

基于物理的光线追踪渲染器

项目成员：王润泽

学号：PB20020480

序号：47

Abstract

本文要介绍了一种基于物理的光线追踪渲染器的实现方法。光线追踪作为一种计算机图形学渲染技术，可以生成高度真实的图像，模拟光线与物体之间的相互作用。文章将讨论光线追踪渲染器的原理、实现细节以及与自身优缺点。

1 引言

计算机图形学中有多种渲染技术，其中光线追踪和光栅化渲染器是两种常用的方法。光线追踪渲染器主要通过模拟光源、物体和相机之间的光线传播来生成真实感图像，适用于高质量的静态图像和动画渲染。与光栅化渲染对比，光线追踪渲染器能够生成高质量的真实感图像，特别是在处理阴影、反射、折射和全局光照等效果时具有优势。

本项目是基于物理的渲染方程实现的光线追踪渲染器，主要贡献包括：

- 处理不同的几何，不同的材质，实现了贴图渲染；
- BVH 光线追踪与物体求交加速；
- OpenMP 并行加速；
- Importance Sampling, NEE, Mixture Sample, MIS 采样方法提高质量。

2 原理

光线追踪渲染器的核心原理是模拟光线在场景中的传播过程。从相机发出的光线与场景中的物体相交，计算交点处的光照信息，包括直接光照和间接光照。直接光照由光源直接照射到物体表面产生，间接光照则是光线在物体表面反射、折射或透射后产生的。

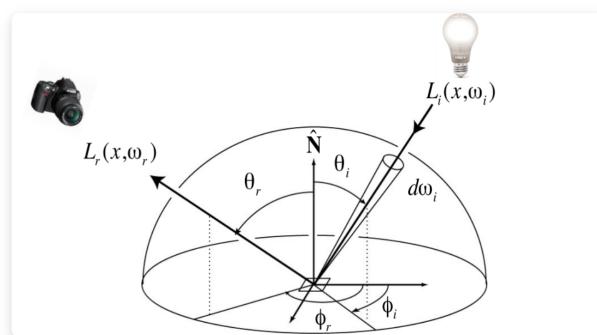


Figure 1: Rendering

用基于物理的直觉和数学的公式化表达，有渲染方程：

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2(\mathbf{n}(\mathbf{p}))} f_r(\mathbf{p}, \omega_i, \omega_o) L_i(\mathbf{p}, \omega_i) \cos \theta_{\omega_i, \mathbf{n}(\mathbf{p})} d\omega_i \quad (1)$$

其中

- L_o 是出射 radiance, L_e 是自发光 radiance, L_i 是入射 radiance

- \mathbf{p} 是渲染点
- ω_i 是入射光方向, ω_o 是出射光方向, $\mathbf{n}(\mathbf{p})$ 为 \mathbf{p} 处法向
- $\theta_{\omega_i, \mathbf{n}(\mathbf{p})}$ 是 ω_i 与 $\mathbf{n}(\mathbf{p})$ 的夹角
- $\mathcal{H}^2(\mathbf{n}(\mathbf{p}))$ 是法向 $\mathbf{n}(\mathbf{p})$ 所在半球
- f_r 是双向散射分布函数 (bidirectional scattering distribution function , BRDF)

记反射方程为

$$L_r(\mathbf{p}, \omega_o) = \int_{\mathcal{H}^2(\mathbf{n}(\mathbf{p}))} f_r(\mathbf{p}, \omega_i, \omega_o) L_i(\mathbf{p}, \omega_i) \cos \theta_{\omega_i, \mathbf{n}(\mathbf{p})} d\omega_i \quad (2)$$

则可以将渲染方程改写为:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + L_r(\mathbf{p}, \omega_o) \quad (3)$$

由于 $L_e(\mathbf{p}, \omega_o)$ 是物体自发光强度, 可以作为已知量, 我们假设除了光源外, 其他物体不会自发光, 所以只要重点关注反射光强 $L_r(\mathbf{p}, \omega_o)$ 即可

$$L_o(\mathbf{p}, \omega_o) = L_e + L_r(\mathbf{p}, \omega_o) \quad (4)$$

在 (2) 式中, 反射光 $L_r(\mathbf{p}, \omega_o)$ 实际上是来自两部分:

- 来自光源的直接入射, 称为**直接光**, 记作 $L_{\text{dir}}(\mathbf{p}, \omega_o)$;
- 来自其他物体反射的间接光, 称作**间接光**, 记作 $L_{\text{indir}}(\mathbf{p}, \omega_o)$

$$L_o = L_e + L_{\text{dir}}(\mathbf{p}, \omega_o) + L_{\text{indir}}(\mathbf{p}, \omega_o) \quad (5)$$

上式中 $-\omega_i$ 是 L_{dir} 和 L_{indir} 的出射方向

3 渲染器的实现

基于物理的光线追踪渲染器的实现可以分为以下几个关键部分:

