图形学大作业实验报告

计 95　刘玉河　2019011560

2021 年 6 月 27 日

1　渐进式光子映射绘制算法

渐进式光子映射（Progressive Photon Mapping），是第一种能够在任意精度下计算所有类型的光传播的算法。它的渲染流程与先追踪光子再追踪视线的光子映射（Photon Mapping）相反，具体流程如下：

1. 从相机处发射并追踪视线，确定视线在漫反射面上的落点，将落点按照其空间坐标，存放在哈希表中。
2. 从光源处发射并追踪光子，在光子落点处利用哈希表寻找周围的视线落点，若光子在视线落点的观察半径范围内，则将其纳入该落点的光强估计中，并按一定的系数缩减视线落点的观察半径。
3. 光子追踪的迭代结束后，计算每一个视线落点的光强，得到渲染好的图案。

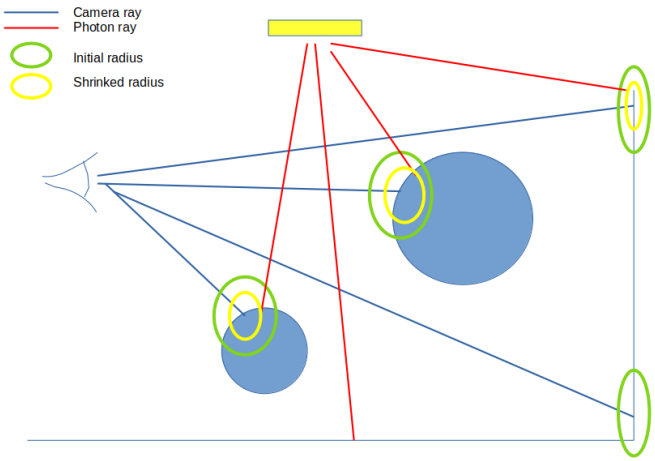

图 1: PPM 渲染算法示意图

类 Python 风格伪代码（完整代码请见附件代码 include/ppm.hpp）：

```
    def trace(ray, isCamera):
        if(hitpoint.type != matte):
            trace(new_ray, isCamera)
        else if isCamera:
```

```
            hitpoints.append(hitpoint)
        else:
            for hp in hash_grid.getNeighbours(hitpoint):
                if hitpoint in hp.radius:
                    hp.flux_add(hitpoint)
                    hp.hit_num++
                    hp.radius_shrink()
    def trace_ray():
        for pixel in camera_frame:
            trace(camera.gen_ray(pixel), isCamera=True)
        build_hash_grid(hitpoints)
    def trace_photon():
        for light in lights:
            trace(light.gen_photon(), isCamera=False)
    def density_estimation():
        for hp in hitpoints:
            color[hp.idx] += hp.flux / (pi * hp.radius^2 * num_photons)
    def main():
        trace_ray()
        trace_photon()
        density_estimation()
```

2 附加功能

2.1 软阴影

由于 PPM 是通过在光源处发射并追踪光子，通过设置光子发射的源点形成一定面区域，就能够实现面光源，进而产生物体的软阴影。

```
// Code clip from include/light.hpp
class AreaLight: public Light {
    Vector3f position; // center of the eclipse
    Vector3f axis1, axis2;  // axes of eclipse
    Vector3f n1, n2;    // normalized axes
public:
    AreaLight() = delete;
    AreaLight(...) : position(p), axis1(a1), axis2(a2)
    {...}
    ~AreaLight() override = default;
```

```
    virtual void genp(Ray& pr, Vector3f *f, int i) override
    {
        (*f) = flux * (D_PI * 4.0) * color; // flux
        auto angle = 2.0 * D_PI * halton(0, i);
        auto radius = halton(3,i);
        auto x1 = axis1 * sin(angle),
             x2 = axis2 * cos(angle);   // (x1, x2) would be the position generates photon
        auto p = 2.0 * D_PI * halton(0, i);
        auto t = 2.0 * acos(sqrt(1. - halton(1, i)));   // p, t for determining direction
        auto st = sin(t);
        pr = Ray(position + (x1 + x2) * radius, Vector3f(cos(p) * st, cos(t), sin(p) * st));
    }
};
```

