪方案。NVIDIA 在 2009 年推出的 OptiX 是首个利用 GPU 加速的光线追踪渲染器，但在当时还不能做到实时，在 2018 年，随着 NVIDIA 推出可加速硬件光线追踪的新架构 Turing，各种图形 API（如 DirectX、Vulkan）都相继支持了硬件光追，游戏引擎 UE、Unity 也相继提供了硬件光线追踪特性。NVIDIA 自家的 OptiX 渲染器也支持了硬件加速，达到了实时光线追踪的标准。硬件光追方案的一个普遍特点是由硬件完成 BVH 遍历和光线求交运算，仅将求交后的回调函数暴露给用户完成。这种方案性能强大，但也有其自身的局限性，需要专用的设备才能运行，并且底层算法固定，不便于更新。

混合渲染方案。UE5 的最新的一代的动态全局照明方案 Lumen 性能非常强大，且

没有使用专用硬件加速。它是一套混合渲染管线（Hybrid Rendering Pipeline），本质是光栅化方法，但在屏幕空间使用基于光线步进（Ray marching）和有向距离场（SDF）的光线追踪补充了细节。除了 Lumen 之外，Unity 也有自己的混合渲染管线，它们的特点是以大量的启发式逻辑来兼顾光栅化的性能和光线追踪的质量，能以较低的开销得到较好的效果，但它们的成效和真正的光线追踪还有差别。

### 1.3 课题动机以及意义

传统的光线追踪算法在 CPU 上运行速度较慢，不适用于实时应用；现有的实时光线追踪技术大多依赖专用硬件加速，不具备普适性和可拓展性。本课题旨在利用通用 GPU 的并行能力，重点关注光线追踪在算法层面的优化，总结集成现有算法和自己的方案，实现一个无需专用硬件支持的并行光线追踪渲染器。该课题的目标包括：

- 通用性：该系统应当能够在支持 OpenGL 的设备上运行，不需要专用硬件 RT-core。
- 实时性：该系统在一般场景中应当至少达到每秒 30 帧，并尽可能的提高帧率。
- 高质量：该系统需要保证高质量的实时渲染，不应为了加速过多地牺牲质量。

本课题作为一种新的通用 GPU 上的实时光线追踪渲染方案，能够为艺术家和 3D 创作者提供支持，也有望在未来接入完整的图形引擎。

### 1.4 专有名词解释

本节将介绍图形渲染领域的若干常用术语，以便于读者理解和本文叙述。

RHI：Rendering Hardware Interface，渲染硬件接口，是一个用于跨平台游戏和图形应用程序开发的概念，它提供了一个抽象的接口，使开发人员可以轻松地访问不同硬件架构上的图形渲染功能，而无需直接处理底层硬件细节。常见的 RHI 包括 OpenGL、Vulkan、DirectX 等等。

渲染管线：即渲染流水线，可能包括着色阶段、后处理阶段等等，每个阶段使用前面阶段的输入，并产生一定输出。在图形学中，通常使用管线这一叫法。

Shader：着色器或着色器程序，即 GPU 上运行的脚本，在 OpenGL 中，Shader 使用 GLSL 语言编写，该语言风格和 C 类似，但不支持指针、递归。

立体角：立体角（Solid angle）即三维球面上的角度，定义为球面投影面积与半径的平方之比。球面上的一个朝向可以用两个角度  $(\theta, \phi)$  表示，该朝向的微分立体角为  $d\omega = \sin\theta d\theta d\phi$ 。立体角通常作为球面积分的单位。

辐射度量学：为了对光照进行度量，基于物理的渲染中使用辐射度量学中的概念，主要包括四个量：

- 辐射通量（Radiant Power）：单位时间辐射的能量，即辐射的功率， $\Phi = \frac{dQ}{dt}$ 。
- 辐射强度（Radiant Intensity）：单位立体角（solid angle）上的辐射通量， $I = \frac{d\Phi}{d\omega}$ 。  
Intensity 与距离无关，通常用于衡量点光源的光强。
- 辐照度（Irradiance）：单位面积接收到的辐射通量， $E = \frac{d\phi}{dA}$ ，对于点光源来说，距离越远，接收方接收的 Irradiance 越小。
- 辐射（Radiance）：单位面积在单位立体角上的辐射通量， $L = \frac{d^2\Phi}{dAd\omega\cos\theta}$ ，此处  $\theta$  为立体角相对投影面的角度。Radiance 的重要性质在于它在光线传播中保持不变，因此可以用来作为光线的度量。

Transform：变换，三维世界下的变换（包括旋转、平移、缩放）可以用一个  $4 \times 4$  的矩阵表示，变换矩阵的连乘可以表达连续多次变换的复合。在常用渲染引擎中，Transform 通常用于表示一个实例相对于父坐标系经历的所有变换，这也是一个顶点向量从模型的局部坐标系变换到父坐标系要乘上的矩阵。

GBuffer：几何缓冲区，它是一个存储了渲染管线中所需数据的缓冲区，通常是一个屏幕大小（屏幕宽  $\times$  屏幕高）。渲染管线的一些阶段可以将原始的几何信息（法线、深度等）存储在 GBuffer 中。然后，后续的渲染阶段就可以直接从中读取所需的数据。

HDR：高动态范围（High Dynamic Range, HDR）即使用更高的位深度存储图像，可以表达超过 1 的颜色值，便于进行物理空间的辐射度计算。相对的，由于普通的 RGB 格式图片和显示器输出（低动态范围，LDR）能够表达的亮度范围受限，在 HDR 范围下计算的像素颜色需要经过色调映射（ToneMapping）和伽马矫正再输出。常用的色调

映射函数有 ACES。

贴图/纹理：纹理是将图片贴在三维模型上，以表达各处不同（Spatially varying）细节的技术，它利用 3D 模型中的 uv 信息实现。uv 标识了模型的每个顶点在纹理贴图上对应的 2D 坐标 ( $u, v$ )，图元中的片段 uv 可以由顶点 uv 插值得到。纹理不仅能直接改变颜色，也可以用于修改其他参数，例如法线、粗糙度等。通常，各类纹理也被称为贴图（Map），如法线贴图（Normal map）等。

环境贴图：用于充当画面中的背景，通常是一个立方体贴图或映射到球面的平面贴图。渲染时，若光线没有击中任何物体（或光栅化中一个像素不包含任何片段），就在环境贴图的对相应方向上采样，作为该像素的颜色。环境贴图常常包含太阳等极高亮度的背景，通常是 HDR 范围的，环境贴图不仅仅可以作为背景，也可以作为光源。

## 1.5 本章小结以及本文的主要内容

本章首先介绍了图形渲染的课题背景，包括实时渲染和离线渲染的概念、传统渲染管线和现代光线追踪技术的发展；然后调研了光线追踪的优化算法以及相关渲染方案，最后明确了本课题的动机和意义，并给出了本文用到的一些专用名词的解释。

在第二章，本文将详细展开实时路径追踪算法的实现以及优化。在第三章，本文将阐述如何具体实施一个 GPU 上的并行光线追踪系统。在第四章中，本文展示了该系统的效果。最后在第五章总结与展望，指出该系统的局限性与进一步优化空间。

## 第 2 章 实时路径追踪算法设计与优化

### 2.1 路径追踪

#### 2.1.1 渲染方程

1986 年，Kajiya 提出渲染方程（The Rendering Equation）<sup>[4]</sup>，他基于辐射度量学，从物理上精确地描述了光能在场景中的流动，该方程是一个积分的形式，准表达了平面上一点向某方向发出的辐射与球面其他方向的入射辐射的关系：

$$L_o(x, \omega_0) = L_e(x, \omega_0) + \int_{\Omega^+} L_i(x, \omega_i) f_s(x, \omega_i, \omega_0) \cos(\omega_i) d\omega_i \quad (2.1)$$

式中， $\omega$  为立体角（Solid angle），描述三维球面的一个方向； $L_i, L_o$  分别表示入射和出射辐射， $L_e$  为物体表面自发光； $f_s$  为双向散射分布函数（Bidirectional scattering distribution function, BSDF），描述  $\omega_i$  方向的入射光，经过表面一点  $x$  后散射向  $\omega_0$  方向的出射光的比例，它可以表达物体的材质属性，本文将在材质系统（3.4节）中详细介绍。式中涉及了一些辐射度量学中的概念，在1.4节有更详细的解释。

渲染方程在几何光学意义下是客观正确的，它奠定了现代真实感渲染方法的理论基础，理论上只要求解了这个积分，就能得到真实的渲染结果。后续人们的主要研究任务就是如何在计算机中更快、更好地求解这个积分。

#### 2.1.2 蒙特卡洛路径追踪

蒙特卡洛方法通过随机的采样来估计一个积分，具体而言，设要求解积分是  $\int_a^b f(x) dx$ ，可以按照一个任意的概率密度  $p(x), x \in [a, b]$  中抽取  $N$  个样本，用这些样本估计该积分值为：

$$I = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)} \quad (2.2)$$

这一估计方法是无偏的，随着样本数的增加，蒙特卡洛估计的结果将会逐渐收敛于正确的理论解。

路径追踪（Path tracing）<sup>[4]</sup> 是一种广为使用的光线传输算法，它在球面空间（立体

角）上采样光线的下一次弹射方向，使用蒙特卡洛方法来估计渲染方程的解，因而路径追踪也是无偏的。将蒙特卡洛方法应用在渲染方程上可以写作：

$$L_o(x, \omega_0) = L_e(x, \omega_0) + \frac{1}{N} \sum_{i=1}^N \frac{L_i(x, \omega_i) f_s(x, \omega_i, \omega_0) \cos(\omega_i)}{p(\omega_i)} \quad (2.3)$$

式中最困难的部分是  $L_i(x, \omega_i)$ ，它需要递归求解。具体而言，估计着色点  $x$  对观察方向  $\omega_0$  的辐射  $L_o(x, \omega_0)$  时，首先按照一个概率分布（可以是均匀分布或其他任意分布）在球面上随机选取下一条光线方向  $\omega_i$ ，然后再朝该方向发射射线，假设击中点  $x'$ ，则有  $L_i(x, \omega_i) = L_o(x', -\omega_i)$ ，于是可以递归求解，直到射线击中光源，再将来自光源的辐射回溯，逐层计算渲染方程的其他部分（BSDF 等），最终得到相机的接受到的辐射的估计值。

不难注意到，由于  $L_i$  的递归性质，蒙特卡洛估计只能使用一个样本 ( $N = 1$ )，否则会出现指数爆炸的问题。过少的样本数会使得估计的方差过大，在最终渲染结果中产生难以接受的噪声（尽管如此，这依然是无偏的）。一种朴素的想法是每个像素进行多次采样（Samples per pixel，下文称 SPP 数），各自进行路径追踪后取均值的方式来降低方差，但这样又会导致更高的开销。

在离线渲染中，通常使用数千到数万 SPP 才能得到一个较为干净的渲染结果，这在实时应用中是不现实的。另外，朴素的光线求交算法需要计算光线与场景中所有图元的交点，每帧的计算复杂度是  $O(SPP \times \text{图像宽} \times \text{图像高} \times \text{场景图元数} \times \text{光线反射次数})$ ，要达到实时必须在 34ms 内完成，是几乎不可能的。本文将在 2.2 节介绍路径追踪的加速算法，在 2.3 节介绍路径追踪的降噪算法，这两者往往是相辅相成的，降噪意味着可以使用更少的 SPP 以提升帧率，加速意味着可以使用更多的 SPP 来减小噪声。

## 2.2 路径追踪加速

### 2.2.1 BVH

层次包围盒（Bounding Volume Hierarchies, BVH）是一种空间数据结构，它使用二叉树结构对场景中的图元进行划分，从而加速光线与图元的求交运算。具体而言，BVH