- **场景描述**: 定义场景中的物体、光源、相机等元素, 以及它们的几何、材质和光照属性。
- **光线追踪**: 计算从相机发出的光线与场景中物体的交点, 以及交点处的法向量、纹理坐标等信息。
- **光照计算**: 在交点处计算直接光照和间接光照, 使用光照模型如 BRDF 等。
- **加速技术**: 使用空间分割 (如 BVH、KD-Tree) 、多线程技术 (如 OpenMP) 和蒙特卡洛重要性积分 (Monte Carlo) 提高渲染速度

3.1 场景描述

3.1.1 光线

构建出光线类 `ray.h` :

$$\mathbf{P}(t) = \mathbf{A} + t\mathbf{b} \quad (6)$$

其中 \mathbf{A} 是相机的位置, \mathbf{b} 是光线的方向, $\mathbf{P}(t)$ 是光线在世界坐标系下的一个直射点

而光线追踪主要涉及到:

- 计算光线方向: 从相机到像素方向
- 确定哪个物体是相交的
- 计算出相交点的像素

3.1.2 物体求交

在项目中主要实现了由球体求交 `sphere.h`、矩形求交 `rec.h`、三角形求交 `triangle.h` 等，在代码具体实现时，都抽象成了一个 `class hittable` 类：

```
class hittable {
public:
    //给定一条光线，判断是否与该物体相交，若相交，将修改hit_record
    //t_min和t_max是光线可以遍历的参数范围，只判断在这个范围内的交点
    virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec) const = 0;
    //返回物体的包围盒 output_box. bool值表示是否有包围盒 比如无限大平面就没有
    virtual bool bounding_box(double time0, double time1, aabb& output_box) const = 0;
    //它表示从物体表面采样的概率密度函数
    virtual double pdf_value(const pointf3& o, const vecf3& v) const {
        return 0.0;
    }
    //输入光线的起点，返回从该点出发到物体表面采样点的方向
    virtual vecf3 random(const vecf3& o) const {
        return vecf3(1, 0, 0);
    }
};
```

下面介绍一下球体求交和三角形求交的方法

球体

假设空间中的球满足参数 (\mathbf{C}, r) , \mathbf{C} 是位置, r 是半径, 设交点为 $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ 球体求交就是常规的二元一次方程方程组求解:

$$(\mathbf{P}(t) - \mathbf{C}) \cdot (\mathbf{P}(t) - \mathbf{C}) = r^2 \quad (7)$$

$$t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0 \quad (8)$$

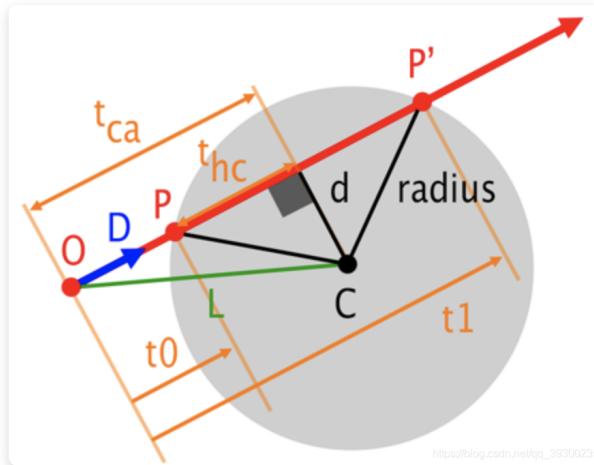


Figure 2; Sphere

解出 t , 即可获得交点位置.

三角形

三角形平面内任意点 P 可由三角形三个顶点用重心坐标表示：

$$P = wA + uB + vC \quad (9)$$

根据重心坐标性质 $w + u + v = 1$

$$P = (1 - u - v)A + uB + vC = A + u(B - A) + v(C - A) \quad (10)$$

光线表示为： $\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$ ，得到：

$$\begin{aligned} O + tD &= A + u(B - A) + v(C - A) \\ O - A &= -tD + u(B - A) + v(C - A) \end{aligned} \quad (11)$$

写成矩阵则是

$$[-D, \quad (B - A), \quad (C - A)] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A \quad (12)$$

解出线性方程组即可获得交点坐标和重心坐标 t, u, v, w

3.1.3 材料

物体的材料通常具有的属性包括：颜色吸收系数，光线散射分布，折射或镜面反射等，在代码实现中抽象成了 `class material` 类

```
class material {
public:
    // 材料本身具有自发光的属性，可以认为是一种特殊的纹理
    virtual color emitted(
        const ray &r_in, const hit_record &rec, double u, double v, const pointf3
&p) const {
        return color(0, 0, 0);
    }
    // 不同的材料对光线有不同的散射方程和颜色吸收系数，二者组成了BRDF
    virtual bool scatter(
        const ray &r_in,
        const hit_record &rec, // 击中点
        scatter_record &srec // 散射记录
    ) const {
        return false;
    }
    // 返回该材质散射光方向的概率密度函数，代表了BRDF的一项
    virtual double scattering_pdf(
        const ray &r_in,
        const hit_record &rec,
        const ray &scattered) const {
        return 0;
    }
};
```

