

目录

1	绪论	1
1.1	背景	1
1.2	光线追踪.....	2
2	Whitted-Style 光线追踪	3
2.1	Bling-Phong Model.....	3
2.2	Whitted-Style Ray tracing	3
3	加速结构	7
3.1	轴对齐包围盒	7
3.2	空间划分与物体划分	7
3.3	BVH 加速结构	8
4	路径追踪	11
4.1	渲染方程与 BRDF	11
4.2	蒙特卡洛路径追踪	12
5	代码实现	15
5.1	Whitted-Style 光线追踪	15
5.2	BVH 加速结构	16
5.3	蒙特卡洛路径追踪	17
6	参考文献	18

1 緒論

1.1 背景

渲染 (rendering) 是计算机图形学中的一项重要任务，计算机程序获取由排列在 3D 空间中的许多几何对象组成的场景，通过渲染的方法并计算显示从特定视点观察呈现的 2D 图像，完成从 3D 物体到像素的处理。

根据渲染技术的不同，可以将渲染归结为两类，一类是 object-order rendering 图 1.1，主要用到了光栅化技术，在渲染过程中依次考虑场景中的物体，将物体的各种信息计算并存储在 framebuffer 中。另一类是 image-order rendering 图 1.2，主要使用了光线追踪技术，在渲染流程中依次对每个像素进行着色。本文主要阅读总结并实现了基于光线追踪方法的一些渲染技术。

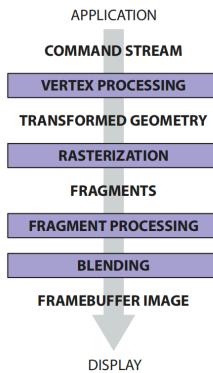


图 1.1 光栅化

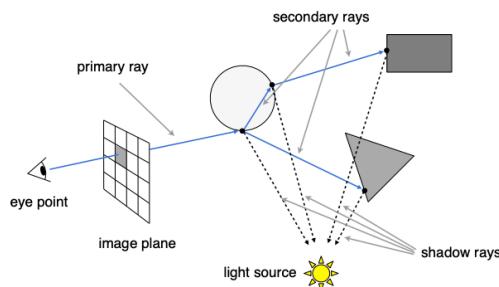


图 1.2 光线追踪

1.2 光线追踪

光线追踪程序的基本原理是从光源发出光子，光子传播并与物体表面或次表面产生作用，根据几何光学中的定理计算每个像素的着色，并最终在摄像机上进行成像，一种经典的方法是 Whitted 提出的全局光照模型^[1]。在光线追踪算法中，每个像素通过光子更好地获得全局光照效果。但光线追踪方法通常需要更大的计算量，因此主要用于离线渲染(Off-line rendering)，而光栅化方法主要用于实时渲染(On-line rendering)。

在光线追踪的计算中，一项重要的操作是光线和物体求交，在复杂的场景中这会带来巨大的计算量。因此，实际中都需要一些方法来加速求交的过程。空间加速结构是计算机图形学研究中的重要部分，常见的方法有 Oct-Tree^[2]，KD-Tree^[3]，BVH 加速结构^[4]。

whitted-style 光线追踪模型也依然无法准确计算出漫反射光线和直接光照的贡献，基于辐射度量学方法的渲染方程^[5]很好的解决了这一问题。进一步可以通过蒙特卡洛路径追踪算法递归解出渲染方程，从而获得更加真实的全局光照效果。

本文将阅读总结主要的光线追踪方法，从 Whitted-style 光线追踪开始，得到一个基本的光线追踪渲染模型，通过 BVH 加速结构加速光线与物体求交的过程，最后引入渲染方程和蒙特卡洛路径追踪算法，获得更真实的全局光照模型。

此外，本文还基于《GAMES101——现代计算机图形学入门》^[6] 的框架进行了算法实现，得到的实验结果和程序源码将通过附件形式上交。

2 Whitted-Style 光线追踪

2.1 Bling-Phong Model

Bling-Phong 反射模型基于 Lambert's cosine law^[7]，可以用于物体表面的着色。在该模型中，反射光的强度与表面法线和光源方向向量的点乘成正比，能够很好的模拟物体的漫反射表面。