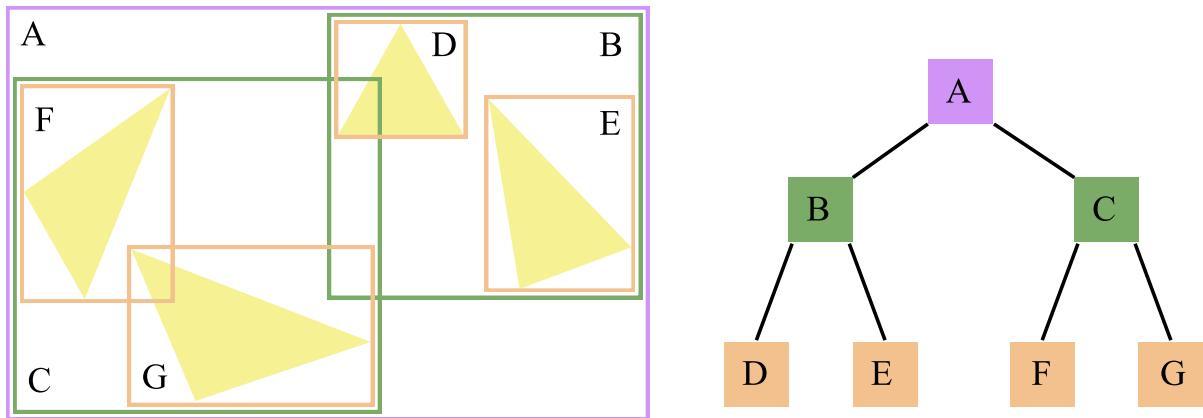


图 2.1 BVH 划分示意图

树的每个节点都记录了一个轴对齐包围盒 (Axis-aligned Bounding Box, AABB)，根节点的包围盒最大，包含了场景中的所有图元，划分时，首先将所有图元按照某一轴上的位置排序，然后选取一个中间点将图元划分为两份，形成两个较小的包围盒并由两个子节点控制，子节点会递归进行划分，直到一个节点包含的图元数量少于设定的下限。图2.1展示了 BVH 树的构建过程，根节点 A 包含了四个三角形，它按照横轴划分为 B 与 C 两片区域，各自包含两个三角形，B 与 C 又进一步划分为 DEFG，至此每个叶子节点仅包含一个三角形，构建结束。注意：为了直观，图中将外侧包围盒略微扩大了，实际上许多包围盒边界（例如 A,C,F 的左边界）是重叠的。

光线与 BVH 求交时，从根节点出发，首先令光线与包围盒求交，若光线与包围盒没有交点，则一定与其中的所有图元都没有交点，可以全部跳过；若有交点，再递归地进入其两棵子树中求交，直到抵达 BVH 的叶子节点，再与叶子节点下的所有图元求交。BVH 使得场景求交的复杂度降低到  $\log$  级别。

图2.1中可以看到，由于图元有一定面积，划分后的两侧包围盒可能有所重叠（如图中 B 与 C），但每个图元仅属于一个空间（即便图元恰好被另一侧的空间覆盖，也无需考虑）。相比其他基于空间划分的数据结构（八叉树、Kd-tree 等），BVH 基于物体的划分策略消除了图元求交的冗余计算，但会导致一定程度上的空间的冗余。SAH (Surface Area Heuristic) 是一种启发式策略，它同时考虑 XYZ 三个轴向，选取使两侧包围盒面积之和最小的划分点，降低了空间冗余。本文选择使用 SAH 优化的 BVH 加速求交，在下

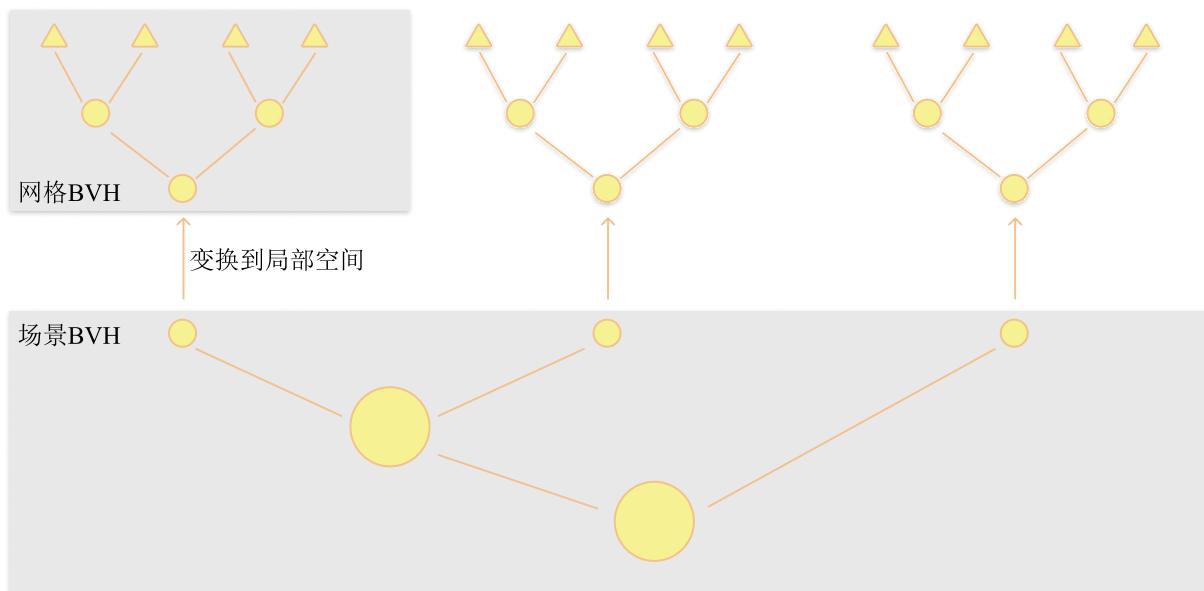


图 2.2 DynamicBVH 结构示意图

一节中，本文将进一步设计一种动态的 BVH 重构方案，以支持动态场景。

### 2.2.2 DynamicBVH

实时渲染的重要特性是“动态”，也就是说场景中的物体位置、相互关系每帧都可能会发生变化，而 BVH 算法本身并没有给出一个动态维护的方案。朴素的想法是：每帧都根据当前场景的图元位置重新构建一次 BVH。尽管这个构建过程与像素数、采样数无关，但它难以并行，只能在 CPU 中完成，是独立于光线追踪开销的一个不小的开销。经测试，在一个约 34000 个三角形的场景下，每帧重构 BVH 就需要花费约 102ms。需知如今的高精度模型动辄数十万三角形，即便依赖 LOD 等算法降低精度，这种开销在每秒至少 30 帧的实时应用中还是不可接受的。

为此，本文设计了一种动态的 BVH 重构方案，核心思路是：场景中剧烈变化的通常只有各个模型之间的位置关系，而三维模型本身的几何结构（网格）通常是不变的。因此，该方法预先为每个几何模型单独创建一个 BVH（下文称为网格 BVH），在局部空间下管理它本身的图元，再使用一个总的 BVH（下文称为场景 BVH）来管理场景中的所有实例。该方法使用一个简单的包围盒包裹实例，将该包围盒视为一个图元，作为实例的替代参与求交运算，并由场景 BVH 进行划分。仅在实例的包围盒被光线击中

时，才进入相应的网格 BVH 进行图元的求交运算。DynamicBVH 的结构如图2.2所示。在3.3节，本文给出了 DynamicBVH 求交的具体实现。

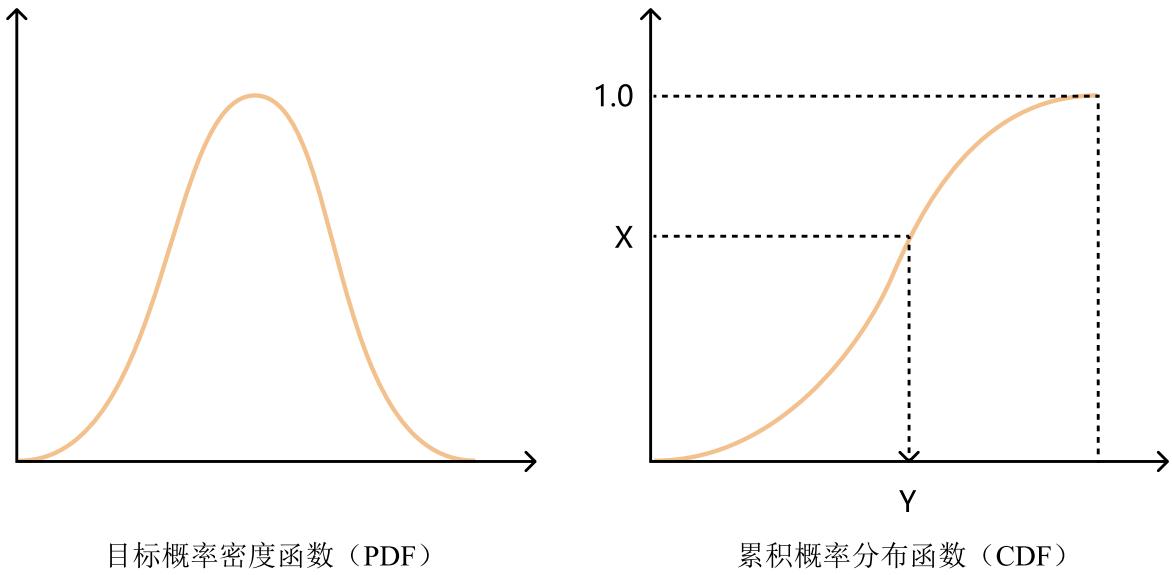
利用这样的结构，本文成功将两层 BVH 的重构操作隔离开来：在物体简单的移动、旋转和缩放时，仅重构场景 BVH，这一过程的复杂度与场景中的模型个数相关；仅在场景中有物体增删时，才去重构模型本身的 BVH。如此一来，频繁进行的动态更新只需要用开销较小的重构操作来维护，而开销大的重构操作并不会经常发生，大大降低了 BVH 重构需要的时间。

DynamicBVH 不仅仅包含上述算法，还有一套相应的数据存储和迁移机制，使得其每帧向 GPU 传输的数据规模也仅与模型数量有关，而非三角形个数。这一机制将在3.2.2节详细介绍。使用 DynamicBVH，本文在同样的 34000 个三角形的场景下测试，每帧的 BVH 重构加上数据传输仅需花费 4ms。

### 2.2.3 GPU 并行追踪

GPU 相比 CPU 拥有更多的核心数，但每个核心的控制能力较弱，擅长大规模的并行计算。应用程序给出的并行任务会被划分成一个个线程束（通常是 32 个线程一组）交由流式多处理器（SM）执行。SM 以单指令多数据（Single Instruction/Multiple Data, SIMD）方式执行，所有线程都执行相同的命令，但各自使用的数据不同。正是这种运行方式使得 GPU 的并行计算能力远远强于 CPU。

光线追踪需要向每个像素发射射线并追踪着色，各个像素的追踪是高度可并行化的，因此可以利用 GPU 加速计算。GPU 上的编程环境与传统 CPU 有所不同，一方面它通常不支持递归、指针和面向对象，算法和系统框架需要专门的设计；另一方面，GPU 算法的并行度非常重要，程序的分支结构会导致线程束分化，一侧分支需要等待另一侧完成才能执行，导致并行度的降低。除此之外，还需要进行数据在 CPU 端和 GPU 端之间的迁移工作，具体的方案将在第三章中详细介绍。