在本项目中逐个实现了漫反射材质、金属材质、镜面反射、折射、以及体渲染，下面简单介绍一下

漫反射

漫反射分布主要是大多材料的主要特点，在采样时往往是按照 $\cos \theta$ 分布来采样的，主要是因为漫反射材料的 $BRDF$ 是常数，考虑渲染方程中的 $\cos \theta$ 项后，渲染结果会更加准确

金属材质

金属材质是表面比较光泽的材质，在代码实现时，可以认为近似认为镜面反射加上一些微弱的散射分布

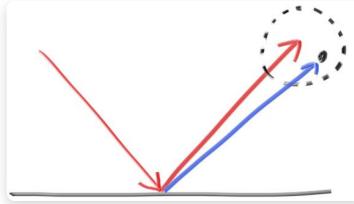


Figure 3. 金属

Dielectric 材质

对于类似于玻璃球，三棱镜等由明显的折射现象的材质，在渲染时不仅要考虑到材料的反射，还要考虑折射，由折射方程很容易得到光线方向：

$$\eta \cdot \sin \theta = \eta' \cdot \sin \theta' \quad (13)$$

$$\begin{cases} \mathbf{R}'_{\perp} = \frac{\eta'}{\eta} (\mathbf{R} + \cos \theta \mathbf{n}) = \frac{\eta'}{\eta} (\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n}) \mathbf{n}) \\ \mathbf{R}'_{\parallel} = -\sqrt{1 - |\mathbf{R}'_{\perp}|^2} n \end{cases} \quad (14)$$

在代码实现时要注意全反射的问题

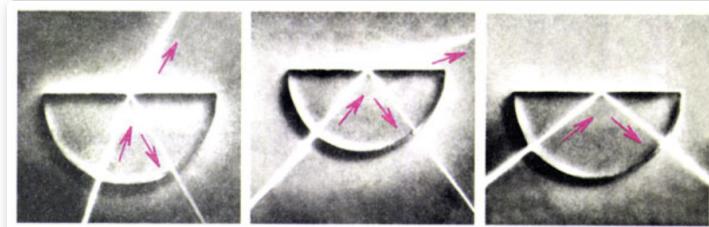


Figure 4. 折射

体渲染

体渲染也叫做参与介质渲染，往往用来渲染云雾、皮肤、水果等透明可穿透的物质，代码采取的方法是均匀各向同性的凸物体，主要包括：

- 随机选择体积内前进距离，距离选择符合指数衰减分布， σ 可以粗略认为是体密度，即密度高，每次行进步伐越短：

$$x \sim \exp(-\sigma x) \quad (15)$$

- 随机选择散射方向，即方向分布为均匀球面的分布

$$p(w_i) = \frac{1}{4\pi} \quad (16)$$

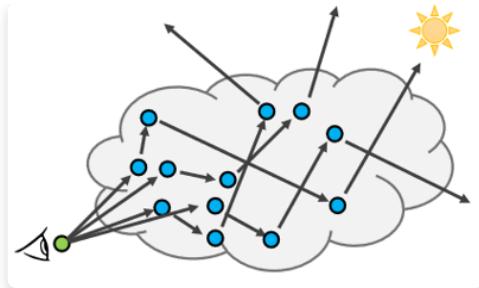


Figure 5. 体渲染

3.2 光线追踪求交

在初始进行搭建渲染框架时，采取物体求交判断的方法是逐个遍历

```

for (const auto& object : objects) {
    if (object->hit(r, t_min, closest_so_far, temp_rec)) {
        hit_anything = true;
        closest_so_far = temp_rec.t;
        rec = temp_rec;
    }
}

```

当物体很少时，不会对渲染造成什么影响，但当场景中的数量倍增很多时，逐个遍历的策略往往是难以接受的 $O(n)$ 。

考虑到这是对固定场景下的重复搜索过程，我们可以对场景中的物体进行排序，有了次序量的统计，我们就可以利用二分查找来将搜索的时间复杂度降至 $O(\log n)$

BVH 加速求交

基本思想：将场景中的几何体划分为一系列嵌套的边界体积（Bounding Volume，例如包围盒或包围球）。这些边界体积按照层次结构进行组织，形成一个树状结构。树的每个节点代表一个边界体积，包含该边界体积内的所有几何体。叶子节点包含实际的几何体或者几何体的一部分。

