Progressive Photon Mapping

—计算机图形学大作业报告

吴克文 梁家硕

2017 年 5 月

目录

1 综述 **2**

2 环境 **2**

3 运行 **2**

4 流程与算法 **2**

4.1 Lighting Equation 3

4.2 Ray Tracing Pass 4

4.3 Photon Tracing Pass 5

4.4 Progressive Updation 6

5 数理计算 **6**

5.1 射线与球的求交算法 6

5.2 射线与空间凸多边形的求交算法 7

5.3 射线与 KD-Tree 节点立方体的判交算法 9

6 效果展示 **10**

7 总结 **10**

1 综述

Photon Mapping 作为全局光照领域的主流算法，以其高效率，能处理多种光照效果等特点，一直受到广泛的关注。

然而，Photon Mapping 算法的一个主要问题在于，使用光子进行光能估计的过程引入了偏差。理论上，要完全消除偏差，需要存储无穷的光子，这从计算机存储角度来看是不可接受的。

为此，Toshiya Hachisuka 提出了 Progressive Photon Mapping(又称渐进式光子映射)，采用多遍的绘制流程，通过不断向场景中发射光子达到不断减小偏差的目的，亦解决了 Photon Mapping 的存储问题。

2 环境

Arch Linux x86_64 Linux 4.10.13-1-ARCH
gcc (GCC) 6.3.1 20170306
cmake version 3.8.0
GNU Make 4.2.1
OpenGL version: 3.0 Mesa 17.0.5

3 运行

```
$ cmake .
$ make
$ ./updation .
```

4 流程与算法

本作业参考了三篇论文 [1][3][4]，以及书 [2] 和 Stanford CS148 课件，综合效果和实现难度进行了调整，删去了繁琐的细节调整和性能优化部分。

注:

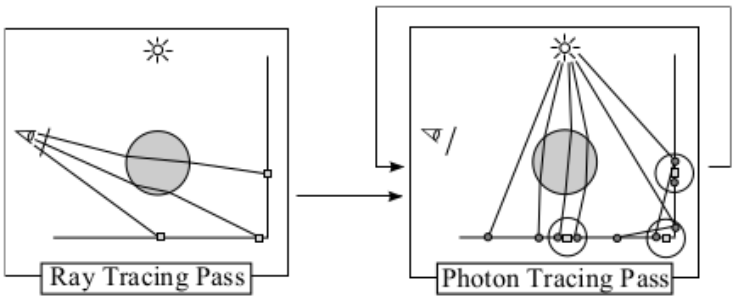
BRDF 参数及读取代码来自网站`http://www.merl.com/brdf/`。

Algorithm 1: Progressive Photon Mapping

Input: source.obj, material.mtl

Output: output.png

```
Read model and store material info.;
Build Model KD-Tree based on model info in .obj;
Perform Ray Tracing Pass to restore hitpoints;
for iterations do
    Perform Photon Tracing Pass;
    Build Photon KD-Tree with Photon Map;
    for hitpoints do
        Find photons near the hitpoint;
        Accumulate their impact on the hitpoint;
        Update hitpoint info.;
    end
end
Generate output.png using hitpoints' info.;
```



以下为详细算法剖分：

4.1 Lighting Equation

定义 4.1 BRDF(双向反射分布函数), 全称为 Bidirectional Reflectance Distribution Function，用来定义给定入射方向上的辐射照度如何影响给定出射方向上的辐射率。更笼统地说，它描述了入射光线经过某个表面反射后在各个出射方向上的分布效果。

利用 BRDF 描述光照方程，即，

$$L_o = \int_{i \in in} BRDF(\omega_i, \omega_o)\, dE_i \tag{1}$$

$$= \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i \, d\omega_i \tag{2}$$

其中，E_i 为入射辐照度，L_o, L_i 分别为出射和入射辐照率，θ_i 为入射光与平面法线的夹角，ω_o, ω_i 分别为出射和入射方向。

考虑到计算方法，将直接光照提出，得到便于计算的光照方程，

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} BRDF(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})\, d\omega_i \tag{3}$$