## 2.3 路径追踪降噪

### 2.3.1 高质量采样

尽管蒙特卡洛方法使用任意的采样概率分布都可以得到无偏的结果，但重要性采样理论<sup>[7]</sup>指出：概率分布的选取会极大程度上影响估计的方差，从而影响画面收敛于稳定的速度。蒙特卡洛估计可以视为对随机变量  $\frac{f(x_i)}{p(x_i)}$  的期望的估计（该随机变量的期望就是待求积分），因此，采样概率  $p(x)$  与被积函数  $f(x)$  形状越相似，该随机变量的方差就越小，估计也就越准确。

理想的采样概率分布应尽可能与被积函数匹配，而渲染任务的被积函数——渲染方程，是一个复杂形式，包含了来自下一条光线的能量（L 项）、双向散射分布函数（BSDF）和一个 cos 项。很难同时考虑它们的乘积，因此需要分开考虑。

双向散射分布函数和 cos 项一般有明确的解析形式，可以通过逆变换采样（Inverse transform sampling）方法采样。图2.3展示了二维逆变换采样的过程：首先计算出目标分布的累积概率分布函数（Cumulative Distribution Function, CDF），该函数的值域在 [0, 1] 之间，然后生成 [0, 1] 之间均匀的随机数  $X$ ，再通过 CDF 的逆函数映射到对应样本  $Y$ （如右图中箭头所示），就得到了符合目标概率密度的采样。对于 BSDF 和 cos 的乘积项，

可以先将其归一化，视为一个概率密度函数，然后在立体角空间下进行逆变换采样。

来自下一条光线的能量需要递归求解，无法在采样时就获知，但凭经验得知：来自直接光照的能量往往是最高的，因此可以考虑对直接光照进行采样。对于面积光源，本文均匀地在所有光面上采样；对于环境贴图带来的光照，本文预先对环境贴图的亮度进行积分，得到离散的累积概率分布，再使用逆变换方法 + 二分法求离散逆函数的值。需要注意，被积函数的积分单位是立体角，在光源上的采样需要变換回球面积分域上。

多重重要性采样（Multiple importance sampling, MIS）方法修改了蒙特卡洛的估计式，为各个采样策略分配权重以将它们结合，一般地，有  $M$  种采样策略，MIS 估计式为：

$$I_{MIS} = \sum_{i=1}^M \frac{1}{N_i} \sum_{j=1}^{N_i} w_i(x) \frac{f(x)}{p_i(x)} \quad (2.4)$$

其中， $w_i$  必须满足  $\sum_{i=0}^M w_i(x) = 1$ ，即在每个采样点，所有采样策略的权重和为 1。注意，在每个采样点可以使用不同的权重分配，这是 MIS 相较于朴素的加权平均的本质区别。本文选取一种启发式平衡的权重：

$$w_i(x) = \frac{p_i(x)}{\sum_{j=1}^M p_j(x)} \quad (2.5)$$

该权重能使各个采样策略更关注自身分布的波峰，在自身的波峰处，自身的权重更大，于是最终的采样结果能符合目标函数的多个波峰，最大化地降低噪声。

实际应用中，由于路径追踪只能追踪下一条光线（追踪多条会导致递归时的指数爆炸问题），无法同时统计所有采样策略的贡献，一种朴素的想法是等概率地选取各个采样策略，并将估计结果除以相应概率，这样也是无偏的。但基于下一事件估计（Next event estimation, NEE）对直接光照和间接光照的积分域进行划分，可以得到一种更加高效的方案：仅在直接光照区域使用启发式平衡权重，在间接光照区域，BSDF 采样的权重为 1，光源采样的权重为 0。采样时，同时对光源和 BSDF 进行采样，光源样本若被遮挡（需要一次射线检测，但不用递归），则直接丢弃，否则按照启发式平衡权重贡献；BSDF 采样样本若直接击中光源，就按照启发式平衡权重贡献，否则权重为 1 并继续追

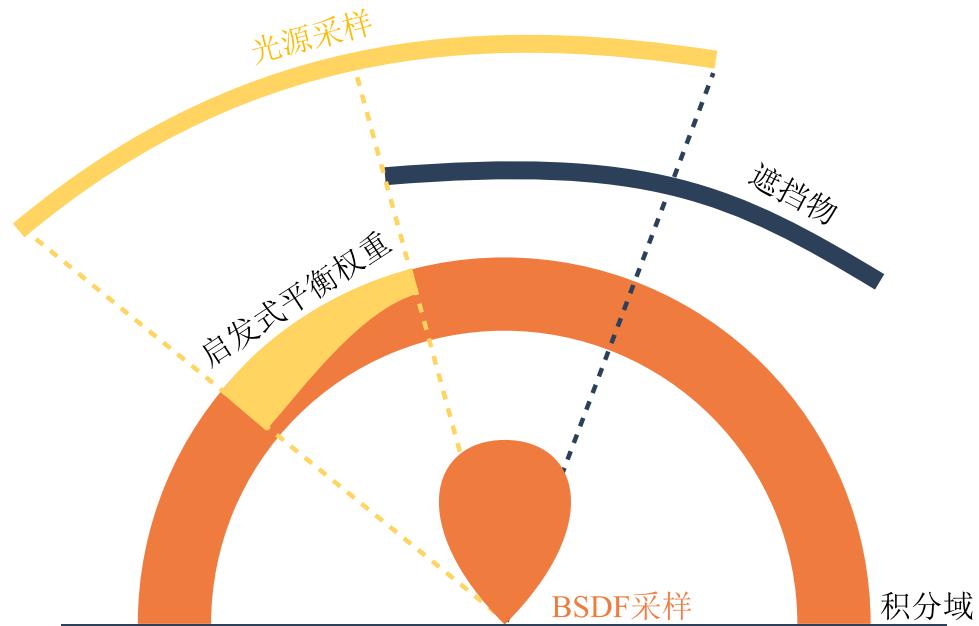


图 2.4 NEE + MIS 采样方案示意图

(a) 均匀采样      (b) 仅光源采样      (c) 仅 BSDF 采样      (d) 朴素 MIS      (e) NEE + MIS

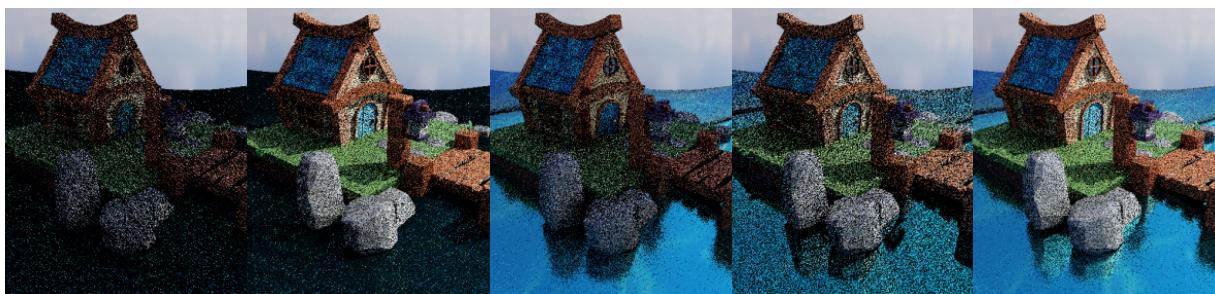


图 2.5 各种采样策略的比较

踪。这样做好处在于：1) MIS 权重考虑了光源的可见性，相比与朴素的做法，启发性更强；2) 可以仅用一次追踪，同时考虑 BSDF 和光源，无需按照概率划分。图2.4给出了 NEE+MIS 方案的示意图，用圆环表示了两种策略的 MIS 权重。图2.5在本系统中对比了本节提到的几种采样策略，均在 1spp 下渲染。可以看到 BSDF 采样有助于呈现光滑表面的反射效果（水面），而光源采样有助于阴影的呈现，MIS 将二者的优势成功结合了起来，NEE 又大大提升了样本的有效性。

### 2.3.2 时域样本复用

时域样本复用策略主要关注相邻帧之间的共同信息。TAA (Temporal Antialiasing) 是一个经典算法，在动态场景下，它利用当前帧和上一帧的相机投影矩阵，通过重投影

找到像素在上一帧画面中的对应位置（两帧之间像素的偏移量被称为 Motion vector，运动矢量），将当前帧与历史帧的颜色混合以提高有效样本数，实现抗锯齿。路径追踪也可以利用这种思想来复用历史样本以降低噪声。注意：在复用上一帧样本时，也隐含地复用了上一帧复用过的更早的历史样本。

但时域样本并不总是有效，随着场景中物体的运动、光照环境的变化，历史样本有可能已经失去意义，重投影得到的点也有可能不准确（例如，一个位置在当前帧中可见，但在上一帧中被遮挡）。错误地混合无效的历史样本会造成延迟、鬼影问题，本文使用下面几种策略来缓解这一问题：

- 几何一致性检测：利用缓存的 GBuffers（法线、深度、物体 ID 等等）判断两帧的样本的几何属性是否一致，不一致则直接丢弃。注意，很多不一致无法用几何信息判断出来，例如阴影的变化。
- 颜色截断：利用当前像素和周围像素的颜色，计算出颜色的均值和方差，若历史样本与均值相差过大（在数个标准差之外），则将其截断（clamp）到合理范围内。
- 指数加权平均：混合时，按照历史帧 80%，当前帧 20% 的比例进行混合，这样在混合过程中，历史样本的实际权重会随着时间的推移快速下降，产生“样本过期”的效果，错误混合的影响也能快速消失。

时域样本复用可以在 HDR 范围或是 LDR 范围下进行，在 LDR 范围下进行混合能更有效地抑制极高亮度的噪点，但会损失一些能量，导致画面偏暗；在 HDR 范围下进行则能更好地保持画面亮度，但对高亮噪点的抑制效果较差。一些综合性的降噪技术（如 SVGF<sup>[15]</sup>）通常会同时在 HDR 范围和 LDR 范围进行时域样本重用。

时域样本复用是一种非常有效的降噪技术，在大部分场景下都能极大地提升有效样本数，然而，它存在的延迟、鬼影问题目前还没有完善的解决方案，只能通过各种启发式的 Trick 来缓解。

### 2.3.3 图像滤波

在渲染结束后，可以再对得到的图像进行后处理以降低噪声。传统滤波方法使用高斯滤波核混合相邻像素，在去除噪声的同时也抹去了图像的高频信息，使得图像边缘变得模糊。渲染降噪和传统的图像去噪不同的是，渲染过程中产生了更多的信息，例如法线、位置、深度、反射率等等，这些信息是无噪声（noise-free）的，可以存储在 GBuffer 中。联合双边滤波（Joint bilateral filter）<sup>[14]</sup> 利用这些额外信息加以指导，控制高斯滤波核的权重，在降噪的同时保持画面原有的边界。

例如，可以在高斯滤波核的权重上额外乘上一个法线权重：

$$w_n = \max(0, n(p) \cdot n(q))^{\sigma_n} \quad (2.6)$$

其中  $p, q$  是要混合的两个像素， $\sigma_n$  是一个额外的控制参数，可以调节法线权重的重要性。除了法线之外，本系统还使用了深度和颜色信息作为指导。