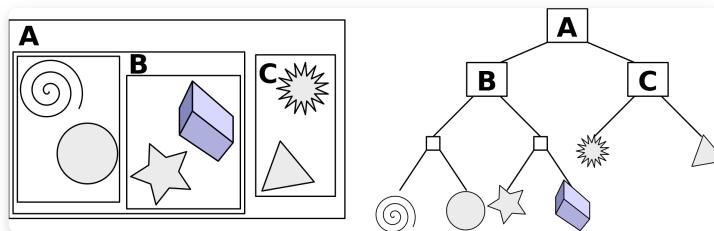


Figure 6. BVH

AABB: 常见的边界组织方式是 *Aligned Bounding Boxes*，一种简单的包围盒类型，用于表示一个物体或场景中一组物体的边界范围。AABB是与坐标轴对齐的矩形包围盒，这意味着它的边沿与坐标轴平行。因此，AABB可以在每个轴上用两个端点表示，即最小点 (x_{min} , y_{min} , z_{min}) 和最大点 (x_{max} , y_{max} , z_{max})。有时写成 (x_0, y_0, z_0) , (x_1, y_1, z_1)

判断盒子(slab)与光线是否相交十分简单，已知光线方程为: $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ ，对 x 轴有

$$t_0 = \frac{x_0 - A_x}{b_x}, \quad t_1 = \frac{x_1 - A_x}{b_x} \quad (17)$$

y, z 轴同理，如果与盒子有交点，那么很自然的存在重叠，如下图：

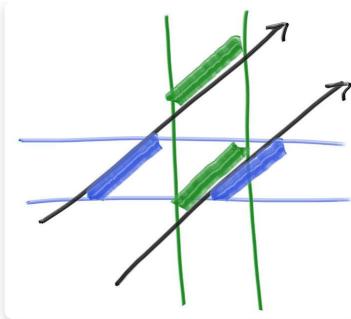


Figure 6. AABB求交

求交过程：当执行光线与场景的相交测试时，BVH可以显著提高性能。首先，从根节点开始，测试光线是否与边界体积相交。如果没有相交，那么光线与该边界体积内的所有几何体都不相交，不需要进一步测试。如果相交，继续检查该节点的子节点，直到到达叶子节点。这样，只有与光线可能相交的边界体积内的几何体才会被测试，从而显著减少了相交测试的次数，提高了光线追踪的性能。

```
bool bvh_node::hit(const ray& r, double t_min, double t_max, hit_record& rec) const{
    //中序遍历
    //如果当前结点的包围盒没有被击中，直接返回false，避免无效的搜索
    if(!box.hit(r, t_min, t_max)) return false;
    bool hit_left = left->hit(r, t_min, t_max, rec);
    bool hit_right = right->hit(r, t_min, hit_left ? rec.t : t_max, rec);
    return hit_left || hit_right;
}
```

值得一提的是： BVH树的构造时间复杂度在 $O(n \log n)$ ，但由于光线求交很频繁，构造一个树的分摊时间复杂度就很低落。

OpenMP并行加速

不仅物体求交是光线追踪渲染中频繁的操作，对每个像素点渲染也是一个大量密集的过程，考虑求出每个像素点的过程是类似的，可以利用 OpenMP实现并行加速的效果

```
std::vector<color> img(width * height, color(0, 0, 0));
#pragma omp parallel for shared(img)
for (k = 0; k < width * height; k++) {
    j = k / width; i = k % height;
    pixel_color = color(0, 0, 0);
    for (s = 0; s < spp; s += 1) {
        u = (i + random_double()) / (width - 1);
        v = (j + random_double()) / (height - 1);
        r = scene.cam->get_ray(u, v);
        pixel_color += ray_color(r, method);
    }
    img[k] = pixel_color;
}
```

3.3 光照计算

让我们再来看一下散射方程

$$L_o(w_o) = L_e + \int_{\Omega} f(w_i, w_o, x) L_i(w_i) \cos \theta d\omega_i \quad (18)$$

对于后面的积分项，在三维空间中采取离散化的 Riemann 求和方法消耗的时间代价非常高，会有维度爆炸的问题。所有，在光线追踪渲染器中，往往使用的是 Monte Carlo 重要积分法求解渲染方程

