Final Project: Report

赵赫扬 2018011419

2021 年 1 月 10 日

目录

1 基本框架 **2**

2 光线追踪算法: Ray Tracing + PPM **2**

2.1 概述 2

2.2 对直接光照的优化与软阴影 5

2.3 焦散效果 5

2.4 openmp 5

3 光线追踪算法：path tracing **6**

4 抗锯齿 **6**

5 kd-tree 求交加速 **7**

6 平面贴图 **8**

7 景深 **8**

8 对复杂 obj 文件的支持 **9**

8.1 顶点法向插值 9

8.2 纹理映射 9

9 自制场景 **10**

1 基本框架

在这次 project 中延用了 PA1 的基本框架，在 code_pt 或 code_ppm 中运行 **bash run.sh** 即可在对应的 bin/文件夹下得到可执行的 render 程序。

code_pt 是基于 path tracing 实现的渲染器，核心部分在 main.cpp 中的 radiance 函数，可通过 openmp 在路径追踪时加速。通过将 1 个 pixel 分为 4 个 subpixel 进行采样实现了抗锯齿。由于 path tracing 基于多次采样，自带软阴影的效果。除此以外，扩展了 perspective camera 类，支持在 txt 中设置焦距，模拟景深效果。

code_ppm 是基于 ppm 实现的渲染器，光子贴图 (Photon Map) 相关的数据结构详见 photon.hpp，而 photon tracing 函数，与最后渲染使用的 ray tracing（同样使用了 openmp 加速）代码均在 radiance.hpp 中。在 Mesh 中加入了 kd 树求交优化并且完成了对 mesh 的顶点法向插值与纹理映射 (详见 mesh.hpp 与 mesh.cpp)。在 plane 中支持延任意方向，以任何比例在任何位置插入贴图。通过将 1 个 pixel 分为 4 个 subpixel 进行采样实现了抗锯齿。通过对面光源多次采样实现了软阴影。扩展了 perspective camera 类以模拟景深效果。

本文中用到的场景配置文件均可在 testcases 文件夹中找到，平面贴图均在 map 文件夹下，纹理贴图与对应 obj 模型均在 mesh 文件夹下。最后的渲染好的成品单独放在了外面的 pictures 文件夹下。

2 光线追踪算法: Ray Tracing + PPM

2.1 概述

渐进光子映射是一种基于光子映射的高效渲染算法。光子映射的基本思路分为两步：

首先通过从各个光源随机发射大量构建光子贴图。具体实现方式如下：发射光子时，维护光子的能量，方向，位置信息，与场景求交，遇到物体时：

1. 以一定的概率被吸收（根据材质进行俄罗斯轮盘赌），然后从光源重新发射下个光子。

2. 若未被吸收且该表面为漫反射面，将光子的信息记录到光子贴图中。

3. 根据材质，采样光子经过材质后的能量与方向，迭代，直到光子被吸收或迭代深度过大。

本 project 中 photon 的结构如下：

```
class Photon : public Ray {
public:
    Photon() : Ray(Vector3f::ZERO, Vector3f::ZERO) {
        this -> power = Vector3f::ZERO;
    }
```

```
    Photon(const Vector3f& orig, const Vector3f& dir, const Vector3f& power)
      : Ray(orig, dir) {
        this -> power = power;
    }
    Photon(const Photon& r) : Ray(r) {
        power = r.power;
    }

    const Vector3f &getPower() const {
        return power;
    }

    void setPower(const Vector3f& power) {
        this -> power = power;
    }

    void setOrigin(const Vector3f& origin) {
        this -> origin = origin;
    }

    void setDir(const Vector3f& dir) {
        this -> direction = dir;
    }
private:
    Vector3f power;
};
```

Listing 1: Photon

第二步，对每个像素点进行光线追踪。与经典的 ray tracing 基本一致，区别就是在光线追踪到漫反射面时根据光子贴图近似计算辐射度。近似的理论分析如下

$$
\begin{aligned}
L_r(x,\vec{\omega}) &= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}') |\vec{n_x}\cdot\vec{\omega}'| \mathrm{d}\omega_i' \\
&= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) \frac{\mathrm{d}^2\Phi_i(x,\vec{\omega}')}{\cos\theta_i \mathrm{d}\omega_i' \mathrm{d}A_i} |\vec{n_x}\cdot\vec{\omega}'| \mathrm{d}\omega_i'
\end{aligned}
$$