其中，L_e 为直接光照，$\mathbf{x}$ 为空间坐标，$\mathbf{n}$ 为平面法向量。

4.2 Ray Tracing Pass

从观察点出发，通过光线追踪来获得可见点 (hitpoints)，同时计算直接光照的贡献。

算法流程如下:

Algorithm 2: Ray Tracing Pass

Input: Camera, Lights, Width, Height, Scene

Output: Hitpoints, Pre-image

```
Initialize pre-image(width,height);
Initialize hitpoint list;
for i in range(0,width) do
    for j in range(0,height) do
        Initialize ray with (i,j) and camera place;
        Get the intersection of ray and scene;
        while intersection is specular do
            Determine whether it is reflected or refracted;
            Change the direction of ray;
            Re-get intersection;
        end
        Accumulate direct illumination;
        Store hitpoint;
    end
end
Return hitpoint list and pre-image;
```

注：

在镜面较多的场景中可用反（折）射次数作为阈值强制结束 Ray Tracing Pass。

4.3 Photon Tracing Pass

每轮 Photon Tracing Pass，从光源随机方向发射一批光子，追踪每个光子的运动轨迹，考虑到效率，将光子能量的衰减用随机被物体表面吸收（或达到折反射阈值）来控制，这样每个光子的能量即为定值，折反射仅改变其颜色向量（通过 BRDF 计算）。

同时，反射、折射和吸收的概率比可用 Schlick's Approximation 计算，

$$R(\theta_i) = R_0 + (1 - R_0)(1 - \cos\theta_i)^5 \tag{4}$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{5}$$

其中，n_1, n_2 为分界面折射率，$R(\theta_i)$ 和 $1 - R(\theta_i)$ 分别为反射、折射或吸收的概率。之后，折射和吸收的比率可由材质参数 Tr 确定。

算法流程如下：

Algorithm 3: Photon Tracing Pass

Input: Lights, Scene

Output: Photon Map

```
Initialize Photon Map;
for number of photons do
    Rand its initial direction;
    while photon not absorbed and not up to limitation do
        Get its intersection with scene;
        Store intersection in Photon Map;
        if photon not absorbed then
            Determine whether it is reflected or refracted;
            Update its direction;
        end
    end
end
Return Photon Map;
```

注：

由于直接光源已在 Ray Tracing Pass 计算过，故每个光子与场景的第一个交点不必计入 photon map。

4.4 Progressive Updation

结束 Photon Tracing Pass 后，需要枚举每个 hitpoint，同时统计其半径 R 内光子对其亮度影响。

此处采用 *photon KD-Tree* 优化邻近点的搜索。

最后减小 hitpoint 半径，达到每轮增加亮度的同时，收敛到正确值。

记 $N(\mathbf{x})$ 为上轮后在 hitpoint $\mathbf{x}$ 半径 $R(\mathbf{x})$ 内的光子数，$M(\mathbf{x})$ 为本次新增光子数，同时 $\hat{N}(\mathbf{x}), \hat{R}(\mathbf{x})$ 分别为新累计光子数和半径，则有如下更新，

$$\hat{N}(\mathbf{x}) = N(\mathbf{x}) + \alpha M(\mathbf{x}) \tag{6}$$

$$\hat{R}(\mathbf{x}) = R(\mathbf{x})\sqrt{\frac{N(\mathbf{x}) + \alpha M(\mathbf{x})}{N(\mathbf{x}) + M(\mathbf{x})}} \tag{7}$$

同样的，记 $\tau_N(\mathbf{x}, \omega)$ 和 $\tau_M(\mathbf{x}, \omega)$ 为在 $\mathbf{x}$ 处，入射光方向为 ω 的前光强和新增光强（未乘 BRDF 系数），则有

$$\tau_{\hat{N}}(\mathbf{x}, \omega) = (\tau_N(\mathbf{x}, \omega) + \tau_M(\mathbf{x}, \omega))\frac{N(\mathbf{x}) + \alpha M(\mathbf{x})}{N(\mathbf{x}) + M(\mathbf{x})} \tag{8}$$