重要性采样积分分为以下步骤：

1. 假设存在一个被积函数 $f(x)$ 在定义域 $[a, b]$ 之间

$$I = \int_a^b f(x) dx \quad (19)$$

2. 选择一个非负的概率密度函数 PDF $p(x)$ ，定义在 $[a, b]$ 上

3. 对 $\frac{f(r)}{p(r)}$ 作期望的平均， r_i 是服从 $p(r)$ 的概率分布采样得到的

$$\frac{1}{N} \sum_i^N \frac{f(r_i)}{p(r_i)} \quad (20)$$

根据大数定理，无论我们选择什么样的概率密度函数 PDF p ，最终都能收敛到正确的结果

但是：如果 p 的形状越接近常数，那么收敛的会越快，即 f/p 越接近常值函数，方差越小，收敛速度越快

$$error = \sqrt{\frac{var(f/p)}{N}} \quad (21)$$

所以选择出一个好的概率分布函数，往往能决定一个光线追踪渲染器的质量

3.3.1 Scatter Sample

我们以最简单的漫反射为例，漫反射的 BRDF 方程为常数，观察积分式(18)，对待求积分式，我们往往很难预先知晓入射光强的分布信息，最简单的假设是入射光都是均匀分布入射的，所以我们的采样为

$$p(w_o) = \cos \theta / \pi \quad (22)$$

按照这个思路，只要能够递归的找到入射光强大小，那么既可以计算出像素的渲染结果

```
color ray_color(ray_in, scene, depth-1){  
    get scene intersection x, normal n  
    Le = w_Le * material->emitt(x, -w)  
    if depth==0 || x.isLight()  
        return Le  
    pdf,scatter_ray = sample from BRDF  
    return Le +  
        x.BRDF(r_in,scatter_ray)*(r_in.dir*n)*  
        ray_color(scatter_ray, scene, depth-1) / pdf;  
}
```

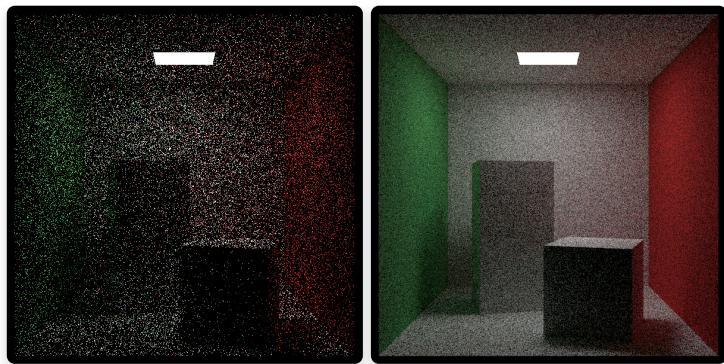


Figure 7. BRDF Sample, left 8spp,right 256spp

3.3.2 Light Sample

朴素的想法是,如果我们已经知道了光源的分布,那么光线大概也是从光源的方向来的,那么我们也可以估计 L_i 的分布,以更大的概率找到正确的光路

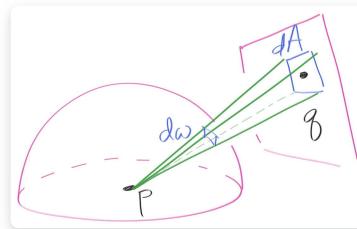


Figure 8. Light Sample

设 **pdf** 为 $p(w_i)$, 那么按角度采样和按光源表面积采样的概率只是变量代换的关系

$$p(w_i)dw_i = p(A)dA \quad (23)$$

$$p(w_i) \frac{dA \cdot \cos \alpha}{distance^2(p, q)} = \frac{1}{S}dA \quad (24)$$

得到:

$$p(w_i) = \frac{distance^2(p, q)}{\cos \alpha \cdot S} \quad (25)$$

```
color ray_color(ray_in, scene, depth-1){
    get scene intersection x, normal n
    Le = w_Le * material->emitt(x, -w)
    if depth==0 || x.isLight()
        return Le
    //pdf,scatter_ray = sample from BRDF
    pdf,shadow_ray = sample from light
    return Le +
        x.BRDF(r_in, shadow_ray)*(r_in.dir*n)*
        ray_color(shadow_ray, scene, depth-1) / pdf;
}
```

结果:

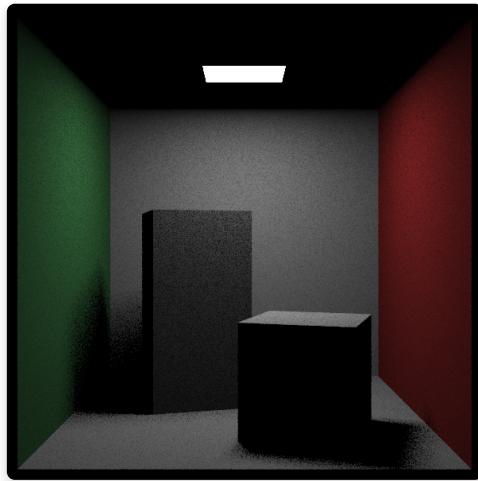


Figure 9. light Sample, 8spp

虽然图像噪点减少，但是从这张图片也反应了一些问题，对于来自墙壁的间接反射光没有很好的展示，图像背向光则不再有颜色产生

3.3.3 Mixture

既然间接光采样可以更好的展现图像反射光，而直接光采样可以很好去噪，提高寻找光路的效率

```
if (random_double() < 0.5)
    pick direction according to pdf_scatter
else
    pick direction according to pdf_light
```