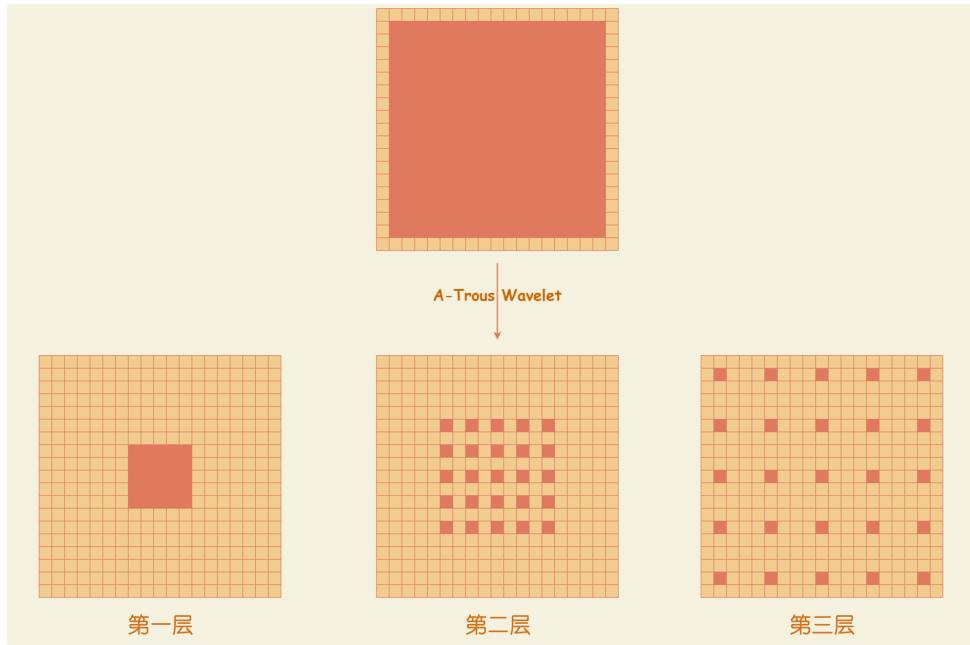
图像空间的滤波通常需要比较大的滤波核（例如 SVGF 论文中使用了  $65 \times 65$ ），逐像素枚举如此大的范围是一个不小的开销，SVGF 使用 A-Trous Wavelet 分层滤波器，每层的滤波核都仅有  $5 \times 5$  大小，但每层的步长都会翻倍；进行 5 层滤波后，就得到了等效  $65 \times 65$  的覆盖范围。图2.6是该滤波器的示意图，注意，它不完全等价于一个单独的  $65 \times 65$  滤波核，但同样可以实现大范围的像素滤波。

图2.7在本系统中展示了联合双边滤波的保边效果，它能够较好地保留图像的几何边界，但对于阴影和反射图像的边界，还不能很好地保留。

### 2.3.4 SVGF

SVGF(Spatiotemporal Variance-Guided Filtering)<sup>[15]</sup> 是一个综合了时间滤波和空间滤波的算法，目的是仅依靠极低的采样率（1spp）重建出较好的效果。它的核心思路是利用方差信息来评估噪声强度，然后动态地调节滤波核的强度，在噪声较大的区域，使用更激进的滤波核。注意，方差仅仅只是噪声的一个不完美的代理，不能完全代表噪声。

SVGF 重建器的输入是 1spp 下渲染的图像，首先在 HDR 空间进行时域样本重用，

图 2.6 A-Trous Wavelet 滤波器，图改编自网络博客<sup>[16]</sup>

(a) 降噪前 (1spp + 时域样本重用)



(b) 高斯滤波



(c) 联合双边滤波



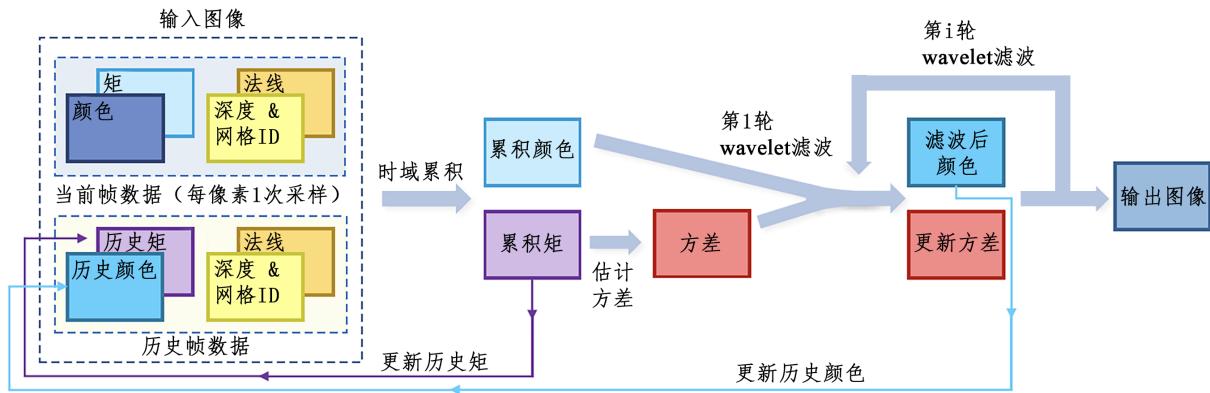
图 2.7 高斯滤波与联合双边滤波的对比

类似 TAA，但需要更严格的一致性检测，并且不对颜色进行 Clamp。这一步的目的是尽可能提高有效样本的个数。参与重用的除了颜色之外，还有亮度的一阶矩和二阶矩 (First and Second Moment)  $\mu_1, \mu_2$ ，用公式  $\sigma^2 = \mu_2 - \mu_1^2$  来计算方差。若重用的有效样本数量过少，则还需要再取相邻  $7 \times 7$  的像素来估计方差。

随后，利用几何信息以及估计的方差信息进行联合双边滤波，方差主要用于控制滤波器对颜色差异的容忍度，像素  $p, q$  之间的颜色差异权重定义为：

$$w_l = \exp\left(-\frac{|l_i(p)-l_i(q)|}{\sigma_l \sqrt{g_{3x3}(Var(l_i(p)))} + \epsilon}\right) \quad (2.7)$$

其中  $\sigma_l$  是一个额外的控制参数， $l_i$  为亮度 (luminance)，在分母中，使用一个额外的

图 2.8 SVGF 重建器，图改编自 SVGF 原论文<sup>[15]</sup>

高斯滤波器对相邻  $3 \times 3$  像素的方差加权平均（这一步可以视为对方差图像的简单滤波，因为方差图像也是有噪声的），最后使用一个 EPS 项避免除零。图2.8展示了完整的 SVGF 重建器。

另一个优化是先将渲染结果拆分为直接光照与间接光照，并各自除去基础色 (albedo，可能来自贴图) 后分别通过上面的重建器降低噪声，最后各自乘上 albedo 后再合并。这样能有效区分出直接光和间接光产生的噪声，避免它们混在一起，除去底色的操作则可以避免模糊掉纹理的边界。SVGF 的重建都发生在 HDR 空间，本文在直接光照和间接光照合并后，再转换到 LDR 空间下进行一次额外的 TAA，降低最终的噪声。

## 2.4 本章小结

本章首先介绍了路径追踪算法以及它的物理依据，然后介绍了路径追踪中的几种加速和降噪算法，总结集成为了本系统的优化方案，包括适应动态场景的 DynamicBVH、NEE + MIS 的采样方式和 SVGF + TAA 的后期降噪。

## 第 3 章 并行光线追踪渲染系统设计与实现

### 3.1 系统概述

#### 3.1.1 开发环境

本系统分为 CPU 端与 GPU 端，CPU 端使用 C++ 语言编程，主要负责系统逻辑；GPU 端使用 GLSL 着色器语言（Shader）编程，主要负责运算。本系统使用 OpenGL 作为渲染硬件接口（RHI）来进行数据传输。

库方面，本系统使用 GLAD 访问 OpenGL 规范接口，GLFW 实现窗口，GLM 辅助线性代数运算，Assimp 用以导入通用模型 3D 模型，ImGui 实现用户界面，OpenCV 进行图像读写。

#### 3.1.2 需求与挑战

要利用 GPU 的算力实现一个并行光线追踪系统，有以下需求与挑战：

- 数据传输与组织：数据在 CPU 与 GPU 之间的传输是一个不小的开销，在动态场景下更需要频繁地更新数据，本系统需要合理组织架构以尽可能减小传输开销。
- GPU 端的算法实现：由于 GPU 核心的控制能力较弱，GPU 编程通常不支持递归、指针和面向对象，算法和数据结构都需要专门的设计。
- 其他支持系统：一个完整的渲染器除了核心的光线传输算法之外，还需要诸如材质系统、光照系统、场景系统等系统的支持，还需要基础的交互功能（UI）和对通用模型的适配功能等。

### 3.2 GPU 上的数据组织与传输

#### 3.2.1 OpenGL 提供的数据接口

OpenGL 提供了一些与 GLSL 着色器交互的方式：1) 使用 uniform 变量传输一些简单的变量，它是只读的，并且不擅长传输大批量的数据，本系统用它来保存一些渲染设置，例如屏幕宽高、相机信息等等；2) Texture（纹理）以图片的形式将数据传

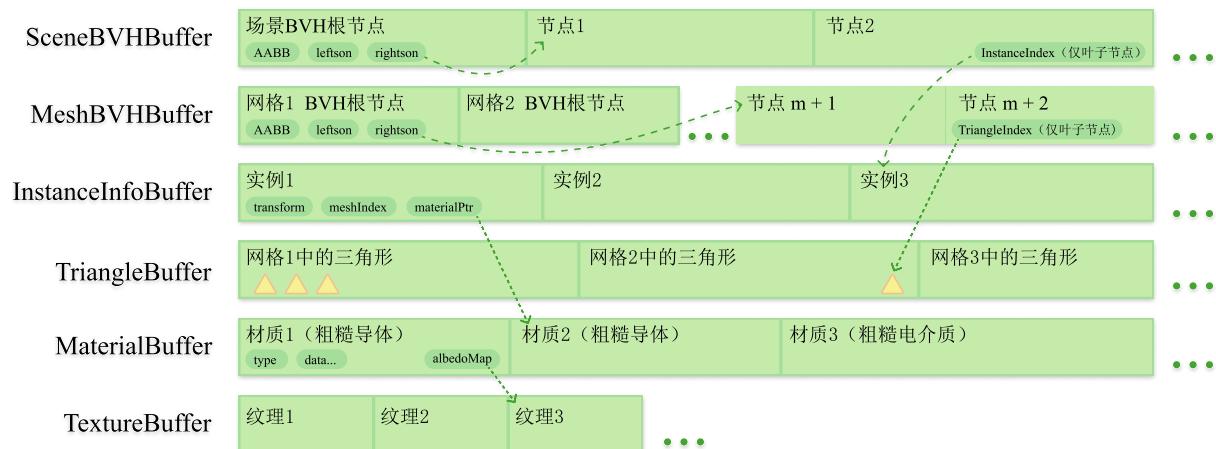


图 3.1 光线追踪部分存储结构示意图

递到 Shader，并自动支持了插值采样，可以用来传递材质的贴图、环境贴图等等；3) FrameBuffer（帧缓冲）是着色器的输出，默认直接输出到屏幕，但也可以绑定到额外的纹理上，用于在渲染的同时存储 GBuffers；4) 其他自定义的大规模数据可以选择 Texture Buffer Object (TBO) 以浮点数组的形式传入，使用 Uniform Buffer Object (UBO) 或 Shader Storage Buffer Object (SSBO) 以结构化的数据形式传入，TBO 与 UBO 都是只读的，SSBO 是可读写的，可以用作着色器的输出。为了避免繁琐的数据转换，本项目统一使用 SSBO 作为自定义数据的载体。

### 3.2.2 场景数据组织

执行光线追踪需要的数据有：场景中所有网格和所有三角形图元、场景加速结构 (DynamicBVH)、每个实例的信息 (材质、变换等)、材质用到的所有纹理。这些要素是相互耦合的，例如材质中要使用到纹理，加速结构需要包含具体的图元素引，等。图3.1展示了这部分的存储架构，它们都是使用 SSBO 存储的结构体。其中，SceneBVH 和 MeshBVH (即2.2.2节描述的场景 BVH 和网格 BVH) 结构相似，都包含 BVH 节点的包围盒 (AABB) 和索引形式的树上指针，SceneBVH 的叶子节点包含实例索引，可以在 InstanceInfoBuffer 中获取实例信息 (变换矩阵、模型索引、材质索引)；MeshBVH 的叶子节点包含三角形图元素索引，指向 TriangleBuffer 中具体的三角形。InstanceInfoBuffer 和 SceneBVHBuffer 每当场景发生变化时就需要更新，但更新它们的