公式，式 2-1 表示了 Bling-Phong 模型中物体表面的 *Intensity* 大小，由 I_a 环境光项， I_d 漫反射项， I_s 镜面反射项目构成。

$$I = I_a + I_d + I_s = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}'_j)^n \quad (2-1)$$

Bling-Phong 的模型假设每个光源都位于距离场景中的物体无限远的一点，因此，虽然该模型能够对物体表面的漫反射做出较好的模拟，但环境光照以及镜面反射的计算都不够精确，在复杂的场景中很难做出很好的全局光照效果，并且使用这一方法很难做出较好的软阴影效果。

2.2 Whitted-Style Ray tracing

在真实的场景中，物体表面的光照不仅仅需要考虑直接光源，光线打到场景中，经过多次弹射最终呈现在物体表面的光线更为真实，而这一物理现象无法用 Bling-Phong Model 进行描述。光线追踪正是为解决这种问题所提出的。

2.2.1 基本光学假设

在 Whitted-style 光线追踪算法中^[8]，需要先对光线做出一些基本假设，满足几何光学。

- 光线一定沿着直线传播
- 光线之间无法碰撞
- 光线路径可逆

2.2.2 算法描述

在满足了以上假设之后，可以将 Whitted-Style Ray tracing 的主要思想分为两个步骤^[8]。

- Ray Casting
- Recursive (Whitted-Style) Ray Tracing

Ray Casting 图 2.1，即从摄像机向投影进平面上的每一个像素点投射出一条光线，计算得到光线在真实世界坐标中的向量表示，判断光线与物体的交点，并通过计算是否呗其他物体遮挡，可以获得较为真实的阴影效果。在找到了着色点之后，通过 Recursive (Whitted-Style) Ray Tracing 的方法计算着色，实现全局光照效果。

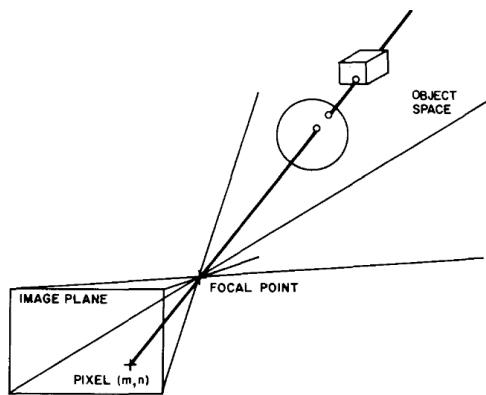


图 2.1 Ray Casting

Recursive (Whitted-Style) Ray Tracing 图 2.2，光线在场景中传播时，会经过多次反射和折射，反射光和折射光也可以发生二次反射和折射。因此光线追踪是一个递归过程，每一个点的颜色贡献来自于直接光照和间接反射光照与间接折射光照。对于每条从相机投射出的光线，找到在递归传播次数（程序中一般用 `max depth` 进行表示）内的所有着色点，通过递归方式对每个着色点的颜色进行累加求和，可以得到该像素点的最终着色。

2.2.3 着色模型

在 Whitted-Style 光线追踪算法中，通过式 2-2 进行着色， S 和 T 分别代表着色点带给观察视点的反射和折射分量。物体的反射率和入射角度与入射空间和物体的折射率有关，