其中 f 为 BRDF 函数。可以用存好的光子贴图中邻近的光子来对通量 Φ 做近似，则

$$
\begin{aligned}
L_r(x,\vec{\omega}) &\approx \sum_{p=1}^{n} f_r(x,\vec{\omega_p},\vec{\omega}) \frac{\Delta\Phi_p(x,\vec{\omega_p})}{\Delta A} \\
&\approx \frac{1}{\pi r^2} \sum_{p=1}^{n} f_r(x,\vec{\omega_p},\vec{\omega}) \Delta\Phi_p(x,\vec{\omega_p})
\end{aligned}
$$

近似式的计算涉及大点集的 k 近邻计算，需要对 Photon Map 建立 kd 树来加速近邻点的快速查找。具体实现如下

```
void buildPhotonTree(PhotonTree* node, Photon* head, int left_ind, int right_ind) {
    node -> cut = head[left_ind];
    node -> minX = std::min_element(head + left_ind, head + right_ind + 1, cmp_x) -> getOrigin().x();
    node -> maxX = std::max_element(head + left_ind, head + right_ind + 1, cmp_x) -> getOrigin().x();
    node -> minY = std::min_element(head + left_ind, head + right_ind + 1, cmp_y) -> getOrigin().y();
    node -> maxY = std::max_element(head + left_ind, head + right_ind + 1, cmp_y) -> getOrigin().y();
    node -> minZ = std::min_element(head + left_ind, head + right_ind + 1, cmp_z) -> getOrigin().z();
    node -> maxZ = std::max_element(head + left_ind, head + right_ind + 1, cmp_z) -> getOrigin().z();
    if (left_ind == right_ind) {
        return ;
    }
    int mid = (left_ind + right_ind) / 2;
    if (node -> dim == 0)
        std::nth_element(head + left_ind, head + mid, head + right_ind + 1, cmp_x);
    else if (node -> dim == 1)
        std::nth_element(head + left_ind, head + mid, head + right_ind + 1, cmp_y);
    else
        std::nth_element(head + left_ind, head + mid, head + right_ind + 1, cmp_z);

    node -> right = new PhotonTree();
    node -> right -> dim = (node -> dim + 4) % 3;
    node -> cut = head[mid];
    buildPhotonTree(node -> right, head, mid + 1, right_ind);
    if (mid > left_ind) {
        node -> left = new PhotonTree();
        node -> left -> dim = (node -> dim + 4) % 3;
        buildPhotonTree(node -> left, head, left_ind, mid - 1);
    }
}
```

Listing 2: PhotonTree

其中 $minX, maxX, minY, maxY, minZ, maxZ$ 维护了一个子树的包围盒，便于搜索时快速剪枝。在搜索时，维护一个 heap 来存当前查到的前 k 个最近点。

而在光子映射的基础上，每次适当缩小近似中的 k 与检索范围半径 r，即可实现 PPM。本 project 中设置的迭代缩小比例为定值 α，并在半径变小时灵活调整增加的采样光子数。每次迭代与之前的结果求指数平均。

2.2 对直接光照的优化与软阴影

考虑到光直接打到漫反射面的辐射度可以由 $\int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}') cos\theta_i \mathrm{d}\omega_i'$ 对每个光源直接求和算出。点光源可以通过光强直接计算，而面光源需要在面上进行多次均匀随机采样近似直接光照的辐射度，以达到软阴影的效果。具体代码实现在 light.hpp 中。

而考虑单独计算直接光照后，对应的，在构建光子贴图中只存储第二次及以上碰到物体的光子，这样既可以避免重复计算，也让更多的光子被用在模拟 color bleeding 与焦散等光学效果上，提高渲染的效率。

2.3 焦散效果

当较近的光源照到表面比较复杂的透明物体时，会发生“漫折射”，这种现实中的光学效果可以通过 PPM 算法模拟出来。图 1、图 2 是不同位置点光源照到玻璃杯的光学效果的模拟。

2.4 openmp

在第二步时加入语句

```
#pragma omp parallel for schedule(dynamic, 1) private(t_finalColor)
    num_threads(NUM_threads)
```