表 3.1 系统中实现的所有 RenderPass

RenderPass名称	主要输入	主要输出	备注
Rasterization	模型顶点数据	GBuffers（深度，法线，UV，基础色等），运动矢量	光栅化替代第一次光线求交
PathTracing	GBuffers，场景数据	直接/间接光照渲染图	路径追踪
SVGFTemporalFilter	渲染图，GBuffers，运动矢量	时域累积的渲染图/一二阶矩，有效样本数	历史帧信息在Pass内管理
SVGFVarianceFilter	渲染图，一二阶矩，有效样本数	方差	
SVGFSpatialFilter	渲染图，方差，GBuffers	滤波后的渲染图	内部进行5轮A-Trous wavelet滤波
SVGFMerge	直接/间接光照渲染图，基础色	合并直接/间接光照后的渲染图	
ToneMappingGamma	HDR渲染图	经过ToneMapping、Gamma校正后的LDR渲染图	
TAA	渲染图，GBuffers，运动矢量	时域累积的渲染图	历史帧信息在Pass内管理
Displayer	任意图	输出到屏幕	
StaticBlender	渲染图	静态时域累积的渲染图	相当于离线渲染

代价较小；MeshBVH 和 TriangleBuffer 都是在模型加载后就建立好的，不需要每帧更新。材质数据（MaterialBuffer）将在3.4节详细介绍。

在 CPU 端，本系统使用管理器模式（Manager），用一个 ResourceManager 管理渲染所需的所有资源（纹理和网格）并负责组织 Buffer 数据。在渲染前，ResourceManager 构建 MeshBVHBuffer 和 TriangleBuffer 并传入 GPU；每帧，本系统更新场景中的所有实例信息，按照各个实例在当前帧的位置来重建 SceneBVH，再构建 Buffer 并传入 GPU。材质被视为实例的一种属性（而非资源）每帧同步更新；纹理则是利用了 OpenGL 的 Bindless Texture，在创建之初时就加载到 GPU 中，每帧需要更新的只有索引。

这种设计在诸多耦合的信息中解耦了两层 BVH，使得 MeshBVHBuffer 和 TriangleBuffer 能够在动态场景下保持不变，让每帧需要传输的数据规模取决于实例数而非三角形个数，大大降低了开销。

### 3.2.3 RenderPass

由于图形渲染中涉及大量的后处理（Post-processing）任务，包括时空降噪、色调映射等等，这些任务很难在一次并行计算中同时完成，通常是将渲染结果再次输入 GPU 来进行后处理运算。本文将“数据输入 GPU，经过一系列并行运算，最后得到输出的过程”笼统地称为一个 Pass。多个 Pass 组合就成为了一个完整的渲染管线。

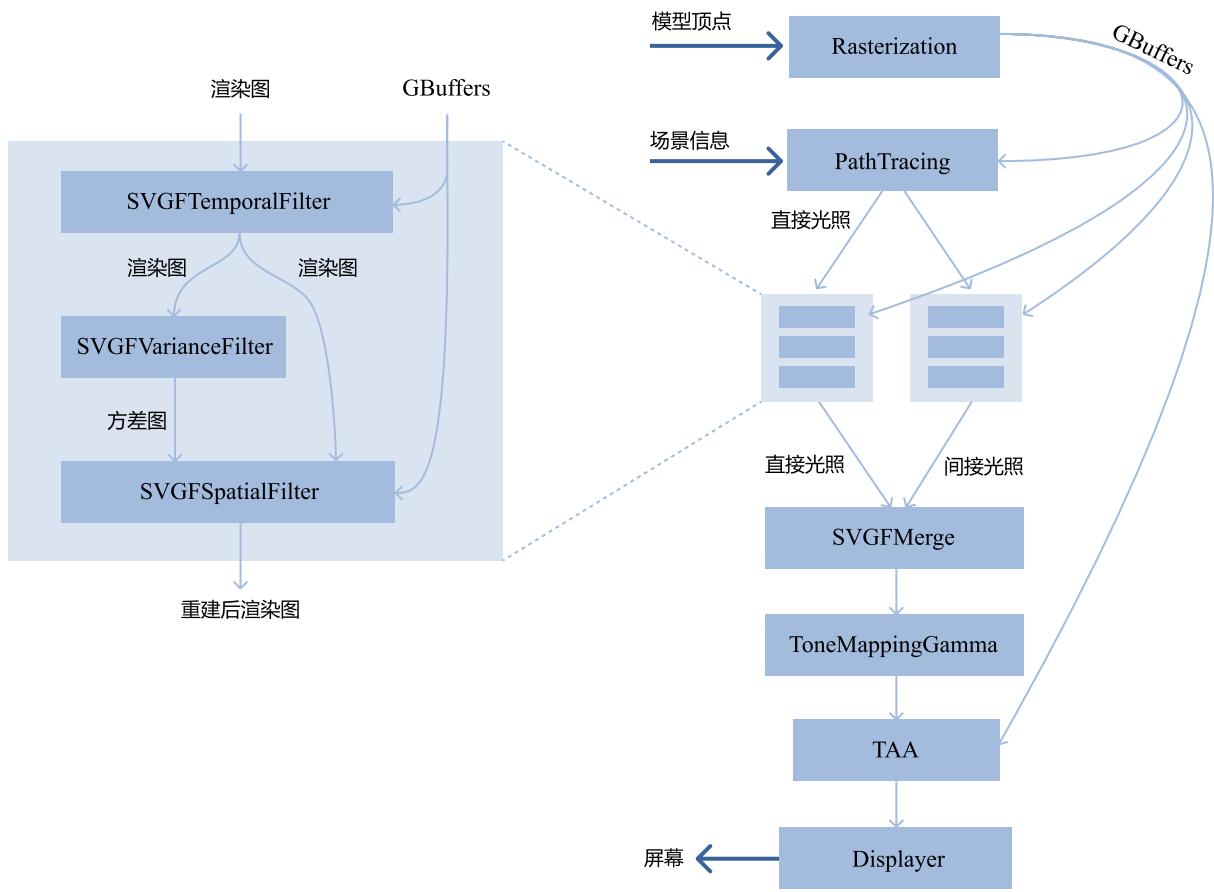


图 3.2 RenderPass 组合而成的完整渲染管线

在 CPU 端，本系统抽象出了 RenderPass 的概念，每个 RenderPass 拥有一个独立的 Shader 作为计算脚本，可以是 Compute shader 或 Pixel shader，它以一个或数个缓冲 (SSBO 或帧缓冲) 作为输入，计算结果存储在一个或数个输出缓冲中。本系统将缓冲都设置为只读或只写的，禁止了原地修改（尽管 SSBO 本身支持），这样设计是为了让输入和输出彻底分离，避免了并行计算时的资源冲突问题。在渲染管线中，每个 RenderPass (除了最初的渲染 Pass) 都以之前的 RenderPass 的输出作为输入，并且允许保留一些自己的缓存（例如，时域样本重用的 Pass 可以自行存储历史信息）。

除了 PathTracing 输入的场景数据和 Rasterization 输入的顶点数据之外，其他各类 RenderPass 的输入输出都是一个屏幕大小的缓冲，包含逐像素的信息。

本系统基于 RenderPass 的设计封装了所有的渲染流程，表3.1列出了系统中实现的所有 RenderPass，图3.2展示了由这些 RenderPass 组成的完整渲染管线，其中，Rasterization 是光栅化渲染管线，可以用于替代光线追踪的第一次光线求交以降低开销

(在成熟的 RHI 支持下，一次不着色的光栅化开销几乎可以忽略不计)，交点信息保存在 GBuffer 中（包括深度、法线、UV、实例 ID 等）并传递给光线追踪 Pass。StaticBlender 是面向静态场景、静态相机的时域累积 Pass（相当于离线光线追踪），用于得到路径追踪的无偏真实结果，它在渲染管线中启用时，插人在 ToneMapping 之前。

### 3.3 GPU 端算法实现

#### 3.3.1 路径追踪实现

算法1给出了本系统的路径追踪伪代码，采样使用 MIS+NEE 方式（具体在2.3.1节描述）。由于路径追踪仅递归追踪一条光线，可以将其转换为迭代式写法。具体来说，该算法追踪一条光路，每次迭代采样下一次反弹方向，使用一个历史变量记录整条路径的衰减（包括每次反弹时的 BSDF 项和 cos 项的累乘），在击中光源时，根据历史变量直接计算光源辐射对于最终结果的贡献。

注意，严格无偏的路径追踪使用轮盘赌选择法（Roulette wheel selection）每次以一定概率选择是否继续弹射，并将后续弹射的贡献除以此概率。该方法用于无偏地估计无限次弹射的光路贡献。但由于 GPU 运算的特性，一批并行的路径追踪的瓶颈往往取决于最慢的数次追踪，这个方法会对性能产生不小的影响，因此本系统不使用此方法，而是使用一个设定的最大弹射深度，舍弃更高阶的间接光照。

#### 3.3.2 DynamicBVH 求交实现

算法2给出了光线与双层 BVH 求交伪代码，该算法使用一个栈记录待求交的节点，模拟树上深度优先搜索的过程。使用深度优先搜索而非广度优先搜索的原因是：该算法希望尽快更新最近交点（记为  $t_{near}$ ），此后，若与光线某个节点的包围盒有交点，但远于最近交点  $t_{near}$ ，也可以直接舍弃整棵子树。而广度优先搜索则几乎会将所有可能有交点的节点全部入栈，造成不必要的开销。

求交首先在场景 BVH 上进行，当达到一个叶子节点时，跳转到对应实例的网格 BVH 上继续求交。注意网格 BVH 存在局部空间，而场景 BVH 存在世界空间，跳转时

需要将光线变换到局部空间，求得交点后需要变换回世界空间。

### 3.4 材质系统

双向散射分布函数（BSDF）用于描述表面材质的光学性质，具体来说，BSDF 的输入为入射方向  $\omega_i$  和出射方向  $\omega_o$ ，输出为来自  $\omega_i$  方向的辐射经过表面后散射到  $\omega_o$  方向的比例，严格的定义是  $f_s = \frac{dL_o}{dE_i}$ ，其中  $L$  是辐射（radiance）， $E$  是辐照度（irradiance）。

人们已经提出了许多 BSDF 模型，例如经验性的冯氏模型，基于物理的微表面模型，还有用于表达特殊材质的各种模型（布料、皮肤模型等等）等等。一个完善的材质系统应当支持自定义一个材质，因此，本系统开放了对材质的定义，支持以 Shader 的形式编写自定义的 BSDF。一个材质需要实现几个核心部分：evalBSDF, sampleBSDF 和 pdfBSDF。evalBSDF 即 BSDF 函数的计算；sampleBSDF 是仅给出入射方向  $\omega_i$ ，需要对 BSDF 进行重要性采样得到出射方向，并返回 BSDF 值和采样概率；pdfBSDF 是给出入射和出射方向，计算采样到此出射方向的概率，pdf 应当和 sample 匹配。算法1中涵概了这几个函数在路径追踪中的运用。

为了实现上述自定义功能，本系统基于指针分发实现了一种伪面向对象的机制。具体而言，每一种自定义的材质类型都需要有一个类型 ID，并可能有若干属性。在 CPU 端，用户可以设置各个物体的材质类型和属性，渲染时，ResourceManager 会负责构建 MaterialBuffer，它将所有实例的材质信息串联到一维浮点数组中，每个材质总是以类型 ID 开头，后面跟着自己的属性列表，每种材质可以有不同的属性长度。构建 MaterialBuffer 的同时，每个实例也得知了自己的材质在 Buffer 中的起始地址。在 GPU 端，本系统定义了 eval, sample 和 pdf 的母函数，输入材质的起始地址，母函数会根据起始地址的类型 ID，分发给相应的子函数。算法3中的伪代码展示了指针分发的过程和一个纯漫反射材质的定义。

