南开大学 Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机图形学实验报告

基于 nrenderer 框架的光子映射渲染方法

姓名：汪晨

年级：2020 级

专业：计算机科学与技术

2023 年 6 月 11 日

摘要

本实验基于提供的 nrenderer 渲染框架实现了一个使用光子映射方法的插件。

关键字：计算机图形学; 光子映射;

目录

一、 实验目的

主要工作内容为使用 nrenderer 框架实现一个插件，采用光子映射渲染方法对给定的模型文件进行渲染。

二、 算法简介

光子映射算法 (photon mapping) 在处理 SDS 光路和 caustic 效果时表现要优于路径追踪等渲染方法。光子映射算法包含两个阶段，第一个阶段为光子追踪，即从光源处发射光子到整个场景中，根据碰撞表面材质的不同发生相应的变化，如光子在漫反射表面被部分吸收并记录，在玻璃表面得到反射和折射，最终通过漫反射表面的记录得到整个场景的 photon map。第二个阶段为光线追踪，相当于从镜头发出光线，在光线弹射时根据第一阶段形成的 photon map 统计光子密度，从而计算光照。渲染方程如下:

$$L(x,\omega) = \int L_i(x,\omega_i) f_r(x,\omega_i \to \omega) \cos\theta d\omega_i$$

由于无法得到解析解，可通过蒙特卡洛积分近似计算

$$L(x,\omega) = \sum \frac{\Phi_p(x_i,\omega) f_r(x,\omega_i \to \omega)}{\pi r^2}$$

三、 具体实现

(一) photon trace

由 light 发射 n 个起点随机，方向随机的光子，光子与 object 发生碰撞则遇到反射或折射或吸收，若遇到漫反射表面 (包含 diffuse 相关属性) 就将该光子的能量、位置、入射方向等信息记录到 photon map 中，之后改变光子起点和方向后继续发射。递归进行 trace，直到 n 个光子均被漫反射表面吸收，photon map 完成。

photon trace

```
void MyRenderer::photontrace(Ray& ray, Vec3 r, int currDepth)
{
    auto [hitone, islight] = closestHit(ray, false);
    if (!hitone)
        return;
    else
    {
        auto mtlHandle = hitone->material;
        auto scattered = shaderPrograms[mtlHandle.index()]->shade(ray, hitone
            ->hitPoint, hitone->normal);
        auto attenu = scattered.attenuation;
        auto emitted = scattered.emitted;
        Ray next = scattered.ray;
        if (scene.materials[mtlHandle.index()].hasProperty("diffuseColor"))
        {
            photon p;
            p.position = hitone->hitPoint;
```

```
            p.r = r;
            p.in = ray;
            p.out = next;
            p.norm = hitone->normal;
            photoncontainer.push_back(p);
            if (currDepth >= depth || rand() % 1000 < 200) return;
            float cos_0 = abs(glm::dot(hitone->normal, ray.direction));
            photontrace(next, attenu * r * cos_0 / (scattered.pdf * 0.8),
                currDepth + 1);
        }
        else if (scene.materials[mtlHandle.index()].hasProperty("ior"))
        {
            if (currDepth >= depth || rand() % 1000 < 200) return;
            photontrace(next, attenu * r / (scattered.pdf * 0.8), currDepth +
                 1);
            next = scattered.refraction_ray;
            photontrace(next, scattered.refraction * r / (scattered.pdf * 0.8
                f), currDepth + 1);
        }
    }
}
```