2.2 柏林噪声材质

通过利用 Perlin Noise，可以使物体上的点呈现空间相关的颜色纹理，参考课件和网络资料，主要利用柏林噪声实现了木头 wood 和大理石 marble 两种材质。

```
// Code clip from include/texture.hpp
#include "FractalNoise.hpp"
#include "Perlin.hpp"
#include "material.hpp"
#include <cmath>
float cut(float x){
    return x - (int)x;
}
//https://www.researchgate.net/publication/340769728_REALISTIC_RENDERING_OF_WOOD
Vector3f wood(const Vector3f& x, Perlin* gen)
{
    float g = fabs(sin(x.x() + 2 * gen->noise(x.x(), x.y(), x.z()))) * 10;
    return cut(g) * Vector3f(0.345, 0.188, 0.074);
}
Vector3f marble(const Vector3f& x, Perlin* gen)
{
    return (cos((3 * x.x() -  2 * x.y() + 20 * gen->noise(x.x(), x.y(), x.z()))) + 1)/2
    * Vector3f(0.761, 0.698, 0.502);
}
```

此外，单纯利用点的 z 坐标实现了彩虹纹理。

```
Vector3f rainbow(const Vector3f& x, Perlin* gen)
{
    float z = abs(x.z()*5);
    float zz = z - floor(z);
    switch((int)z % 6) // smooth transition from (255,0,255) to (0,0,0)
    {
        case 0: return Vector3f(1,0, 1 - zz) * 0.8;
        case 1: return Vector3f(1,zz, 0) * 0.8 ;
        case 2: return Vector3f(1-zz, 1, 0) * 0.8;
        case 3: return Vector3f(0, 1, zz) * 0.8;
        case 4: return Vector3f(0, 1 - zz, 1) * 0.8;
        case 5: return Vector3f(zz, 0, 1) * 0.8;
        default: return Vector3f(0,0,0) * 0.8;
    }
}
```

2.3 纹理映射

通过在网格文件中指定每个顶点对应材质文件中的 uv 坐标，即可在渲染时通过坐标插值确定每一个三角面片的材质。对于参数曲面的贴图，直接由参数曲面的 uv 值确定其在贴图中的坐标。

```
// Code clip from src/mesh.cpp
Mesh::Mesh(const char *filename, Material *material) : Object3D(material) {
    //...
        } else if (tok == texTok) { // Texture point
            int vid, u, v;
            ss >> vid >> u >> v;
            vts[vid-1].first = u;
            vts[vid-1].second = v;
        }
    // ...
}
// Code clip from include/triangle.hpp
inline Vector3f getColor(float beta, float gamma) {
    if(material->texture_type == 1) // texture_type == 1 means it uses bmp texture file
    {
        float alpha = 1 - beta - gamma;
```

```
        int u = texcord[0][0] * alpha + texcord[1][0] * beta + texcord[2][0] * gamma;
        int v = texcord[0][1] * alpha + texcord[1][1] * beta + texcord[2][1] * gamma;
        return material->getUV(u, v);
    }
    return material->Color;
}
}
```

2.4 参数曲面解析法求交

求射线与参数曲面交点等价于求解下述方程：

$$F(x) = F(t,u,v) \triangleq L(t) - P(u,v) = 0 \tag{1}$$

$$L(t) = \text{ray.pointAtParameter(t)} = O + d \cdot t \tag{2}$$

$$P(u,v) = \sum_i \sum_j P_{ij} B_{i,m}(u) B_{j,n}(v) \tag{3}$$

实际求解时，先通过参数曲面的 KdTree 存储的网格化模型进行粗求交，得到对应的 t 和相交网格对应的 (u, v)，再利用牛顿迭代法逼近精确解：

$$x_{i+1} = x_i - [F'(x_i)]^{-1} \cdot F(x_i) \tag{4}$$

$$F'(x) = \begin{bmatrix} \frac{\partial F}{\partial t} & \frac{\partial F}{\partial u} & \frac{\partial F}{\partial v} \end{bmatrix} = \begin{bmatrix} d & -\sum_{u,v} PB'(u)B(v) & -\sum_{u,v} PB(u)B'(v) \end{bmatrix} \tag{5}$$