作为样例，本系统中实现了基于微表面模型的粗糙导体和粗糙电介质两种材质，具体实现可以在开源代码仓库中找到，此处不再展开。

---

**Algorithm 1:** Path tracing

---

**Input:** 光线  $ray = (origin, toward)$   
**Output:** 直接光照  $di$ , 全局光照  $gi$

```
1 function shade(ray)
2      $di, gi \leftarrow 0;$ 
3      $history \leftarrow 1;$  // 光线路径上累积的权重
4      $isect \leftarrow rayIntersectScene(ray);$ 
5     if  $isect$  on light then
6          $di, gi \leftarrow evalLight(isect);$ 
7         return  $di, gi;$ 
8     end
9     for  $dep \leftarrow 1$  to  $MAXDEP$  do
10         $coord, material \leftarrow fetchIntersection(isect);$ 
11         $\omega_i \leftarrow toLocal(coord, -ray.toward);$ 
12        // 光源采样, 返回采样方向、概率和辐射
13         $sdir, p_l, radiance \leftarrow sampleLight(isect);$ 
14         $tsect \leftarrow rayIntersectScene(isect.position, sdir);$ 
15        if  $tsect$  on light then
16             $\omega_o \leftarrow toLocal(coord, sdir);$ 
17             $f_s \leftarrow evalBSDF(material, \omega_i, \omega_o);$ 
18             $p_s \leftarrow pdfBSDF(material, \omega_i, \omega_o);$ 
19             $gi \leftarrow gi + history \times radiance \times f_s \times cos(\omega_o) / (p_l + p_s);$  // MIS
20            ;
21        end
22        // BSDF 采样, 输入入射方向, 返回出射方向、概率和 BSDF 值
23         $\omega_o, p_s, f_s \leftarrow sampleBSDF(material, \omega_i);$ 
24         $sdir \leftarrow toWorld(coord, \omega_o);$ 
25         $tsect \leftarrow rayIntersectScene(isect.position, sdir);$ 
26        if  $tsect$  on light then
27             $radiance \leftarrow evalLight(tsect);$ 
28             $p_l \leftarrow pdfLight(tsect);$ 
29             $gi \leftarrow gi + history \times radiance \times f_s \times cos(\omega_o) / (p_s + p_l);$ 
30            if  $dep == 1$  then  $di \leftarrow gi;$ 
31            return  $di, gi;$ 
32        else
33            // history 记录本次弹射的衰减, 继续追踪光线
34             $ray \leftarrow Ray(isect.position, sdir);$ 
35             $isect \leftarrow tsect;$ 
36             $history \leftarrow history \times f_s \times cos(\omega_o) / p_s;$  // 间接光方向不使用 MIS (仅 BSDF 采样)
37        end
38    end
39    return  $di, gi;$ 
40 end
```

---

**Algorithm 2:** BVH intersection

---

**Input:** 光线  $ray = (origin, toward)$

**Output:** 最近交点  $nearest$

```

1 global stack, head ;
2 function IntersectMeshBVH(instanceID, ray, nearest)
3   meshID, w2l  $\leftarrow$  getInstanceInfo(instanceID); // 获取网格索引、世界  $\rightarrow$  局部的变换矩阵
4   sthead  $\leftarrow$  head ;
5   head  $\leftarrow$  head + 1 ;
6   stack[head]  $\leftarrow$  meshID ;
7   ray  $\leftarrow$  w2l  $\times$  ray ; // Local space ray
8   while head > sthead do
9     cur  $\leftarrow$  getMeshBVHNodeByID(stack[head]);
10    head  $\leftarrow$  head - 1 ;
11    if cur.isLeaf then
12      isect  $\leftarrow$  IntersectTriangle(ray, cur.triangleID) ;
13      if isect.exist and isect.time < nearest.time then
14        nearest  $\leftarrow$  localToWorldIntersection(w2l, isect) ;
        // 法线变换有:  $(l2w^{-1})^T \times n = w2l^T \times n = n^T \times w2l$ , 无需求逆
15      end
16    else
17      stack[head + 1]  $\leftarrow$  cur.leftsonID ;
18      stack[head + 2]  $\leftarrow$  cur.rightsonID ;
19      head  $\leftarrow$  head + 2 ;
20    end
21  end
22 end
23 function IntersectSceneBVH(ray)
24   nearest  $\leftarrow$  inf ;
25   stack[0]  $\leftarrow$  sceneBVHRootID ;
26   head  $\leftarrow$  1 ;
27   while head > 0 do
28     cur  $\leftarrow$  getSceneBVHNodeByID(stack[head]);
29     head  $\leftarrow$  head - 1 ;
30     isect  $\leftarrow$  IntersectBoundingBox(cur.boundingbox) ;
31     if isect.exist and isect.time < nearest.time then
32       if cur.isLeaf then
33         IntersectMeshBVH(cur.instanceID, ray, nearest) ;
         // nearest 传递引用, 交由该函数更新
34       else
35         stack[head + 1]  $\leftarrow$  cur.leftsonID ;
36         stack[head + 2]  $\leftarrow$  cur.rightsonID ;
37         head  $\leftarrow$  head + 2 ;
38       end
39     end
40   end
41   return nearest ;
42 end

```

---

---

**Algorithm 3:** Material system

---

```

1 function evalBSDF(maddr,  $\omega_i$ ,  $\omega_o$ )
2   typeID  $\leftarrow$  materialBuffer[maddr];
3   if typeID = 1 then return evalDiffuse(maddr,  $\omega_i$ ,  $\omega_o$ );
4   if typeID = 2 then return evalRoughConductor(maddr,  $\omega_i$ ,  $\omega_o$ );
5   ...
6   return NULL;
7 end
8 function sampleBSDF(maddr,  $\omega_i$ )
9   typeID  $\leftarrow$  materialBuffer[maddr];
10  if typeID = 1 then return sampleDiffuse(maddr,  $\omega_i$ );
11  if typeID = 2 then return sampleRoughConductor(maddr,  $\omega_i$ );
12  ...
13  return NULL;
14 end
15 function pdfBSDF(maddr,  $\omega_i$ ,  $\omega_o$ )
16  typeID  $\leftarrow$  materialBuffer[maddr];
17  if typeID = 1 then return pdfDiffuse(maddr,  $\omega_i$ ,  $\omega_o$ );
18  if typeID = 2 then return pdfRoughConductor(maddr,  $\omega_i$ ,  $\omega_o$ );
19  ...
20  return NULL;
21 end
22 function evalDiffuse(maddr,  $\omega_i$ ,  $\omega_o$ )
23  if  $\cos(\omega_o) < 0$  then return 0.0;
24  albedo  $\leftarrow$  materialBuffer[maddr + 1: maddr + 4];
  // 该材质仅有一个颜色属性，放在首地址之后三位。
25  return albedo/ $\pi$ ; // 半球面上均匀分布的 Lambertian 项
26 end
27 function sampleDiffuse(maddr,  $\omega_i$ )
28   $\omega_o \leftarrow$  sampleHemisphere(); // 半球面均匀采样
29  pdf  $\leftarrow$  1.0/( $2\pi$ );
30  return  $\omega_o$ , pdf, evalDiffuse(maddr,  $\omega_i$ ,  $\omega_o$ )
31 end
32 function pdfDiffuse(maddr,  $\omega_i$ ,  $\omega_o$ )
33  if  $\cos(\omega_o) < 0$  then return 0.0;
34  return 1.0/( $2\pi$ )
35 end

```

---

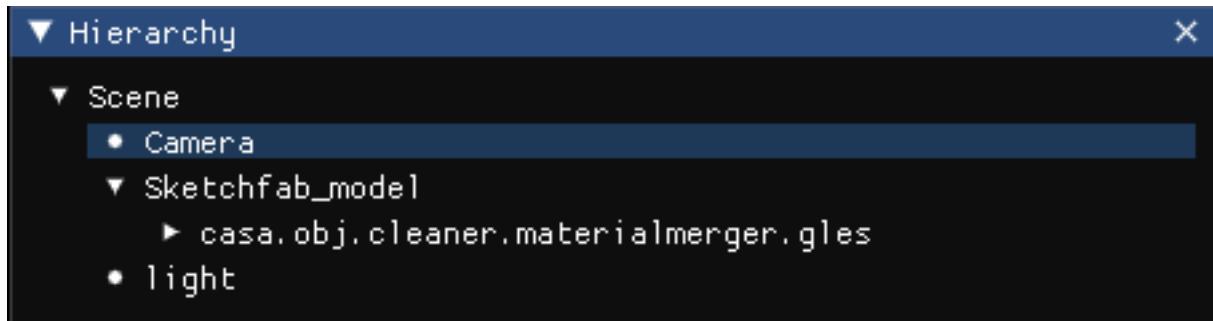


图 3.3 一个分层的场景示意图

## 3.5 场景系统

在各种 3D 软件中，场景都使用一种分层的树形结构组织，本系统也实现了类似的层级结构。场景的唯一单位是实例（Instance），除了代表场景的根实例之外，每个实例都属于另一个父实例，构成了一个树形结构，见图3.3。实例具有 Transform（相对于其父实例的变换矩阵），一个实例要从局部坐标变换到世界坐标，需要乘上它到根节点路径上的所有变换矩阵。实例可以拥有资源（网格、纹理和光照），拥有网格的实例将被渲染。

注意，实例是一个抽象概念，而网格具体的是几何数据资源，一个实例可以没有网格，一个网格也可以被多个实例使用。实例是分层组织的，而资源由 ResourceManager 统一管理，由一个线性表存储，实例仅拥有资源的索引。通用模型导入库 Assimp 也使用了类似的场景结构，本系统在加载模型时将 Assimp 的数据结构转接到自己的数据结构中，实现了通用模型的导入。

每一帧，ResourceManager 负责从场景中提取所有实例的信息（包括预处理出所有实例到世界空间的完整变换矩阵、实例使用的资源索引等）并传入 GPU，GPU 不需要关心具体的场景结构。