得到结果如下：

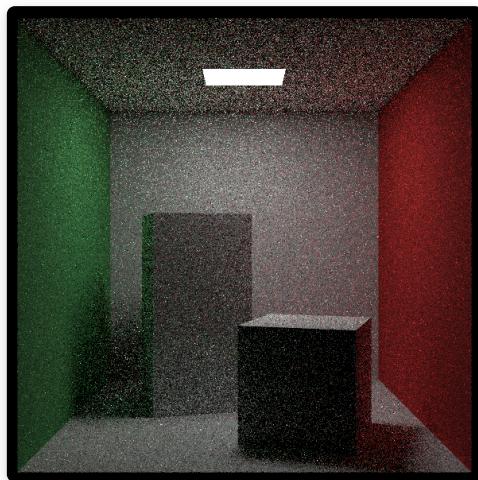


Figure 9. Mixture Sample, 8spp

很明显，对比BRDF，大大去噪；同时也能展现出间接光的效果

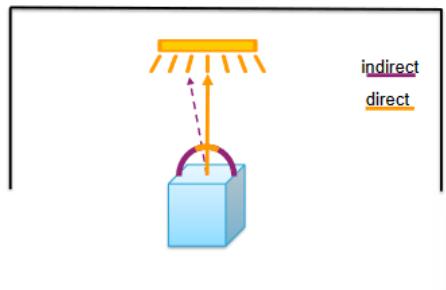
3.3.4 NEE

除了很平凡的将两种方法混合，NEE（Next Event Estimation）也是一种优化技术，用于减少渲染过程中的噪声并提高效率。NEE 通过直接采样光源来更准确地估计场景中的直接光照。这种方法有助于减少渲染中的方差，从而导致更快的收敛和更清晰的图像。

但NEE与直接光源采样不同的是，它是沿着物体表面散射方向不断采样直接光源，当直接光源采样被其他物体遮挡时，则不采样，通过这种方法使得算法有更高的概率找到光源，从而减少噪声和提高渲染效率

```
color ray_color(ray_in, scene, depth+1, is_shdow){  
    get scene intersection x, normal n  
    Le = w_Le * material->emitt(x,-w)  
    if x.isLight(){  
        if(depth==0 || is_shdow)  
            return Le  
    else  
        return 0;  
    }  
  
    //间接光  
    sct_pdf,scatter_ray = sample from BRDF  
    indirect = x.BRDF(r_in,scatter_ray)*(r_in.dir*n)*  
              ray_color(scatter_ray, scene, depth+1,fasle) / pdf;  
    //直接光  
    lgt_pdf,shadow_ray = sample from light  
    dir = x.BRDF(r_in,shadow_ray)*(r_in.dir*n)*  
          ray_color(shadow_ray, scene, depth+1,ture) / pdf;  
  
    return Le + indirect + dir;  
}
```

值得注意的是，对于NEE来说，因为直接光是 `shadow_ray` 是我们需要一直计算的，当最后一根光线 `scatter_ray` 也正好也达到光源时，如果此时把它的光强也算上，则对于最终结果来说，会多计算一根光线。



