

VCL Final Project:Path Tracing

颜锦阳

2026-01-09

1 简要介绍

Path Tracing 不同于 Lab 3 中实现的 Whitted-style Ray Tracing, 是基于渲染方程的全局光照渲染方法, 是现代基于物理真实感渲染的基础。在本次大作业中我基于原有的lab代码, 实现了Path Tracing, 并且与Whitted-style Ray Tracing进行了对比。

在命令行中输入xmake run final即可运行代码。

2 代码实现

在原有代码基础上新写了CasePathTracing.cpp, CasePathTracing.h, PathTracing.cpp, PathTracing.h四个文件。

2.1 原理部分

原理的代码主要在PathTracing.cpp中。

- 首先我们定义了一个随机数生成器, 用于蒙特卡洛积分随机采样。然后定义三个采样函数:

SampleHemisphereUniform: 在单位半球上均匀采样。

SampleHemisphereCosine: 基于余弦权重的半球采样 (重要性采样, 符合Lambertian漫反射)。

SampleHemisphereImportance: 根据粗糙度选择均匀采样或余弦采样 (用于镜面反射和漫反射的混合)。

- 随后实现课上讲的BRDF方法。

Evaluate函数: 根据Cook-Torrance模型计算BRDF值, 包含漫反射和镜面反射项, 并考虑能量守恒。

Sample函数: 根据BRDF属性进行重要性采样, 生成新的光线方向, 并计算该方向的概率密度函数 (PDF)。

PDF函数: 计算给定方向的PDF。

CreateBRDFFromMaterial函数: 根据传入的albedo和metaSpec (包含镜面反射和粗糙度信息) 创建BRDF。

- 接下来是直接光照采样和环境光采样部分, 直接光照从场景中随机选择一个光源, 计算该光源对当前着色点的直接光照贡献。环境光我采用了简单的均匀天空光。

- 然后是核心函数PathTrace: 跟踪一条光线, 进行多次弹射 (maxBounces次)。每次与场景相交, 计算自发光 (如果有)、直接光照 (如果启用Next Event Estimation) 和间接光照。间接光照通过BRDF采样新的方向, 并更新吞吐量 (throughput)。使用俄罗斯轮盘赌 (Russian Roulette) 来终止光线, 以控制计算量。

- 最后是渐进式Path Tracer (ProgressivePathTracer): 维护一个累积缓冲区 (accumulator) 和一个当前帧缓冲区 (buffer)。每次渲染一帧时, 对每个像素进行多次采样 (samplesPerPixel), 调用PathTrace函数。将每次采样的结果累积, 并计算平均值, 得到当前像素的估计值。最后对颜色进行伽马校正 (2.2的倒数次幂) 并存储到 buffer中。

2.2 UI交互部分

UI的代码主要在CasePathTracing.cpp中。我在lab原有代码基础上设置了一系列可调的参数。

- samplesPerPixel: 控制每个像素的蒙特卡洛采样次数, 范围通常 1-512, 值越小 (如1-4): 渲染速度快, 但噪声很大 (颗粒感明显)。值越大 (如64-256): 渲染质量高, 噪声减少, 但渲染时间线性增加。

- maxBounces: 限制光线在场景中的最大反弹次数, 范围通常 1-20, 值小 (如1-3): 只有直接光照, 没有间接光照, 渲染快但效果不真实。值大 (如5-8): 能模拟复杂的间接光照效果 (如颜色渗透、柔和阴影)。
- superSampleRate: 抗锯齿采样, 在像素内进行细分采样。值=1: 无抗锯齿, 边缘可能有锯齿。值=2: 每个像素4个子采样, 减少锯齿。值=4: 每个像素16个子采样, 高质量抗锯齿。
- enableDirectLighting: 是否计算来自光源的直接光照。
- enableRussianRoulette: 自适应终止低贡献的路径。
- enableNextEventEstimation: 显式采样光源以减少噪声。
- skyLightIntensity控制天空环境光的亮度, skyLightColor控制天空光的颜色。