## 3.6 交互系统

本系统使用 ImGui 制作用户界面，如图3.4所示，界面上包括四个组成部分：

- 左上角窗口显示渲染结果，选中该窗口后，可以鼠标旋转镜头方向，键盘 WASD

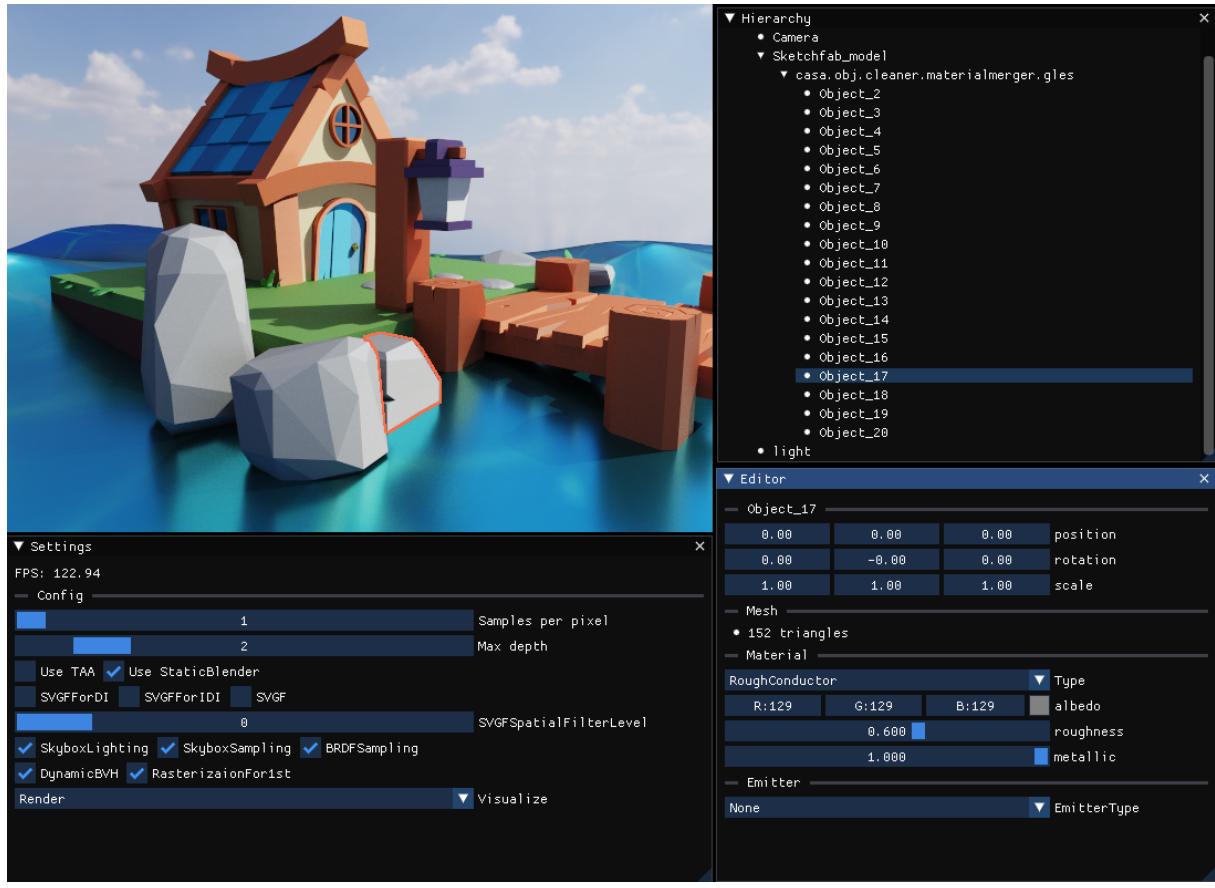


图 3.4 用户界面

在场景中移动，T 键截取当前帧画面，以及一些其他的 Debug 用按键，这部分主要调取的是 GLFW 的窗口 API。鼠标左键可以选中场景中的物体，选中的物体以橙色边框高亮显示（如图中选中的石头），并自动在右侧 Hierarchy 窗口中聚焦，在 Editor 窗口中打开。这部分利用了渲染中得到的实例 ID 缓冲实现，该缓冲包含了每个像素对应的实例编号，高亮则是在渲染管线的最后（DisplayerPass 中）通过像素周围的实例 ID 来简单判定物体边缘。

- 左下角的浮窗包含了一些渲染设置，包括本系统的实时路径追踪算法中的各个组件和一些优化选项的开关。最后一行是可视化选项，默认是展示最终渲染结果，也可以切换为渲染中产生的各个中间结果和 GBuffer（法线，深度，直接光照和间接光照等）。
- 右上角浮窗 Hierarchy 展示了场景的层级结构，这里的一项即一个实例，在此处选中一个实例，会在场景中高亮显示，同时在下方 Editor 中打开。右键实例可以删

除或在实例下导入新的 3D 模型。

- 右下角浮窗 Editor 即编辑器，选中实例后可以更改其变化矩阵（通过更改位置、旋转、缩放的形式），若实例包含网格，则还会显示网格信息以及材质，材质可以更改为任一种已定义的材质，自定义的材质需要在 CPU 端实现 UI 插入，即选择要暴露哪些参数到用户界面。注意，在本系统中材质被视为实例的属性而非资源。最后一行可以为实例附着光源。

### 3.7 本章小结

本章详细介绍了一个 GPU 上并行的光线追踪系统实现，包括 GPU 端的存储结构和数据加载方案、CPU 端的 RenderPass 的概念，并给出了无递归、无指针的核心算法实现。最后介绍了系统中的其他支持子系统，包括材质系统、场景系统和交互系统。

## 第 4 章 系统结果展示与分析

### 4.1 场景编辑示例

图4.1演示了系统的编辑功能。a) 系统启动时默认载入了一个水面上的房屋场景，先在右键 Scene 导入一个兔子模型（Assimp 自动提取了它的材质信息，因此它看起来是棕色）；b) 随后选中路灯删除，再选中兔子，在 Editor 中编辑模型的位置、缩放和朝向，让这只兔子站在路灯的上方，并修改材质的粗糙度和金属度，让它变成一只有黄金质感的兔子；c) 接下来在 Setting 窗口中关闭环境光照，再为兔子添加一个新的子实例（空）并在上面附着一个点光源，调整空实例的原点到兔子正前方。旋转兔子，光源跟随旋转并始终处于兔子正前方。

### 4.2 可视化

图4.2可视化了渲染过程中产生的一些中间结果和 GBuffer，其中深度 (b) 和实例 ID (d) 经过缩放后呈现，间接光照 (e) 和直接光照 (f) 都是已除去基础色的结果。

### 4.3 实时渲染效果展示

本节在几个有挑战性的场景下测试了本系统的性能和画面质量，使用的设备是 NVIDIA GeForce RTX 4060 Laptop GPU，未使用该 GPU 的任何专用光线追踪优化功能。

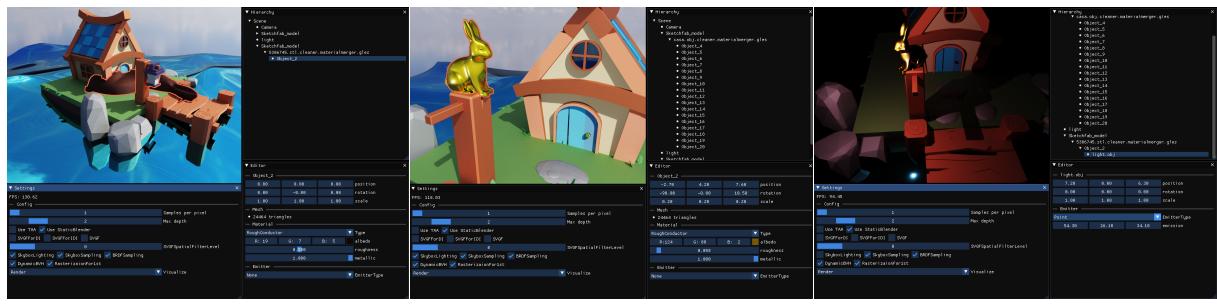


图 4.1 场景编辑示例

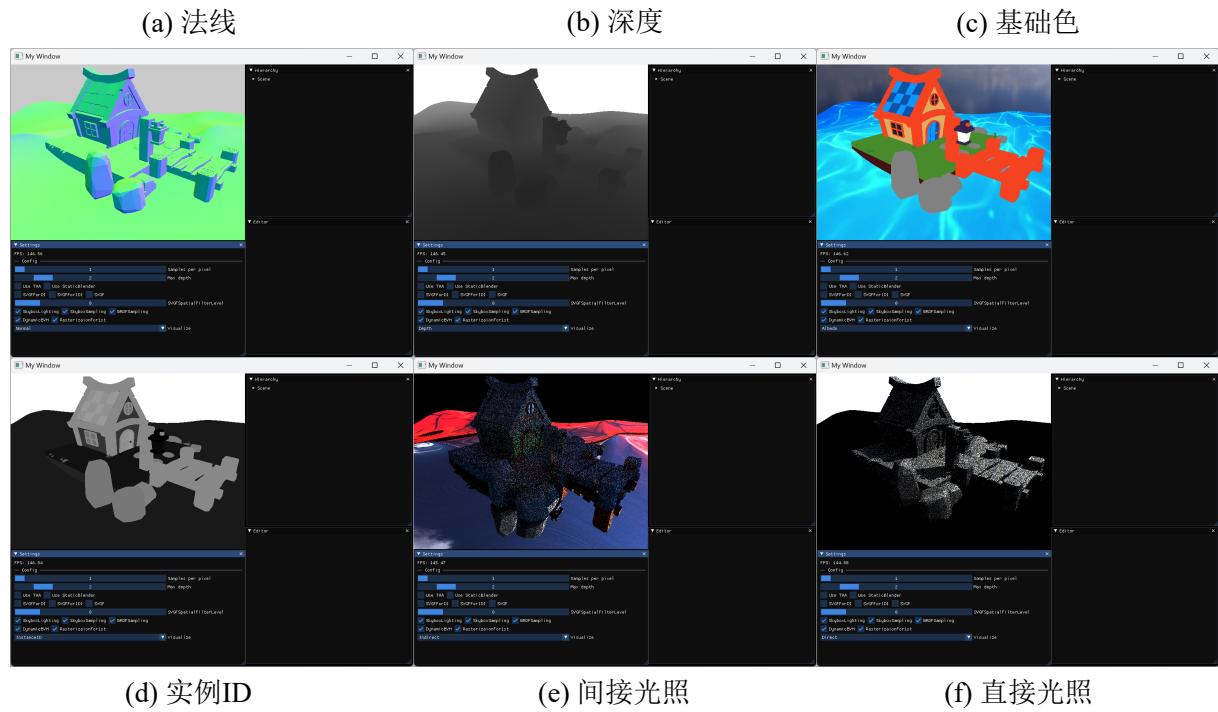


图 4.2 几何信息和中间结果可视化

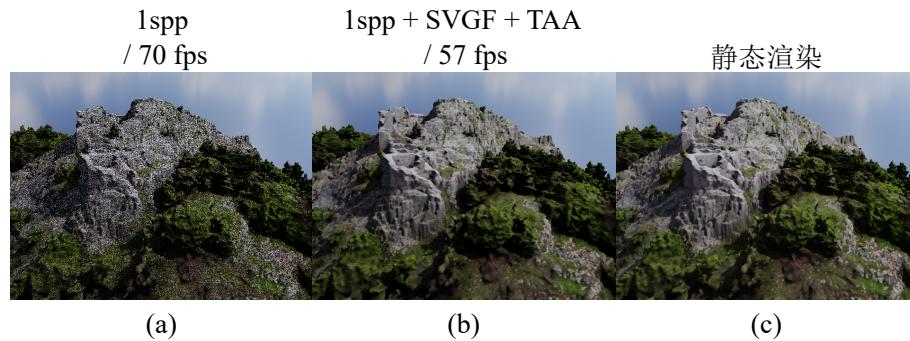


图 4.3 大尺度户外场景渲染

### 4.3.1 大尺度户外场景

图4.3展示了本系统在大尺度户外场景下的性能，该山地场景具有 1.7M 个三角形，83 万个顶点，是前文演示场景的 40 倍，但 DynamicBVH 最大深度依然只有 30，在 1spp 下 (a) 仍然可以跑到 70fps，几乎没有性能损失，这显示了 DynamicBVH 结构对大规模图元的承载能力。当然，随着图元个数的进一步增大，显存将率先遇到瓶颈。