其中，$\alpha \in (0, 1)$ 是一常数。再记总发射光子数为 $N_{emitted}$，ϕ 为光子光强，则最终辐照率表达式为，

$$L(\mathbf{x}, \omega) = \int_{2\pi} BRDF(\mathbf{x}, \omega, \omega')L(\mathbf{x}, \omega')(\mathbf{n} \cdot \omega')\,(d)\omega' \tag{9}$$

$$\approx \frac{1}{\Delta A}\sum_{p=1}^{n} BRDF(\mathbf{x}, \omega, \omega')\Delta\phi_p(\mathbf{x_p}, \omega_p) \tag{10}$$

$$= \frac{1}{\pi R(\mathbf{x})^2}\frac{\tau(\mathbf{x}, \omega)}{N_{emitted}} \tag{11}$$

详细推导及细节见 [1]。

5 数理计算

除了整体性的大算法，该大作业也有一些细节上的数理计算方法也值得一提。这些计算虽然底层，但由于调用次数巨大，需要进行常数优化。

5.1 射线与球的求交算法

用起点 $\mathbf{s}$ 和向量 $\mathbf{v}$ 表示射线 l，用球心 $\mathbf{o}$ 和半径 R 表示球 S，设点 $\mathbf{d} = \mathbf{s} + x\mathbf{v}$ 为线圆交点，则有：

$$\|\mathbf{s} + x\mathbf{v} - \mathbf{o}\| = R \tag{12}$$

令 $\mathbf{t} = \mathbf{s} - \mathbf{o}$，并展开方程 (12)，有，

$$\|\mathbf{t}\|^2 - R^2 + 2(\mathbf{t}, \mathbf{v})x + \|\mathbf{v}\|^2 x^2 = 0 \tag{13}$$

再取最小正值 x 带入，即得交点。

CODE: Ray & Sphere

```
// return INF if no intersection
double intersect(const Ray &ray, const Sphere &s, Point *p,
    double lasthit){
    Point t = ray.bgn - s.center;
    double b = 2 * dotsProduct(t, ray.vec),
           c = t.len2() - sqr(s.radius),
           delta = sqr(b) - 4 * c;
    if (delta < EPS) return INF;  // no intersection
    delta = sqrt(delta);
    double t1 = (-b - delta) / 2,  // calculate two
           t2 = (-b + delta) / 2,  // possible answer
           res;
    if (t1 < EPS) if (t2 < EPS) return INF;  // tangent
                      else res = t2;
       else if (t2 < EPS) res = t1;
               else res = std::min(t1, t2);
    if (res >= lasthit) return INF;  // not first intersected
    *p = ray.bgn + res * ray.vec;
    return res;
}
```

5.2 射线与空间凸多边形的求交算法

多边形所在平面可用任意三点表示，又考虑到精度问题，便在初始化多边形时选取多边形上构成三角形面积最大的三点 $\mathbf{c_1}, \mathbf{c_2}, \mathbf{c_3}$。求出射线与平面的交点，再带回验证是否在凸多边形内即可。

用起点 $\mathbf{s}$ 和向量 $\mathbf{v}$ 表示射线 l，设点 $\mathbf{d} = \mathbf{s} + x\mathbf{v}$ 为射线和多边形的交点。

因为交点在平面上，故，

$$\det(\mathbf{s} + x\mathbf{v} - \mathbf{c_1}, \mathbf{c_2} - \mathbf{c_1}, \mathbf{c_3} - \mathbf{c_1}) = 0 \tag{14}$$