这里提供两组参考配置:

- 质量速度平衡配置

```
samplesPerPixel = 32;      // 中等采样
maxBounces = 5;            // 标准深度
superSampleRate = 2;       // 基础抗锯齿
```

- 高质量渲染配置

```
samplesPerPixel = 128;     // 高采样
maxBounces = 8;            // 深度反弹
superSampleRate = 2;       // 抗锯齿
```

3 与Whitted-style Ray Tracing的效果对比

用一点点比喻来说, Whitted-style Ray Tracing 像是一位精准的工程师, 能用有限的资源清晰地勾勒出光线的几何投影, 适合对实时性要求高、风格化或只需硬阴影的场景。Path Tracing 则像一位追求极致的物理学家, 它模拟光的真实随机行走, 能产生无与伦比的真实感, 但其巨大的计算成本 (表现为初始渲染时的噪点) 是其实现的主要障碍。事实上也正是如此, Ray Tracing计算效率极高, 基本无噪点, 在一两分钟内就可以渲染出结果, 但是阴影硬, 边界锐利, 计算机图形感强, 干净但略显假。而Path Tracing恰恰相反, 软阴影, 边界模糊自然, 可以处理全局光照, 物理上更准确, 能逼近真实, 但需要高采样来实现平滑效果, 渲染一次少则5-10min, 多则30-60min。

如图, 左边是Ray Tracing的结果(参数默认为Sample Rate=5,Max Depth=15,下同), 右边为Path Tracing在高质量渲染配置下的结果。可以看出左图简洁、清晰、有锐利阴影, 右图有噪点、阴影柔和但质量略显粗糙, 主要由于采样数不够多, 但是阴影的柔和平滑效果已经远胜于左图。