Listing 3: openmp

以支持多核并行，并用 NUM_threads 控制线程数。

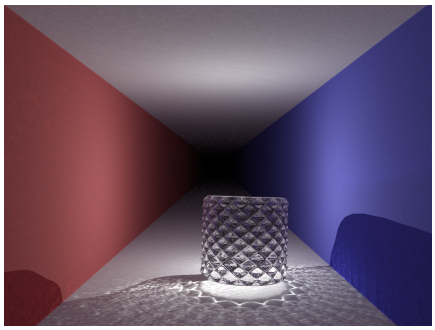

图 1: caustic-1

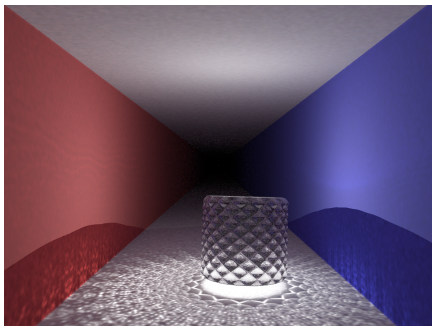

图 2: caustic-2

3 光线追踪算法：path tracing

此 project 同样参考 smallpt 实现了 path tracing 算法,具体参见 code_pt/main.cpp 中 radiance 函数的实现。

但是考虑到渲染时间有限，采样数有限数，path tracing 的结果会带有大量噪点，在最终场景的渲染中并未使用 path tracing 算法。

4 抗锯齿

将每个像素点分为 4 个 subpixel，在 subpixel 中加入经过 filter 的随机抖动，并将结果平均实现抗锯齿效果。

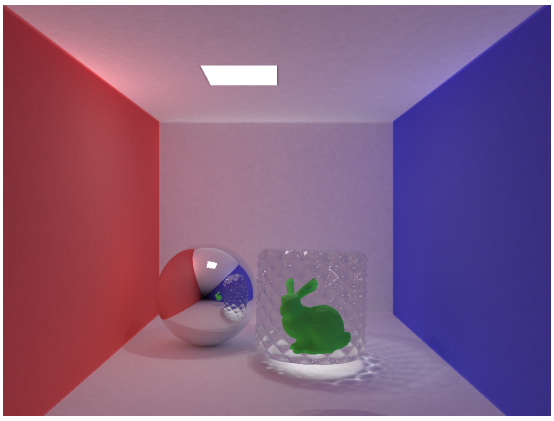

图 3: rabbit

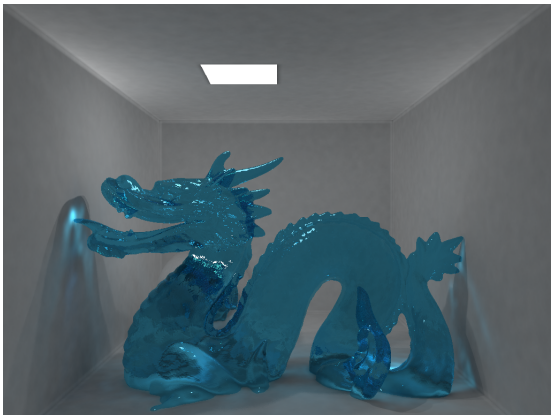

图 4: dragon

5 kd-tree 求交加速

对于 mesh 求交使用 kd-tree 加速。建树方式与上文光子图的情形类似，在此不在赘述，但是注意到此处取小三角面片的一点的某维坐标作为优先级划分，两个子树没有共同的三角面片，但是空间有交。每次求交时，用射线是否与子树的包围盒有交，以及源点与包围盒的最短距离是否超过 hit 已经计算出的 t 来剪枝。

具体判断射线与子树的包围盒是否有交，按照课上讲的方式，可以对每个 Slab 计算其两个平面与光线的交点，并将这两个交点按照其参数 t 的大小排序，记为：t_i^{min}, t_i^{max}。有交当且仅当 $\max(t_0^{min}, t_1^{min}, t_2^{min}) < \min(t_0^{max}, t_1^{max}, t_2^{max})$。

图 3，图 4 是对较复杂 obj 模型的渲染。

6 平面贴图

支持平面上 ppm(p6) 类型的贴图。

在 plane.hpp 中改写了 plane 的变量与求交函数，并且修改 sceneparser，使得可以在设计场景时支持追加图片名称，水平缩放向量 axis_x，竖直缩放向量 axis_y，校正值 offset_x，offset_y。