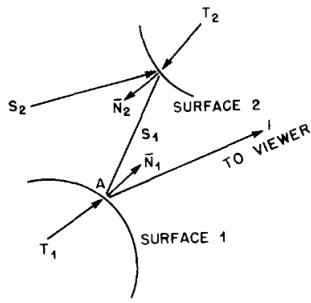


图 2.2 Recursive (Whitted-Style) Ray Tracing

可以通过 Fresnel Reflection law 计算得到，Schlick 也提出了近似的计算方法，大大降低了这一过程计算量。

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s S + k_t T \quad (2-2)$$

至此，已经基本介绍了 Whitted-Style Ray Tracing 中的主要内容，伪代码将在后续章节进一步介绍。

3 加速结构

上一节中提到，在光线追踪中，需要求出光线与物体的交点，如何快速求交从而提升程序运行的效率就是一个亟待解决的问题。在图形学中，大量的模型是通过三角形表示的，Tomas Möller 与 Ben Trumbore^[9] 提出了光线与三角形求交的算法。

然而，每一个像素上投射出的光线都要跟场景中的所有三角形进行求交操作，对于复杂的场景来说，这是非常大的运算量。

3.1 轴对齐包围盒

轴对齐包围盒 (AABB, Axis-Aligned Bounding Box) 图 3.1 由三个与坐标轴平行的对面构成，其加速思想非常简单，即某个物体的包围盒可以理解为空间中能包括它的最小长方体。如果投射出的光线无法与包围盒相交，则一定无法与物体相交。由此，在判断光线与物体相交的时候，可以先判断光线是否与包围盒相交，而光线与包围盒相交的计算量明显小于与三角形相交，因此可以简化掉很多计算。

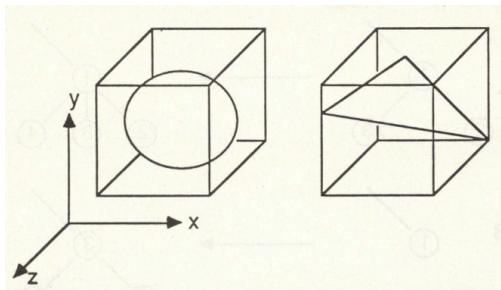


图 3.1 轴对齐包围盒 AABB

3.2 空间划分与物体划分

在计算机图形学中，空间划分（Spatial Partition）和物体划分（Object Partition）是常用的加速结构。

均匀空间划分 Uniform Spatial Partitions (Grids) 指的是将空间均匀地划分成较小的 grid，每一个 grid 构成一个 AABB 结构。在判断与物体相交的过程中，先判断是否与 grid 有交点，再根据 grid 中是否存储了三角面片进一步进行求交运算。这种划分方法，如何选择划

分方格的数目对算法效率提升很有帮助，一般会采用基于经验的方法。

非均匀的空间划分 Non-Uniform Spatial Partitions 经常使用 Oct-Tree , KD-Tree , BSP-Tree^{[3][2]}。使用这种方法时，每个物体可能会划分到多个空间里，从而造成大量的存储冗余，且断包围盒与三角面的是否相交很难，在编程中需要处理很多 special case。

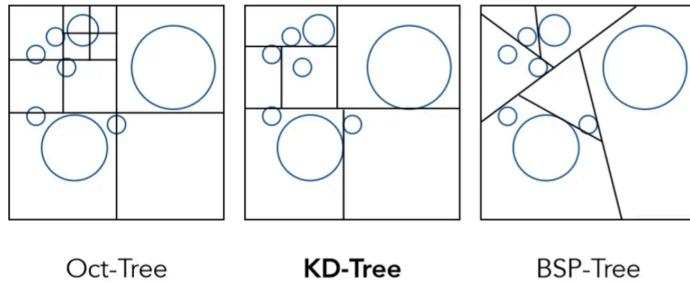


图 3.2 非均匀均匀空间划分

3.3 BVH 加速结构

BVH 加速结构采用了基于物体划分的方法图 3.3，从 object(空间中的对象) 角度出发，将三角形面片进行划分。在 BVH 结构中，每一个三角形面片只会存在于唯一的包围盒之内，从而在很大程度上节省了存储空间，也使得物体与光线的求交计算变得更加简单。