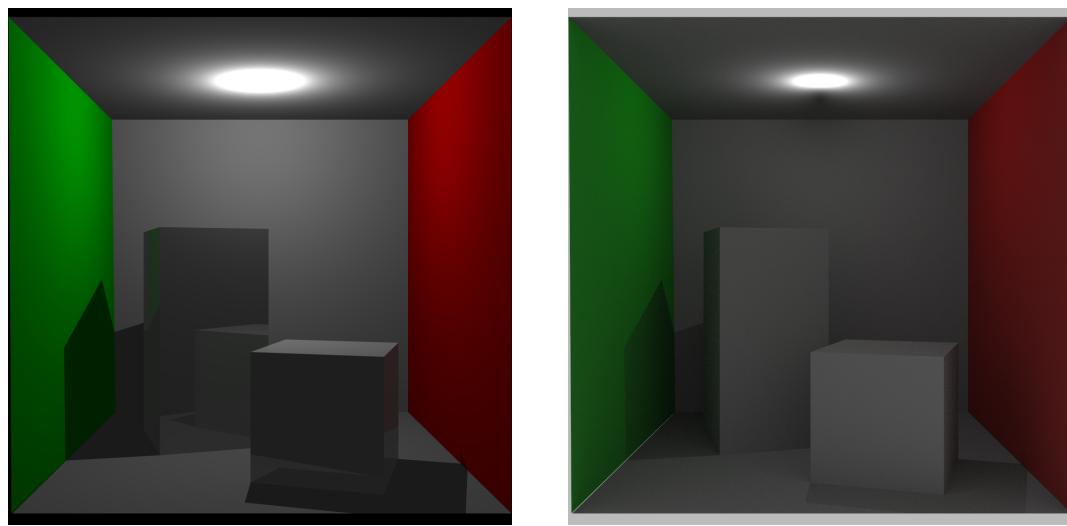


图 1: cornell_box的对比

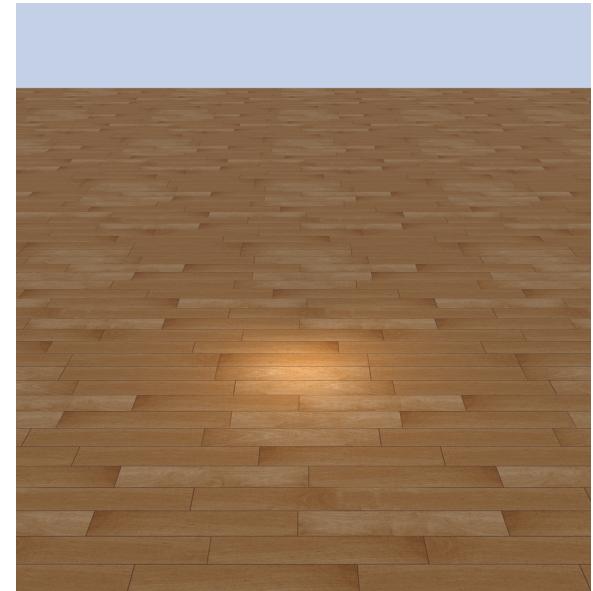
最后我们尝试把配置拉满，采用最高质量的参数追求一下理论最佳，当然渲染速度较为缓慢，本地跑cornell_box大概花了六七个半小时，跑floor大概需要1h，但是最终的效果十分不错。

```
samplesPerPixel = 512;
maxBounces = 20;
superSampleRate = 4;
```

下面是用Path Tracing渲染floor和cornell_box并与Ray Tracing对比的结果，由于其他图案实在耗时过长，笔者没有尝试，但理论上也能渲染出很好的效果。

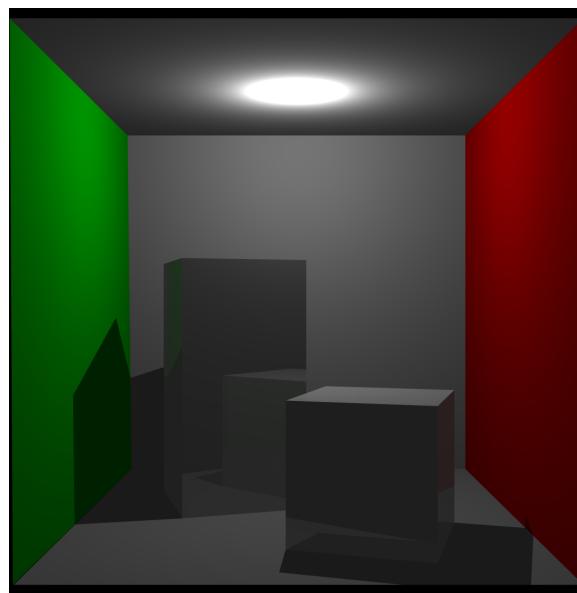


(a) Ray Tracing

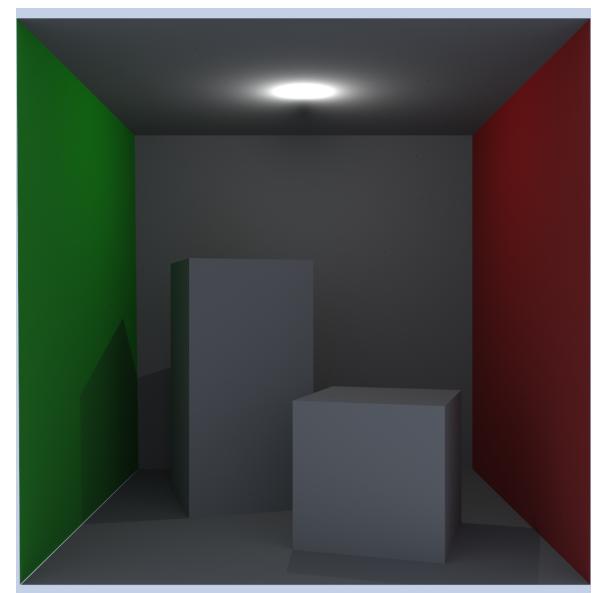


(b) Path Tracing

图 2: floor对比



(a) Ray Tracing



(b) Path Tracing

图 3: cornell_box对比