则求到交点时对应到图片上的像素点 (x, y) 可由下列代码给出。

```
int x = (int) (Vector3f::dot(axis_x, r.pointAtParameter(t_intersect)) - offset_x);
int y = (int) (Vector3f::dot(axis_y, r.pointAtParameter(t_intersect)) - offset_y);
```

Listing 4: image

使用 Image 类中实现好的 LoadPPM 函数来解析图片。

7 景深

在场景文件中输入 camera 的参数时，可选择输入焦距信息。改写了 perspective camera 的 generateRay 函数：若配置了焦距，则先求出原射线与焦平面的焦点 P，（设相机坐标为 C）然后在 C 上加入随机抖动得到 C'，返回射线 C'P。

```
Vector3f Rc = Vector3f(point - Vector2f((double) width / 2, (double) height / 2), distance);
Matrix3f R = Matrix3f(horizontal, up, direction);
Ray r = Ray(center, R * Rc);
Material m = Material(Vector3f::ZERO);
Plane p_focal = Plane(direction.normalized(),
    Vector3f::dot(direction.normalized(), center) + focal_distance, &m);
Hit hit = Hit();
p_focal.intersect(r, hit, -1e8);
Vector3f f_point = r.pointAtParameter(hit.getT());
Vector3f orig = center;
double p1 = 1.,p2 = 1.;
while(p1 * p1 + p2 * p2 > radius * radius) {
    if (Xi == nullptr) p1 = erand48(this -> Xi) * 2. - 1.;
    else p1 = erand48(Xi) * 2. - 1.;
    if (Xi == nullptr) p2 = erand48(this -> Xi) * 2. - 1.;
    else p2 = erand48(Xi) * 2. - 1.;
}
orig = center + p1 * horizontal + p2 * up;
Ray blurred_r = Ray(orig, f_point - orig);
```

```
return blurred_r;
```

Listing 5: DOF

8 对复杂 obj 文件的支持

改写了 Mesh 的 constructor 使其能够正确解读 obj 文件中的顶点法向信息和纹理信息。具体参照 mesh.hpp 和 mesh.cpp 的改动。

8.1 顶点法向插值

用面积占比计算出交点在三角面片上的重心坐标，基于重心坐标对三个法向量进行加权平均，即得该点法向量的近似值。以下为计算重心坐标的代码。

```
Vector3f computeCoordinate(Vector3f& v1, Vector3f& v2, Vector3f& v3,
    Vector3f& v) {
    double totalS = Vector3f::cross(v1 - v2, v1 - v3).length() / 2.;
    double total1 = Vector3f::cross(v - v2, v - v3).length() / 2.;
    double total2 = Vector3f::cross(v - v1, v - v3).length() / 2.;
    double total3 = Vector3f::cross(v - v1, v - v2).length() / 2.;
    return ((1. / totalS) * Vector3f(total1, total2, total3));
}
```

Listing 6: nbarycentric

8.2 纹理映射

若构造 mesh 时读取到 vt 关键字，则程序会尝试打开 mesh 文件夹下于.obj 文件同名的.ppm 格式文件，并在求到交时基于顶点的纹理坐标，与上小节中所说的重心坐标计算该点的纹理坐标，并读取图片获得该点的颜色。

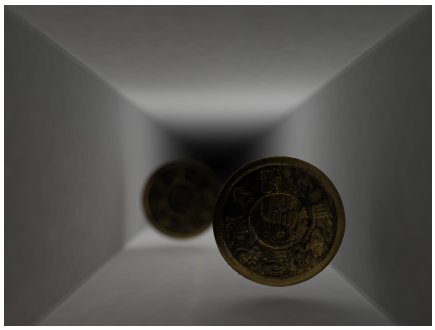

图 5: 景深与纹理映射

9 自制场景

场景中用到的 obj 模型：

- Stanford Dragon: 209227 faces.
- Chair: 96884 faces. (from www.artec-group.com)
- Table: 1000000 faces. (from www.artec-group.com)
- Glass: 88064 faces. (from PBRT) （注：场景中的顶灯其实是在拉伸的玻璃杯中放入点光源制成的，可以看到特殊的焦散效果。）

贴图和场景配置文件可在 Section 1 所述文件夹中找到。

图 6: room