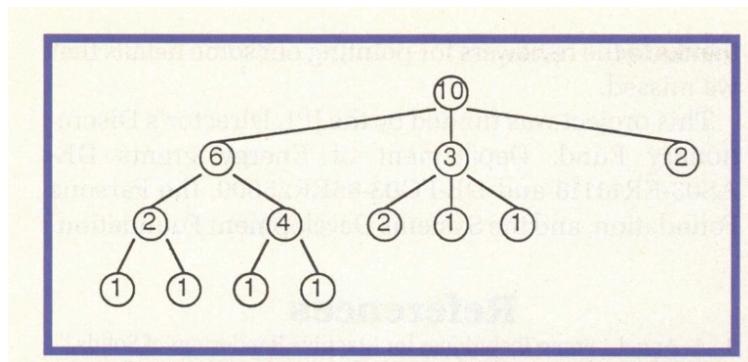


图 3.3 BVH 树

在建立 BVH 的过程中，可以通过最长轴或者中位数物体的思想，递归地对空间进行划分，终止条件可以为叶子结点中三角形面片的数量。

在 BVH 树中，只有叶子节点存储物体的信息。因此，在 BVH 树遍历的过程中，需要递归地对 BVH 节点进行遍历，若遍历到叶子结点时，再计算光线与叶子结点中三角形的交点。

将在后续章节中进一步介绍 BVH 编码实现的伪代码。

4 路径追踪

尽管 Whited-Style 光线追踪模型已经可以得到较为真实的全局光照模型，但这种模型没有对漫反射项进行追踪式 2-2，且在计算直接光照的贡献时，使用了 Blinn-Phong 模型，导致全局光照并不准确。

而路径追踪方法^[10] 基于渲染方程^[5]，实现基于物理的渲染（PBR，Physically Based Rendering），效果会更加真实。

4.1 渲染方程与 BRDF

这里先给出渲染方程的公式

$$L_r(p, w_r) = L_e(p, w_o) + \int_{\Omega} f_r(p, w_i, wo) L_i(p, w_i) n \cdot w_i dw_i \quad (4-1)$$

在渲染方程中，等式左边表示着色点的出射方向反射出的光照强度，即最终在摄像机中的成像。 $L_e(p, w_o)$ 则是物体的自发光项， $\int_{\Omega} f_r(p, w_i, wo) L_i(p, w_i) n \cdot w_i dw_i$ 表示从各个方向的光源对物体反射光的贡献， $L_i(p, w_i)$ 表示了入射光。