观察式4、5，可以发现在牛顿迭代过程中实际需要计算的是 Berstein 基函数及其导数 $B(u), B(v), B'(u), B'(v)$，这个求值过程在 Berstein 类的 evaluate 函数中进行，由于已经在 PA3 中实现，此处不再赘述。

```
// Code clips from include/surface.hpp
bool intersect(const Ray &r, Hit &h, float tmin) override {
    std::queue<KdNode*> que;
    que.push(root);
    while(!que.empty()) {
        // ...
        if(cur->lc == nullptr && cur->rc == nullptr) {// leaf node
            float u = cur->bbox.u, v = cur->bbox.v; // initial uv
            // After Newton iteration we should get result(t), and tangent un, vn
            // (it should be 'ut' and 'vt' for tangent, here it's just a typo.)
            Newton(r, th.t, u, v, result, un, vn);
            if(result > tmin && result < h.t) { // the resulting t must be valid
                re = true;
                h.set(result, Vector3f::cross(un, vn).normalized(),
```

```
                material->Color, material->type, material->texture_type);
            }
        }
        // ...
    }
    return re;
}

bool Newton(const Ray& ray, float t, float u, float v,
            float& ret, Vector3f& un, Vector3f& vn)
{
    float dis;
    vector<double> us, uds, vs, vds;
    Vector3f next = SingleNewton(ray, Vector3f(t, u, v), dis, us, uds, vs, vds, un, vn);
    int step = 0;
    while(dis > 0.1 && step < 10)
    {
        next = SingleNewton(ray, next, dis, us, uds, vs, vds, un, vn);
        step++;
    }
    if(step == 10)  // stop early
        return false;
    ret = next.x();
    return true;
}

Vector3f SingleNewton(const Ray& ray, Vector3f tuv, float& dis,
                      std::vector<double>& us, std::vector<double>& uds,
                      std::vector<double>& vs, std::vector<double>& vds,
                      Vector3f& un, Vector3f &vn) const
{
    //F'(x) = [ray.direction, - partial P/partial u, - partial P/partial v]
    us.clear(), uds.clear(), vs.clear(), vds.clear();
    int ulsk = ub->evalute(tuv.y(), us, uds);
    int vlsk = vb->evalute(tuv.z(), vs, vds);
    Vector3f p = ray.pointAtParameter(tuv.x()); // F(xi)
    Vector3f ud;    // partial P/ partial u
    Vector3f vd;    // partial P/ partial v
    for(int i = 0; i < us.size(); ++i)
```

```
    {
        for(int j = 0; j < vs.size(); ++j)
        {
            Vector3f point = controls[ulsk+i][vlsk+j];
            p -= point * us[i] * vs[j];
            ud += point * uds[i] * vs[j];
            vd += point * us[i] * vds[j];
        }
    }
    dis = p.length();
    Matrix3f Fi = Matrix3f(ray.direction, -ud, -vd).inverse();
    un = ud, vn = vd;
    return tuv - Fi * p;  // x_{i+1}
}
```

2.5 复杂网格模型：AABB&KdTree 求交加速

为了提高场景内的求交速度，使用了 AABB 和 KdTree 对场景内物品进行预处理：先对每一个物品创建一个指向自身的 AABB，再将 AABB 对应到 KdTree 的叶节点中，自底向上建树。其中网格模型自身也是一棵 KdTree，里面包含了所有三角面片的 AABB。