所以我们在代码实现时，要区分两种光线，避免出现光线过强的现象，渲染结果如下

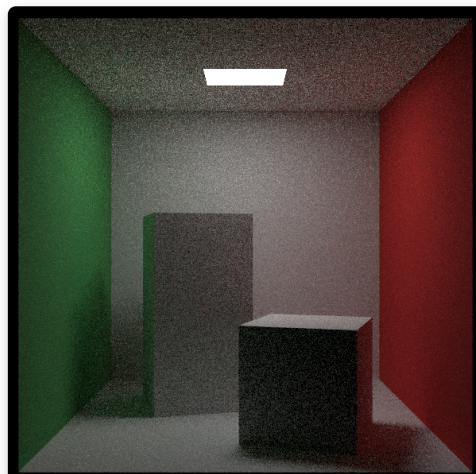


Figure 10. NEE Sample, 8spp

3.3.5 Multiple Importance Sampling

那如果当场景比较复杂时，只是优先采样直接光并不是一个好的测量，比如当光源比较大时，采样间接光，能够更加准确的渲染出图像

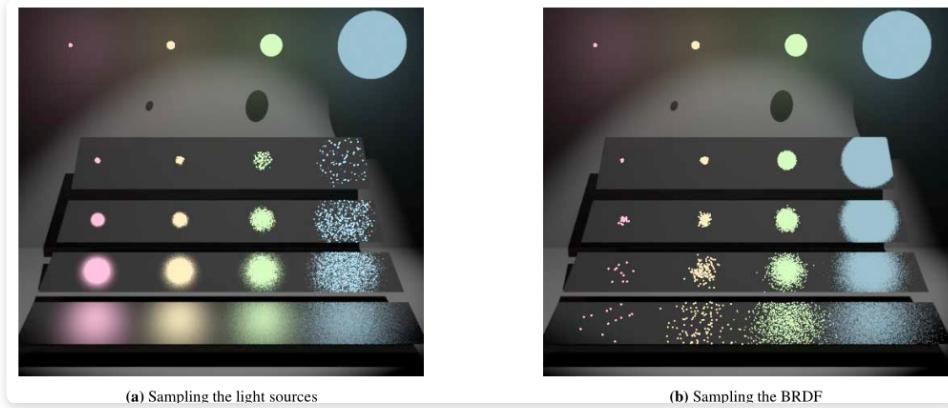


Figure 11. Compare

如何才能更好将二者的优势结合，根据 *Multiple Importance Sampling* 算法，我们可以对两种采样方法适当的加权，使得它们在自己擅长的场景，优势更加凸显。

多重重要采样的思想正是如此，考虑两种函数 $f(x), g(x)$ ，待积分的值是

$$\int f(x)g(x)dx \quad (26)$$

由于我们往往有时只能分别得到 $f(x), g(x)$ 的其中一个分布函数，无法得到二者乘积，MIS方法则可以这样解决

$$F = \frac{1}{n} \sum_{i=1}^n \left(\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)} + \frac{f(Y_j)g(Y_j)w_g(Y_j)}{p_g(Y_j)} \right) \quad (27)$$

其中

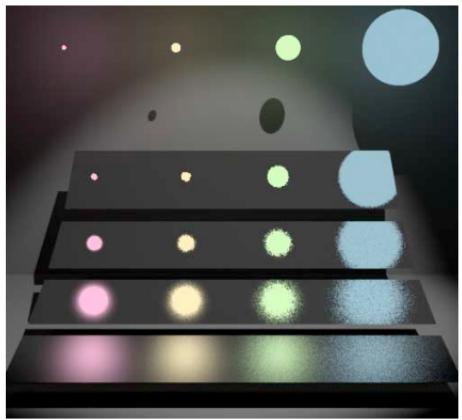
$$w_s(x) = \frac{(p_s(x))^\beta}{\sum_i (p_i(x))^\beta} \quad (28)$$

如果 $\beta = 1$

$$F = \frac{f(X)g(X)}{p_f(X) + p_g(X)} + \frac{f(Y)g(Y)}{p_f(Y) + p_g(Y)} \quad (29)$$

这样当 p_f 值很小， p_g 值则比较大，避免了大方差的引入，可以证明上面的方法是无偏的

根据上面方法，则可以很好的结合二者优势得到渲染图像质量更好



(c) A combination of samples from (a) and (b).