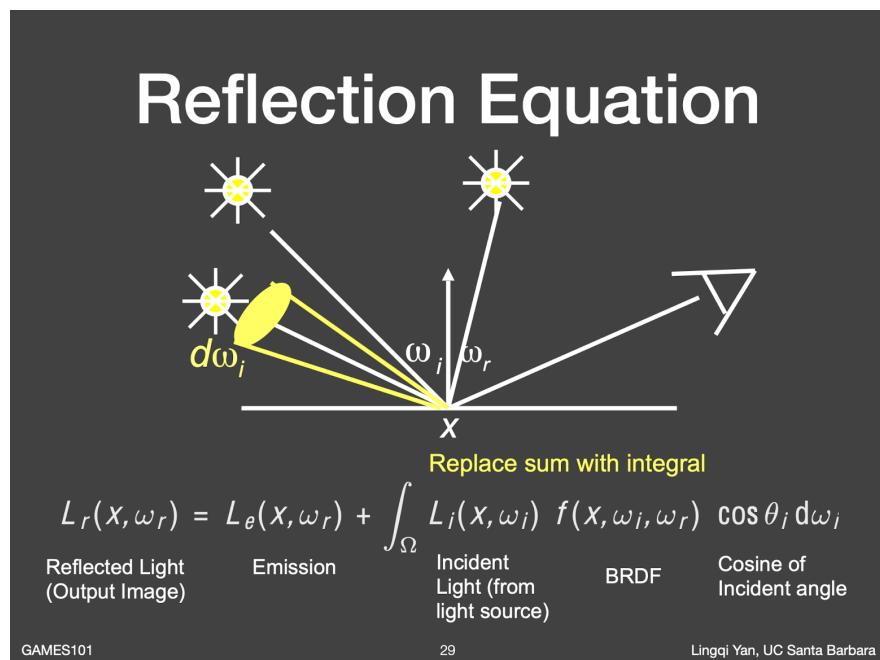


图 4.1 渲染方程

$f_r(p, w_i, wo)$ 表示了双向反射分布函数 (BRDF, Bidirectional Reflectance Distribution Function)，这一函数描述了物体的材质，可以简化地理解成，在某个着色点上，如何将所有的入射能量 w_i 进行累加（积分），并反射到出射方向 w_o 上。

如今已有很多的 BRDF 模型，最常用的是 Cook-Torrance BRDF 模型^[1]。

而在真实场景当中，不仅仅只有光源会对着色点产生贡献，光线在场景中的各种反射和折射光线都会进行累加，因此，渲染方程的求解是一个递归过程。因此式 4-1 是一个递归项。后续渲染方程也需要递归求解。

4.2 蒙特卡洛路径追踪

蒙特卡洛积分是指对函数值进行多次采样，并且求积分作为均值。在计算机程序中，对函数的积分进行离散化表示，可以得到：

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}, X_i \sim p(x) \quad (4-2)$$

因为在渲染方程中，对入射光线求和是一个积分，对其进行采样，将式 4-1 改写可以得到：

$$L_r(p, w_r) = L_e(p, w_o) + \frac{1}{N} \sum_{i=1}^N \frac{f_r(p, w_i, wo) L_i(p, w_i) n \cdot w_i}{p(w_i)} \quad (4-3)$$

另外，在采样的过程中，式 4-1 只是简单的在空间中进行了采样，这会导致光线采样的效率很低，很多投射出的光线可能根本无法集中光源。一种修改的方法是直接对光源区域进行采样图 4.2，改进后的渲染方程通过蒙特卡洛积分求解可以得到：

$$L_r(p, w_r) = L_e(p, w_o) + \frac{1}{N} \sum_{i=1}^N \frac{f_r(p, w_i, wo) L_i(p, w_i) \frac{\cos(\theta) \cos(\theta')}{\|x' - x\|^2}}{p(w_i)} \quad (4-4)$$

到此，就得到了一个初步的全局光照模型，但还有其他的问题没有解决。

一个问题是每一个着色点应该如何向光源采样投射光线，如果投射光线过多，很可能产生指数爆炸的问题。只投射一条光线的方法称为路径追踪，但路径追踪会带来很大的噪声，解决方法是每一个像素投射出多条光线。

Easy! Recall the alternative def. of solid angle:

Projected area on the unit sphere

$$d\omega = \frac{dA \cos \theta'}{\|x' - x\|^2}$$

(Note: θ' , not θ)

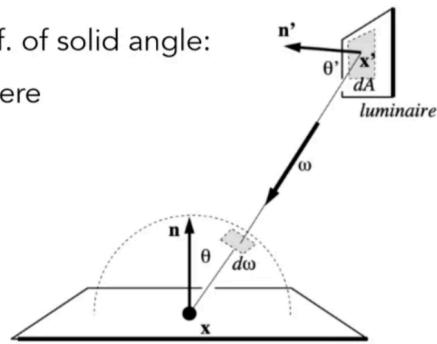


图 4.2 对光源区域进行采样

另一个需要解决的问题是，渲染方程的求解本质是一个递归的过程，如何设置光线停止弹射的条件。如果仅仅设置停止弹射的次数，很有可能无法产生良好的全局渲染效果。一种解决方案是采用俄罗斯轮盘赌（Russian Roulette）算法。即设定一个概率 P ，光线有 P 的概率递归下去，并返回 L_0/P ，有 $1 - P$ 的概率停止传播。通过这种方法，计算得到的光线强度期望是不变的：