使用 AABB 和 KdTree 组合的好处首先在于 AABB 自身的求交算法简单，其次是 KdTree 建树时可以直接利用 AABB 的三个坐标轴对应坐标进行从大到小排序，实现起来比较简便，速度也较为理想。

```
// Code clip from include/aabb.hpp
class AABB
{
public:
    float u, v;
    float axis_planes[3][2]; // 0:x, 1:y, 2:z // 0: min, 1: max
    AABB() {}
    explicit AABB(std::vector<std::pair<float, float>> axis, Object3D* content = nullptr)
    : content(content)
    {
        axis_planes[0][0] = axis[0].first, axis_planes[0][1] = axis[0].second;
        axis_planes[1][0] = axis[1].first, axis_planes[1][1] = axis[1].second;
        axis_planes[2][0] = axis[2].first, axis_planes[2][1] = axis[2].second;
    }
    bool intersect(const Ray &r, Hit &h, float tmin)
    {
```

```
        Vector3f o = r.getOrigin();
        if(inside(o))
            return true;
        float tm = 1e38; int idx = 0; bool re = false; Hit ans;
        for(int i = 0; i < 3; ++i)
        {
            int close = fabsf(axis_planes[i][0] - o[i]) > fabsf(axis_planes[i][1] - o[i])
                        ? 1 : 0;
            Hit ht;
            float temp;
            if (get_axis_plane_t(i, close, r, ht))
                temp = ht.getT();
            else
                continue;
            Vector3f inte = r.pointAtParameter(ht.getT());
            bool inside = true;
            for (int j = 0; j < 3; ++j) {
                if (j == i)
                    continue;
                if ((inte[j] > axis_planes[j][1]) || (inte[j] < axis_planes[j][0]))
                    inside = false; break;
            }
            if((temp < tm) && inside) {
                tm = temp; idx = i;
                ans = ht; re = true;
            }
        }
        if(ans.getT() > h.getT() || ans.getT() < tmin)
            return false;
        h = ans;
        return re;
    }

    bool inside(const Vector3f& v)
    {
        if(v.x() < axis_planes[0][0] || v.x() > axis_planes[0][1])
            return false;
        if (v.y() < axis_planes[1][0] || v.y() > axis_planes[1][1])
            return false;
```

```
        if (v.z() < axis_planes[2][0] || v.z() > axis_planes[2][1])
            return false;
        return true;
    }

    Object3D* content;

    static AABB merge(const AABB &a, const AABB &b) {
        AABB re;
        re.content = nullptr;
        for (int i = 0; i < 3; ++i) {
            re.axis_planes[i][0] = min(a.axis_planes[i][0], b.axis_planes[i][0]);
            re.axis_planes[i][1] = max(a.axis_planes[i][1], b.axis_planes[i][1]);
        }
        return re;
    }
protected:
    bool get_axis_plane_t(int axis, int direction, const Ray& r, Hit &h) {
        Vector3f normal(dir[axis][0], dir[axis][1], dir[axis][2]);
        float D = -axis_planes[axis][direction];
        float t = -(D + Vector3f::dot(normal, r.getOrigin()))
                / Vector3f::dot(normal, r.getDirection());
        if (t > h.getT() || t < 0)
            return false;
        Vector3f n = Vector3f::dot(r.getDirection(), normal) > 0 ? -normal : normal;
        h.set(t, n);
        return true;
    }
};
// Code clip from include/kdnode.hpp
class KdNode
{
public:
    KdNode *lc, *rc;
    AABB bbox;
    KdNode(AABB b, KdNode*lc = nullptr, KdNode* rc = nullptr)
    : lc(lc), rc(rc), bbox(b) { }

    bool intersect(const Ray &r, Hit &h, float tmin)
```

```
    {
        std::queue<KdNode*> que;
        que.push(this);
        Hit tmp; Hit th;
        bool re = false;
        while(!que.empty())
        {
            th = tmp;
            KdNode* cur = que.front();
            que.pop();
            if(!cur->bbox.intersect(r, th, tmin))
                continue;
            if(cur->lc == nullptr && cur->rc == nullptr)
                re |= cur->bbox.content->intersect(r, h, tmin);
            if(cur->lc) que.push(cur->lc);
            if(cur->rc) que.push(cur->rc);
        }
        return re;
    }

    static bool cmpX(const AABB &a, const AABB &b){//...}
    static bool cmpY(const AABB &a, const AABB &b){//...}
    static bool cmpZ(const AABB &a, const AABB &b){//...}

    static KdNode *split(int start, int end, int dir, std::vector<AABB> &bboxs) {
        if (end == start) return nullptr;
        if (end == start + 1) return new KdNode(bboxs[start]);
        if (dir == 0)
            sort(bboxs.begin() + start, bboxs.begin() + end, cmpX);
        else if (dir == 1)
            sort(bboxs.begin() + start, bboxs.begin() + end, cmpY);
        else
            sort(bboxs.begin() + start, bboxs.begin() + end, cmpZ);
        KdNode *lc = split(start, start + (end - start) / 2, (dir + 1) % 3, bboxs),
               *rc = split(start + (end - start) / 2, end, (dir + 1) % 3, bboxs);
        AABB box = AABB::merge(lc->bbox, rc->bbox);
        return new KdNode(box, lc, rc);
    }
};
```

3 参考资料

1. PA1 代码框架
2. 渐进式光子映射资料

   http://graphics.ucsd.edu/~henrik/papers/progressive_photon_mapping/progressive_photon_mapping.pdf
3. smallppm 代码 https://cs.uwaterloo.ca/~thachisu/smallppm_exp.cpp
4. PerlinNoise generator https://github.com/captainhead/PerlinNoiseCpp