(二) ray trace

镜头处的每个像素都射出 n 条射线进行追踪，若光线与 object 碰撞，则使用碰撞点周围特定数量光子的信息来计算光照，本实验选取至多 20 个光子进行计算。

ray trace

```
RGB MyRenderer::trace(const Ray& r, int currDepth) {
    //if (currDepth == depth) return scene.ambient.constant;
    auto [HitOne, islight] = closestHit(r);
    if (HitOne) {
        if (islight)
        {
            return HitOne->normal;
        }
        else
        {
            auto mtlHandle = HitOne->material;
            auto scattered = shaderPrograms[mtlHandle.index()]->shade(r,
                HitOne->hitPoint, HitOne->normal);
            auto scatteredRay = scattered.ray;
            auto attenuation = scattered.attenuation;
            auto emitted = scattered.emitted;
            float dotpara = glm::dot(HitOne->normal, scatteredRay.direction);
            float pdf = scattered.pdf;
            float  disrecord = 0.f;
            Vec3 dir{ 0,0,0 };
```

```
            Vec3 sumreco{ 0,0,0 };
            if (scene.materials[mtlHandle.index()].hasProperty("ior")) {
                RGB next = Vec3(0.0f);
                //to control the optical path iength
                if (currDepth > depth && rand() % 1000 < 200) next = (
                    attenuation + scattered.refraction) * scene.ambient.
                    constant;
                else {
                    auto reflex = trace(scattered.ray, currDepth + 1);
                    RGB refraction = Vec3(0.f);
                    if (scattered.refraction_ray.direction != Vec3(0.f))
                        refraction = trace(scattered.refraction_ray,
                            currDepth + 1);
                    next = attenuation * reflex + refraction * scattered.
                        refraction;
                }
                return emitted + next / (scattered.pdf * 0.8f);
            }
            nowcal = HitOne->hitPoint;
            auto tempvec = findneighbor(disrecord, 20);
            for (int i = 0; i < tempvec.size(); i++)
            {
                auto p = photoncontainer[i];
                float temp2 = glm::dot(-(p.in.direction), HitOne->normal);
                if (temp2 <= 0.f) continue;
                dir += -(p.in.direction);
                sumreco += p.r / (float)(PI * disrecord * disrecord *
                    rendernum);
            }
            float dotans = glm::dot(HitOne->normal, glm::normalize(dir));
            return emitted + attenuation * sumreco * dotans / pdf;
        }
    }
    else
    {
        return Vec3{ 0 };
    }
}
```

为了防止 trace 时间过长，在限制弹射次数（深度）的同时使用俄罗斯轮盘的形式按一定概率吸收光子。值得一提的是，光子追踪算法与路径追踪算法之间具有一定相似性，部分内容可参考框架中的 simple_path_tracing 项目进行实现。

四、 实验结果

刚开始实验时生成的光子同质化较严重，导致时常会出现第二阶段计算光照半径为 0，图片全黑的现象，后对半径的最小值进行限制，取 10^{-3}，从表面上解决了问题，但根本原因尚未明了。

可能是由于第二阶段 ray trace 采样光子每次选取最近光子时都将整个序列重排，加上设备

性能不足，渲染时间被延长了许多，使用 2000 个光子渲染 path_tracing_cornel.scn 时约用时 1h 5min，同时像素点的着色也存在问题。