令 $\mathbf{t}=\mathbf{s}-\mathbf{c_1}$，$\mathbf{c}'=\mathbf{c_2}-\mathbf{c_1}$，$\mathbf{c}''=\mathbf{c_3}-\mathbf{c_1}$，有

$$\begin{aligned}\det(\mathbf{t}+x\mathbf{v},\mathbf{c}',\mathbf{c}'') &= \det(\mathbf{t},\mathbf{c}',\mathbf{c}'')+x\det(\mathbf{v},\mathbf{c}',\mathbf{c}'') && (15)\\ &= \mathbf{t}_x\times yz-\mathbf{t}_y\times xz+\mathbf{t}_z\times xy && (16)\\ &\quad +x(\mathbf{v}_x\times yz-\mathbf{v}_y\times xz+\mathbf{v}_z\times xy) && (17)\\ &= 0 && (18)\end{aligned}$$

其中，

$$\begin{aligned}xy &= \mathbf{c}'_x\mathbf{c}''_y-\mathbf{c}'_y\mathbf{c}''_x && (19)\\ xz &= \mathbf{c}'_x\mathbf{c}''_z-\mathbf{c}'_z\mathbf{c}''_x && (20)\\ yz &= \mathbf{c}'_y\mathbf{c}''_z-\mathbf{c}'_z\mathbf{c}''_y && (21)\end{aligned}$$

考虑如果 $\mathbf{d}$ 在凸多边形内，则有 $\mathbf{d}$ 在平面上每条边的左侧，记平面法向量为 $\mathbf{n}$，多边形逆时针点序为 $\{\mathbf{p_i}\}$，则有，

$$\begin{aligned}(\mathbf{n},(\mathbf{p_{i+1}}-\mathbf{p_i})\times(\mathbf{d}-\mathbf{p_i})) &= \det(\mathbf{n},\mathbf{p_{i+1}}-\mathbf{p_i},\mathbf{d}-\mathbf{p_i}) && (22)\\ &= \det(\mathbf{p_i},\mathbf{p_{i+1}},\mathbf{n})+\det(\mathbf{p_i}-\mathbf{p_{i+1}},\mathbf{n},\mathbf{d}) && (23)\\ &\geqslant 0 && (24)\end{aligned}$$

CODE: Ray & Polygon

```
// return INF if no intersection
double intersect(const Ray &ray, const Polygon &s, Point *p,
    double lasthit){
    Point ts = ray.bgn - s.pList[s.c1];
    double k = ray.vec.x * s.yz - ray.vec.y * s.xz + ray.vec.z
        * s.xy,
           b = ts.x * s.yz - ts.y * s.xz + ts.z * s.xy;
    if (fabs(k) < EPS) return INF;  // parallel
    double t = -b / k;
    if (t < EPS || t >= lasthit) // intersection is behind
        return INF;              // or not first intersected
    Point ret = ray.bgn + t * ray.vec;  // intersection
    // pre-calculation
    double txy = s.normvf.x * ret.y - s.normvf.y * ret.x,
           txz = s.normvf.x * ret.z - s.normvf.z * ret.x,
           tyz = s.normvf.y * ret.z - s.normvf.z * ret.y;
```

```
    for (int i = 0; i < s.num; ++i) {
        int ni = (i + 1) % s.num;
        if (determinant(s.pList[i], s.pList[ni], s.normvf) +
            (s.pList[i].x - s.pList[ni].x) * tyz -
            (s.pList[i].y - s.pList[ni].y) * txz +
            (s.pList[i].z - s.pList[ni].z) * txy < -EPS)
            return INF;  // not on the left
    }
    *p = ret;
    return t;
}
```

5.3 射线与 KD-Tree 节点立方体的判交算法

经过尝试，传统 KD-Tree 在片元较多时光线判交复杂度过大，难以胜任，故我们改进了传统 KD-Tree 建法：

枚举 x, y, z 三个方向的剖分方法，选出每个方向的最优剖分面（最优被定义为完全在左侧的面片和完全在右侧的面片个数差尽量小），经过比较，选出最终分界面。然后将面片分为三组：分界面左侧，经过分界面和分界面右侧。再递归构造左子树、中子树与右子树。