Figure 12. MIS

在我的代码框架中也同样实现了。

4 Result

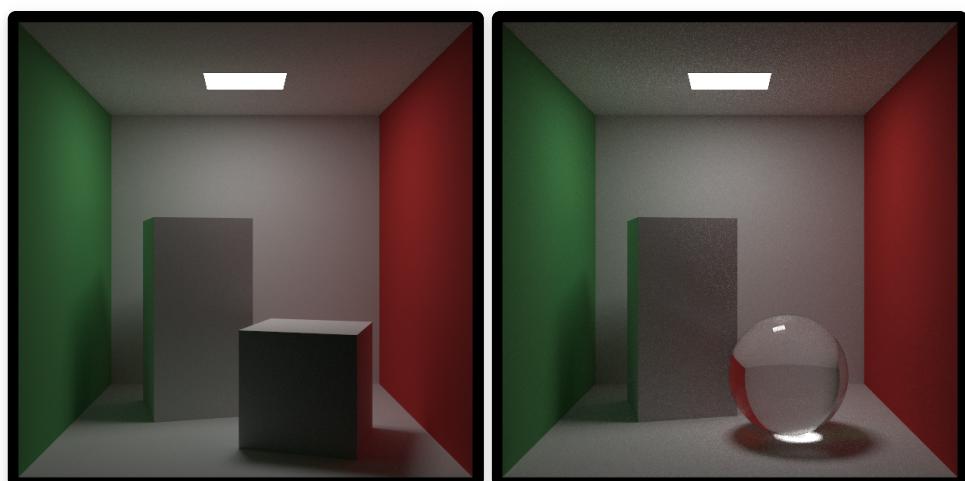


Figure 13. MIS: Cornell-box; Glassy Ball

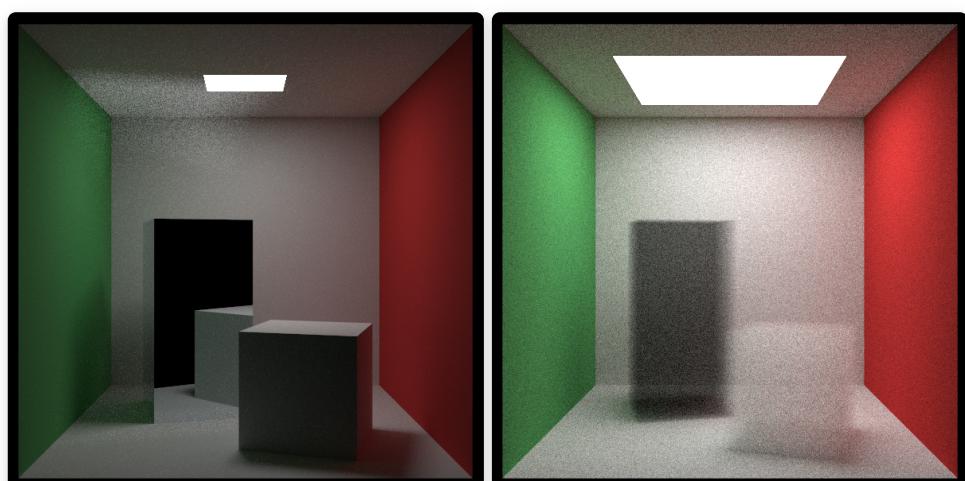


Figure 13. MIS: Mirror; Smoke

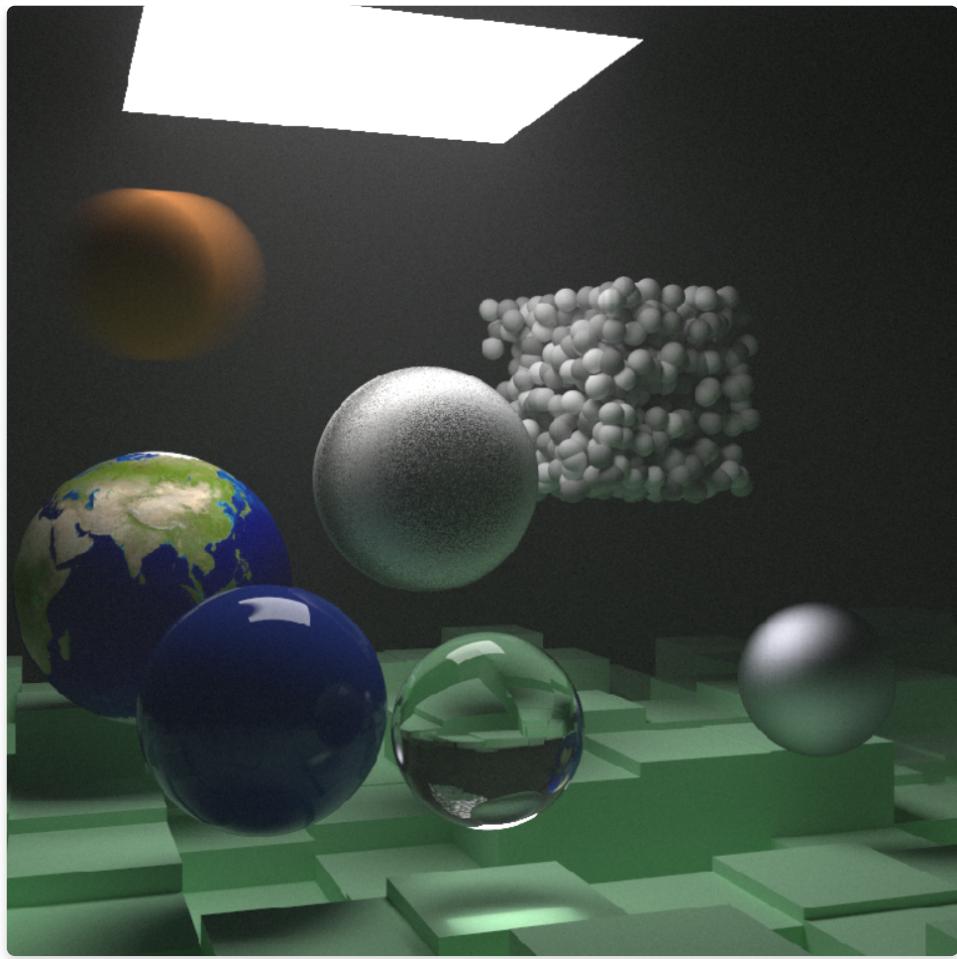


Figure 14. Everything

5 Summary

本报告介绍了基于物理的光线追踪渲染器的原理和实现，并与光栅化渲染器进行了对比分析。光线追踪渲染器在生成高质量真实感图像方面具有优势，但渲染速度相对较慢，所以产生了许许多多优化加速的方法。而对于光栅化渲染器则以渲染速度和实时应用为优势，但生成的图像质量相对较低。未来随着图形硬件性能的提升，实时光线追踪技术将有望在更多应用场景中发挥作用。