$$E = P * (L_o)/P + (1 - P) * 0 = L_o \quad (4-5)$$

到此，就得到了一个正确的路径追踪算法，伪代码和具体实现将在后续章节进行介绍。

5 代码实现

本章的代码实现基于^[6] 提供的框架，实现了 Whitted-Style 光线追踪，BVH 加速结构和蒙特卡洛路径追踪。

5.1 Whitted-Style 光线追踪

Whitted-Style 光线追踪的伪代码和渲染结果如下：

```
//渲染过程 render主函数
vector3f eye_pos ;// 视点位置
for (int j = 0; j < scene.height; ++j)
{
    for (int i = 0; i < scene.width; ++i)
    {
        x = (2 * ((i + 0.5) / scene.width) - 1) * scale * imageAspectRatio;
        y = (1 - 2 * ((j + 0.5) / scene.height)) * scale;
        Vector3f dir = Vector3f(x, y, -1) - eye_pos;
        castRay(eye_pos, dir, scene, 0); //从视点位置投射出一根光线
        {
            // castRay函数 从着色点向场景中投射出一根反射光线，通过非尔涅定律进行着色计算
            // 1.trace一根光线，并求交点，获取交点信息
            trace(eye_pos, dir, get_objects()); //中间涉及到求交操作
            getSurfaceProperties()
            // 2.基于物体材质，使用非尔涅定律进行着色计算 递归光线追踪
            case(REFLECTION_AND_REFRACTION)
            {
            }
            case(REFLECTION)
            {
            }
            default //光源
            {
            }
        }
    }
}
```

图 5.1 Whitted-Style 光线追踪伪代码

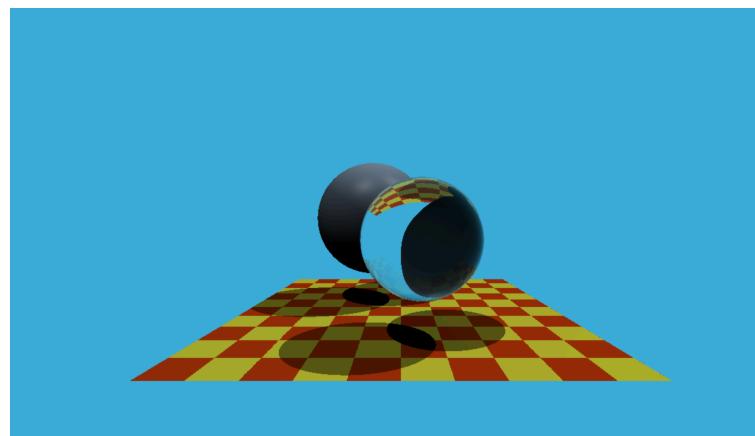


图 5.2 Whitted-Style 光线追踪渲染结果

5.2 BVH 加速结构

BVH 加速结构的伪代码和渲染结果如下：

```
Ray 的数据结构
struct Ray{
    //Destination = origin + t*direction
    Vector3f origin;
    Vector3f direction, direction_inv;
    double t;//transportation time,
    double t_min, t_max;
};

BVH结构与光线求交

Intersection BVHAccel::getIntersection(BVHBuildNode* node, const Ray& ray) const
{
    Intersection result;
    //如果没有和BVH求交，则一定不会和物体求交，直接返回减少计算量
    if (! (node->bounds.IntersectP(ray, ray.direction_inv, dirIsNeg)))//判断与bounding box的交点
    {
        return result;
    }
    //只在叶子节点存储物体
    if (node->left == nullptr && node->right == nullptr)
    {
        return node->object->getIntersection(ray);// 在判断与bounding box相交之后在判断是否与物体相交
    }
    //后序遍历过程
    Intersection inter_left = getIntersection(node->left, ray);
    Intersection inter_right = getIntersection(node->right, ray);
    return inter_left.distance < inter_right.distance ? inter_left : inter_right;
}
```