另外可以看出，在户外直接光照为主导的环境下，本系统的实时渲染结果经过降噪 (b) 已经非常接近离线渲染的参考结果 (c)。

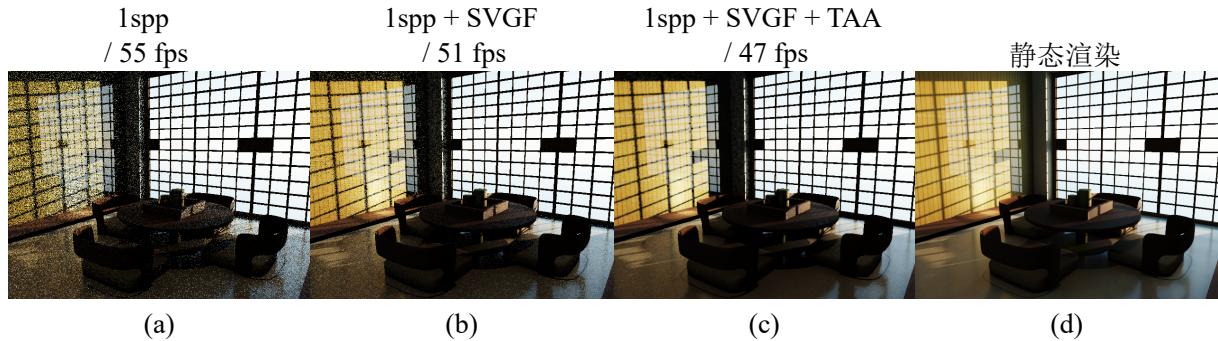


图 4.4 室内场景渲染

### 4.3.2 室内场景

图4.4展示了本系统在室内的渲染效果，室内环境对于光线追踪比较有挑战性，因为需要通过复杂的间接光照来将场景点亮。本例中最大反弹次数设置为 8 次，因此降低了一些性能，但依然可以在实时标准下得到不错的效果。

降噪方面，本文给出了不使用 TAA (b) 和使用 TAA (c) 的两种结果，因为 TAA 是在 LDR 范围内进行的，擅长抹去极高亮度的噪点，可以更有效地抑制噪声，但也会一定程度上导致能量的损失（在后续的几组结果中，本文同样给出 TAA 开/关的两组结果）。SVGF 的空间滤波在此处（包括在后续几组渲染结果中）没有启用，因为它不擅长在复杂的光照环境下保留每一处边界，可以在简单的场景中开启它。

### 4.3.3 水下场景

图4.5展示了本系统在水面场景的渲染效果，水面使用了自定义的粗糙电介质材质，其粗糙度设置为 0.01，折射率设置为 1.2。图中可以清晰地看到折射时水底物体“变浅”的效果以及菲涅尔效应 (d)，这得益于本系统使用的基于物理的路径追踪和微表面材质模型，这种效果是传统的光栅化方法难以实现的。在降噪后，天空中云的反射高光有一些减弱 (b, c)，但总体来说有着不错的效果。

### 4.3.4 复杂散射场景

图4.6展示了本系统渲染一个玻璃瓶的效果，这个玻璃瓶的焦散现象被静态渲染器 (d) 很好地呈现出来。在实时渲染中，焦散的高光不容易采样，本文在保证帧率的同时

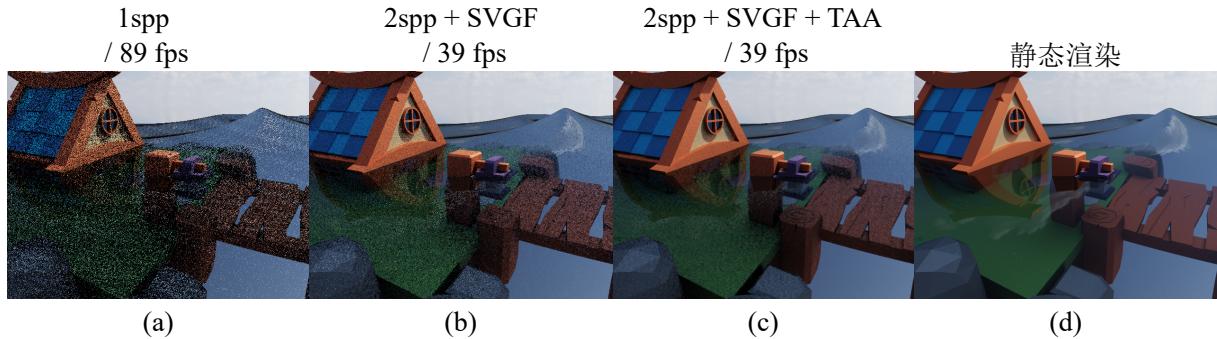


图 4.5 水下场景渲染

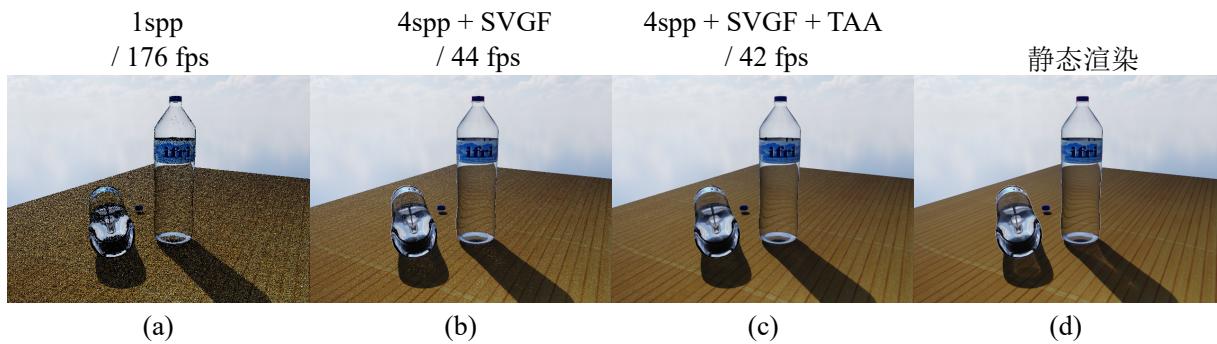


图 4.6 玻璃瓶渲染

尽可能提高了 spp 数，以在渲染结果中看到焦散现象 (b)，TAA 的版本 (c) 中噪声更少，但焦散的高光也会被减弱。

#### 4.4 本章小结

本章展示了本系统的交互功能和可视化功能，然后在几个有挑战性的场景下测试了本系统的性能和画面质量，结果表明，本系统能够胜任复杂场景的实时渲染任务。

## 第 5 章 总结与展望

### 5.1 论文总结

本文调研了现有的实时光线追踪加速和降噪算法，归纳总结并优化后形成了自己的方案。本文利用 OpenGL 提供的接口和 GLSL 编程语言，实现了一个可以在通用 GPU 上运行的实时光线追踪渲染系统，无需专用硬件加速即可以实时标准渲染高品质的画面，并且在几个较为复杂的场景下也能够实现不错的帧率与画面质量。最后，该系统也提供了基础的交互功能，支持用户自由搭建场景并渲染。

### 5.2 不足之处

受限于开发时间与本人水平，本文仍有许多不足之处：

- 多光源支持较差。本系统的光源系统仍然比较粗糙，光源采样方式比较暴力，应对多光源场景会较为乏力。
- 系统交互功能较为简陋。虽然交互功能不是本项目的核心，但一个易于使用的渲染器应当具有更强大的交互功能。
- 对 RHI 的更高抽象。本系统目前仅支持 OpenGL，而现有的成熟渲染器通常会对 RHI 做更高一层抽象，用户可以根据使用场景切换为其他 RHI。

### 5.3 未来展望

后续开发中，本系统计划加入的算法或功能有：

- 多光源支持：近年来，有一些较新的工作致力于解决多光源采样的问题，如 LightBVH<sup>[17]</sup>、ReSTIR DI<sup>[8]</sup> 等，未来会考虑将它们集成进来。
- 更强大的交互功能：包括在场景中直接移动、缩放和旋转物体；一些通用的快捷键等等。
- 更高程度的开放：本系统当前已经开放了材质系统，希望在未来能够将整个渲染管线开放给用户，支持定制化的渲染功能。

## 参 考 文 献

- [1] De Vries J. Learn OpenGL[EB/OL]. (2014-06)[2024-05-13]. <https://learnopengl.com/>.
- [2] Usman Saleem. Ray Tracing vs Rasterized Rendering –Explained[EB/OL]. (2021-07-06)[2023-05-13]. <https://appuals.com/ray-tracing-vs-rasterized-rendering-explained/>.
- [3] Whitted T. An improved illumination model for shaded display[C]//Proceedings of the 6th annual conference on Computer graphics and interactive techniques. 1979: 14.
- [4] Kajiya J T. The rendering equation[C]//Proceedings of the 13th annual conference on Computer graphics and interactive techniques. 1986: 143-150.
- [5] 闫润, 黄立波, 郭辉, 等. 实时光线追踪相关研究综述 [J]. 计算机科学与探索, 2023, 17(02): 263-278.
- [6] Admin. A Macro View of Nanite[EB/OL]. (2021-05-30)[2024-04-29]. <https://www.elopezr.com/a-macro-view-of-nanite>.
- [7] Tokdar S T, Kass R E. Importance sampling: a review[J]. Wiley Interdisciplinary Reviews: Computational Statistics, 2010, 2(1): 54-60.
- [8] Bitterli B, Wyman C, Pharr M, et al. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting[J]. ACM Transactions on Graphics (TOG), 2020, 39(4): 148: 1-148: 17.
- [9] Talbot J F. Importance resampling for global illumination[M]. Brigham Young University, 2005.
- [10] Vitter J S. Random sampling with a reservoir[J]. ACM Transactions on Mathematical Software (TOMS), 1985, 11(1): 37-57.
- [11] Lafortune E P, Willems Y D. A 5D tree to reduce the variance of Monte Carlo ray tracing[C]//Proceedings of the Eurographics Workshop in Dublin. 1995: 11-20.
- [12] Dong H, Wang G, Li S. Neural Parametric Mixtures for Path Guiding[C]//ACM SIGGRAPH 2023 Conference Proceedings. 2023: 1-10.
- [13] Nehab D, Sander P V, Lawrence J, et al. Accelerating real-time shading with reverse reprojection caching[C]//Graphics hardware. 2007, 41: 61-62.
- [14] Kopf J, Cohen M F, Lischinski D, et al. Joint bilateral upsampling[J]. ACM Transactions on Graphics (ToG), 2007, 26(3): 96-es.
- [15] Schied C, Kaplanyan A, Wyman C, et al. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination[C]//Proceedings of High Performance Graphics. 2017: 1-12.
- [16] 花桑. GAMES202 作业集 [EB/OL]. (2023-02-16)[2024-05-13]. <https://www.drfower.top/tags/GAMES202/>.
- [17] Conty Estevez A, Kulla C. Importance sampling of many lights with adaptive tree splitting[J]. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 2018, 1(2): 1-17.

## 致 谢

感谢我的研究生导师王贝贝老师，在我的第一段科研中给了我非常多指导和帮助，让我能够在大四时发表自己的第一篇论文。感谢杨文武老师，在大三给了我最初的科研训练，并指导我完成了这篇毕业论文。感谢在大学期间一直以来帮助、指导我的辅导员和授课老师们。

感谢我的 ACM 队友俞飞洋、何天益，和我一直从铜牌打到金牌。感谢 ACM\_CLUB，信息楼 414 是我大学四年里我待过时间最长的地方，我在这里打 ACM，也在这里复习期末、做项目、搞科研。ACM 见证了我的成长，带给了我最初的自信，让我认识了一群强而有力的伙伴和未来可期的后辈。祝 ACM\_CLUB 越来越好！

感谢在这四年里为我引路的前辈们：感谢 18 级的胡柯青学长，他用耿直而犀利的建议消解了我的焦虑和迷茫，让我能够专心投入 ACM 竞赛；感谢南开的李子轩师兄带我熟悉科研，并在 SIGGRAPH 赶稿的最后阶段帮了我一把，如雪中送炭；感谢不鸣科技的叶佳伟师兄带我实习，带我接触大型项目，让我在如此短的实习期内能有所收获。也要感谢这四年里，为我解惑的每一位前辈。

感谢在我大二遇到困难时，帮助我的所有亲人、老师、同学，还有陌生人们。

最后要感谢我的父母，感谢你们数十年如一日的陪伴，感谢你们一直以来的支持。