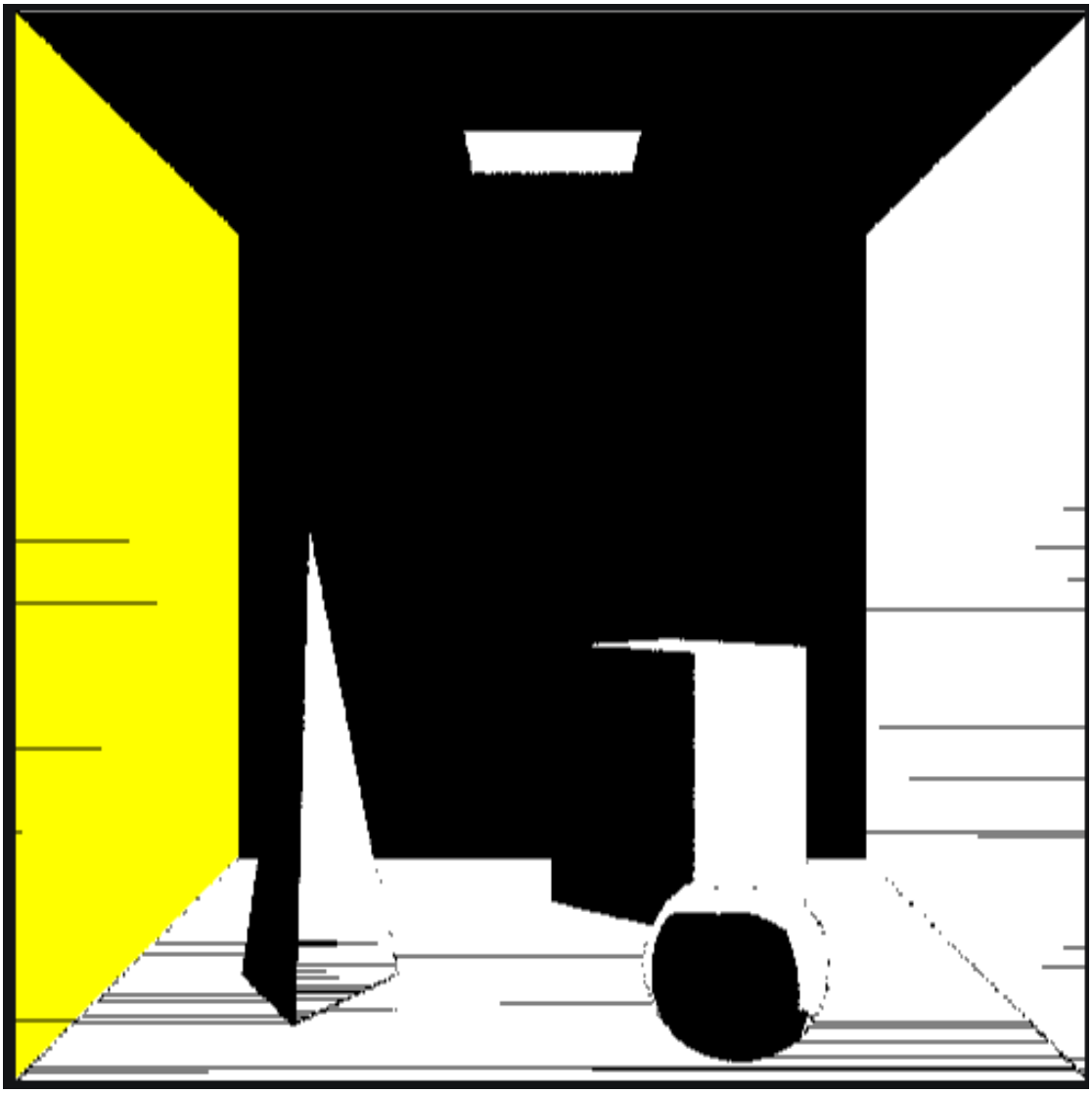

图 1: 渲染输出

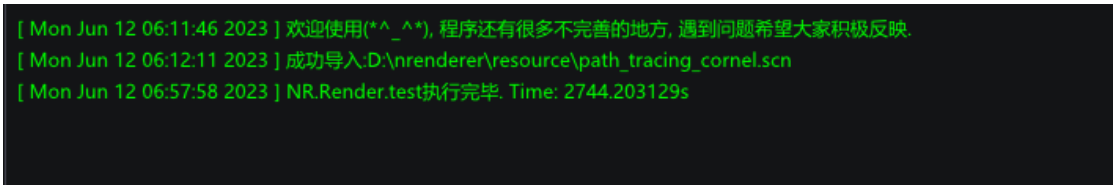

图 2: 用时

五、 实验感想

实验工作量和难度均超出预想之外。渲染框架做得相当优秀，遗憾的是框架代码阅读难度颇高，刚开始写插件也因此无从下手，同时由于本科期间对图形学的相关内容接触甚少，对相关知

识的检索和了解耗时反而超过了写代码的时间，但通过本次实验更深入了解了图形学的相关知识，更学习了解了几种不同渲染算法的大体原理，总体而言收获颇丰。

六、 参考资料

实验过程中主要的参考资料包含以下几项:

1. 往年优秀作业
2. 光子映射系列
3. 各种 ray 与不同类型 object 碰撞的博客