图 5.3 BVH 加速结构伪代码

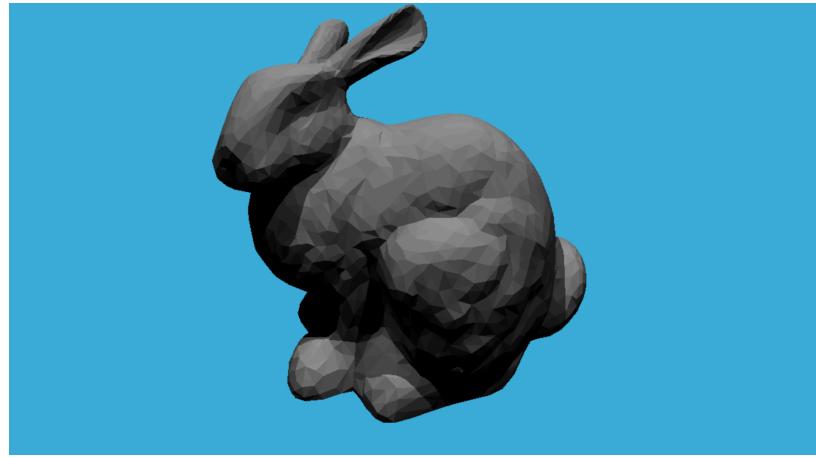


图 5.4 BVH 加速结构渲染结果

在渲染 bunny.obj 模型的时候，在使用同样参数的情况下，使用 BVH 加速结构的渲染速度为 9s，不是用加速结构渲染速度为 2049s，极大提升了渲染速度。

5.3 蒙特卡洛路径追踪

蒙特卡洛路径追踪的伪代码和渲染结果如下：

使用了 Cornell box 测试全局光照效果，因为使用 CPU 模拟 GPU 进行渲染，所以渲染速度较慢，设置的 SPP (samples per pixel) 为 4，所以渲染的结果可能会有一些噪声。

```
// 每个shadingpoint打一根光线 防止指数爆炸
// 但每个pixel进行多次采样 采样的变量为 render.cpp 中的spp
Vector3f Scene::castRay(const Ray& ray, int depth) const
{
    Intersection point_intersection = intersect(ray); //获得场景和物体的交点 object求交点
    // 1、special case
    //如果没打到物体，则返回背景颜色
    //如果打到了光源（光源是一种特殊的object），直接返回光源颜色
    if (point_intersection.m->hasEmission())
    {
        return point_intersection.m->getEmission();
    }
    if (!point_intersection.happened)
    {
        return Vector3f();
    }
    //定义直接光照和简介光照 并且在光源区域采样
    Vector3f L_dir, L_indir;
    Intersection light_intersection;
    float light_pdf;
    sampleLight(light_intersection, light_pdf); // 在光源区域进行采样 打一根光线
    Ray sample_ray(p, ws);
    Intersection check = intersect(sample_ray); //用于判断是直接光照还是间接光照

    // 2, 计算直接光照
    if ((check.distance - distance) > -EPSILON) // 直接光照
    {
        //m->eval 为brdf函数
        L_dir = emit * m->eval(wo, ws, N) * dotProduct(N, ws) * dotProduct(-ws, NN) * pow(distance, 2) / light_pdf;
    }
    // 3, 计算简介光照 使用RR算法
    float random_pro = get_random_float();
    if (random_pro < RussianRoulette)
    {
        return L_dir;// 无法继续打光线
    }

    // 在材质的表面进行采样 sample一根反射光线
    Vector3f wi = (m->sample(wo, N)).normalized();
    Ray reflect_ray(p, wi);
    Intersection i = intersect(reflect_ray); // 简介光照打到的位置
    if (i.happened && !i.m->hasEmission()) // 与不是光源的物体相交（获得反射光）
    {
        L_indir = castRay(reflect_ray, depth + 1) * m->eval(wo, wi, N) * dotProduct(wi, N) / m->pdf(wo, wi, N) / RussianRoulette;
    }
    //4. 返回直接光照与间接光照 求和
    return L_dir + L_indir;
}
```