此建树方法，考虑到了经过分界面的面片数量不会太多，且左右区域的包围盒不交时，便于后续射线求交的剪枝。事实证明，这种建树方法，结合下面介绍的剪枝算法，将 KD-Tree 判交过程加速颇多。

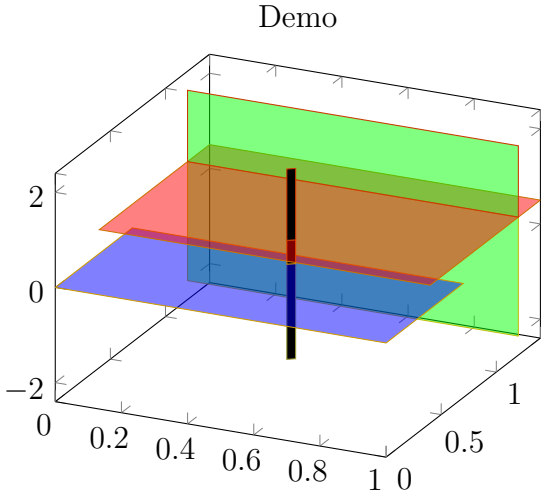
基于这样的建树模式，介绍如下判交算法：

首先，KD-Tree 预处理出外接立方体的边界，并记录每次划分区域的分界面。

不失一般性地，记 t 轴为当前 KD-Tree 节点分界面，为 x, y, z 轴中的某一个。x_{min}, x_{max} 分别为立方体在 x 轴上的边界，同理定义 y_{min}, y_{max} 和 z_{min}, z_{max}。用起点 $\mathbf{s}$ 和向量 $\mathbf{v}$ 表示射线 l，设点 $\mathbf{d} = \mathbf{s} + x\mathbf{v}$ 为交点。计算出当前射线在立方体六个平面上的交点（事实上为三对正 x 值），如果三对 x 范围的交为空，则表示射线与立方体无交，退出。若有交，则可行的最小值即为射线与立方体的交点 $\mathbf{c}$（若起点 $\mathbf{s}$ 在立方体中，则规定 $\mathbf{c} = \mathbf{s}$）。

考虑判断该交点的位置，若在分界面左侧，则先与 KD-Tree 当前节点左子树判交，然后将交点与 $\mathbf{c}$ 的距离与 $\mathbf{s}$ 到中子树的最短距离 x' 进行比较，若已短于 x' 则不必判中子树与右子树，直接返回，反之则再与中子树判交，再考虑右子树。同理可得 $\mathbf{c}$ 在分界面右侧的情况。

还可以加上射线方向性的剪枝。考虑 $\mathbf{c}$ 在分界面左侧，且 $\mathbf{v}_t \leqslant 0$，则显然 l 不会与右子树有交。值得一提的是，此时 l 与左子树有交并不能保证它就是最近交点，因为左包围盒与中包围盒有交时情况较为复杂，如下图所示：



上图中绿色平面为分界面，红色面在中子树中，蓝色面在左子树中，黑色的射线从上而下。射线与整个包围盒的交点在左子树，故先求得射线与（左子树）蓝色面的交点，但是，它与中子树（红色面）的交点才是最近点。

注：

为了压常数，代码写的不是很可读，在此略去。

6 效果展示

7 总结

本算法还有诸多可以提高的空间，例如：

1. 引入凹凸纹理贴图
2. 并行计算优化效率
3. Stochastic Progressive Photon Mapping 及其他进阶算法

限于时间、精力，无法进一步探索，实属遗憾。

参考文献

[1] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Transactions on Graphics (TOG)*, 27(5):130, 2008.

[2] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*, volume 364. Ak Peters Natick, 2001.

[3] Ben Spencer and Mark W Jones. Progressive photon relaxation. *ACM Transactions on Graphics (TOG)*, 32(1):7, 2013.

[4] 李睿, 陈彦云, and 刘学慧. 基于自适应光子发射的渐进式光子映射. 计算机工程与设计, 33(1):219–223, 2012.