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 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

2 运行

```
$ make
```

3 核心算法

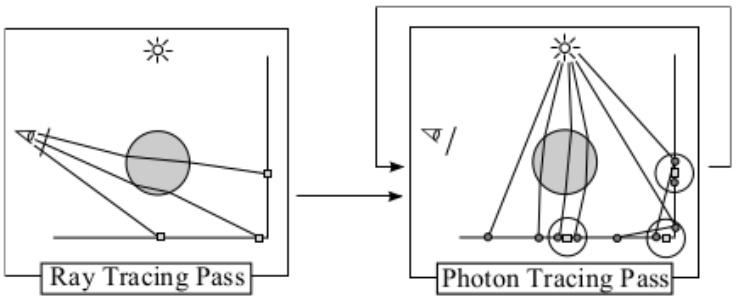
本作业参考了三篇论文 [1][3][4]，以及书 [2] 和 Stanford CS148 课件，综合效果和实现难度进行了调整，删去了繁琐的细节调整和性能优化部分。
注:

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:

3.1 Lighting Equation

定义 3.1 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。

3.2 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。

3.3 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。

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。

3.4 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]。

4 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。

4.1 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，

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

再取最小正值 x 带入，即得交点。

CODE: Ray & Sphere

```
bool Ray::intersect(const Sphere &s, Point *p) const{
    Point t = bgn - s.center;
    double b = 2 * dotsProduct(t, vec),
           c = t.len2() - sqr(s.radius),
           delta = sqr(b) - 4 * c;
    if (delta < EPS) return 0;  // no intersection
    delta = sqrt(delta);
    double t1 = (-b - delta) / 2,  // calculate two
           t2 = (-b + delta) / 2,  // possible solution
           res;  // final answer
```

```
    if (t1 < EPS) if (t2 < EPS) return 0;  // tangent
                  else res = t2;
       else if (t2 < EPS) res = t1;
               else res = std::min(t1, t2);
    *p = bgn + res * vec;
    return 1;
}
```

4.2 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，

$$xy = \mathbf{c}'_x\mathbf{c}''_y - \mathbf{c}'_y\mathbf{c}''_x \tag{19}$$

$$xz = \mathbf{c}'_x\mathbf{c}''_z - \mathbf{c}'_z\mathbf{c}''_x \tag{20}$$

$$yz = \mathbf{c}'_y\mathbf{c}''_z - \mathbf{c}'_z\mathbf{c}''_y \tag{21}$$

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

```
bool Ray::intersect(const Polygon &s, Point *p) const{
    Point ts = bgn - s.pList[s.c1];
    double k = vec.x * s.yz - vec.y * s.xz + vec.z * s.xy,
           b = ts.x * s.yz - ts.y * s.xz + ts.z * s.xy;
    if (fabs(k) < EPS) return 0; // parallel
    double t = -b / k;
    Point ret = bgn + t * 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 - 1; ++i)
        if (determinant(s.pList[i], s.pList[i + 1], s.normvf) +
            (s.pList[i].x - s.pList[i + 1].x) * tyz -
            (s.pList[i].y - s.pList[i + 1].y) * txz +
            (s.pList[i].z - s.pList[i + 1].z) * txy < -EPS)
            return 0; // not on the left
    if (determinant(s.pList[s.num - 1], s.pList[1], s.normvf) +
        (s.pList[s.num - 1].x - s.pList[1].x) * tyz -
        (s.pList[s.num - 1].y - s.pList[1].y) * txz +
        (s.pList[s.num - 1].z - s.pList[1].z) * txy < -EPS)
        return 0;

    *p = ret;
    return 1;
}
```

4.3 射线与 KD-Tree 节点立方体的判交算法

CODE: Ray & ???

```

```

5 效果展示

6 总结

本算法还有诸多可以提高的空间，例如：

1. 引入纹理贴图
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.