图 5.5 蒙特卡洛路径追踪伪代码

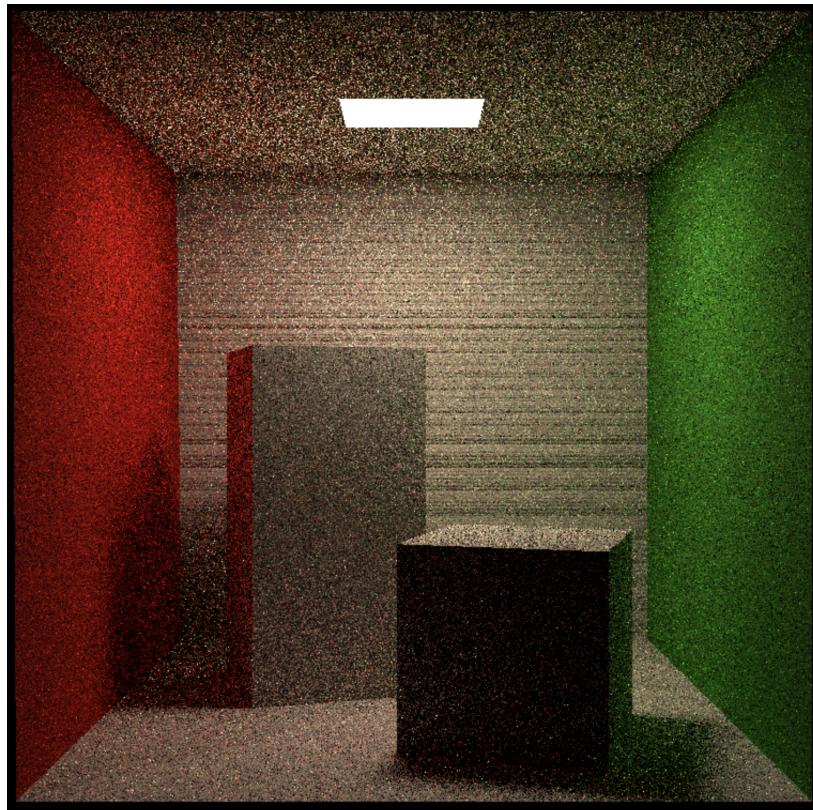


图 5.6 蒙特卡洛路径追踪渲染结果

6 参考文献

- [1] WHITTED T. An improved illumination model for shaded display[G]//ACM Siggraph 2005 Courses. [S.l. : s.n.], 2005: 4-es.
- [2] GLASSNER A S. Space subdivision for fast ray tracing[J]. IEEE Computer Graphics and applications, 1984, 4(10): 15-24.
- [3] BENTLEY J L. Multidimensional binary search trees used for associative searching[J]. Communications of the ACM, 1975, 18(9): 509-517.
- [4] GOLDSMITH J, SALMON J. Automatic creation of object hierarchies for ray tracing[J]. IEEE Computer Graphics and Applications, 1987, 7(5): 14-20.
- [5] KAJIYA J T. The rendering equation[C]//Proceedings of the 13th annual conference on Computer graphics and interactive techniques. [S.l. : s.n.], 1986: 143-150.
- [6] 闫令琪. 现代计算机图形学入门[EB/OL]. 2020. <https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html>.
- [7] PHONG B T. Illumination for computer generated pictures[J]. Communications of the ACM, 1975, 18(6): 311-317.
- [8] 孙小磊. 计算机图形学十二：Whitted-Style 光线追踪原理详解及实现细节[EB/OL]. 2021. <https://zhuanlan.zhihu.com/p/144403005>.
- [9] MÖLLER T, TRUMBORE B. Fast, minimum storage ray/triangle intersection[G]//ACM SIGGRAPH 2005 Courses. [S.l. : s.n.], 2005: 7-es.
- [10] LAFORTUNE E P, WILLEMS Y D. Bi-directional path tracing[J]., 1993.
- [11] LearnOpenGL[EB/OL]. <https://github.com/JoeyDeVries/LearnOpenGL>.