真实感渲染

尹绍沣　　2022012760　　致理-信计 21

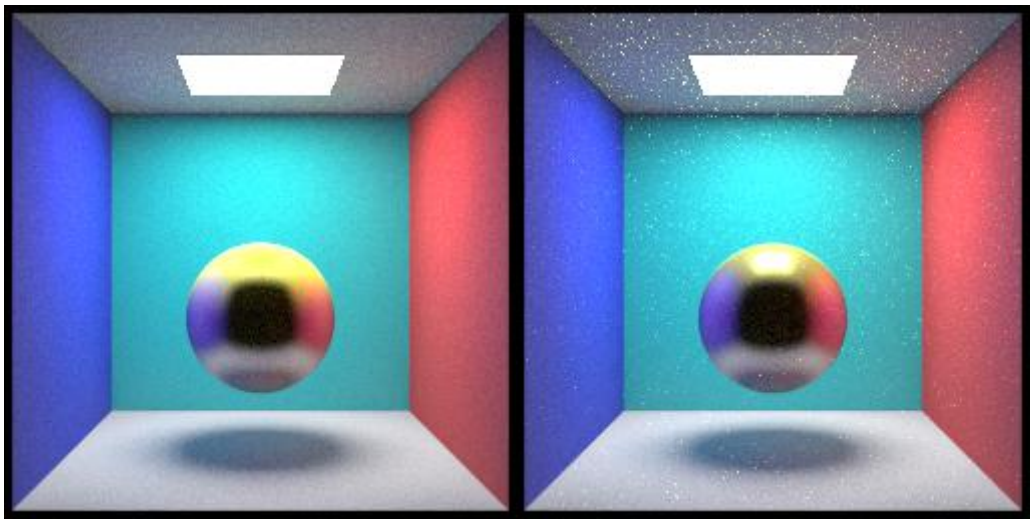
实现功能

验收后实现的功能：（按从基本到高阶的顺序）

****此处放一些过程图，结果图请见最后****

1. Glossy 材质（改用`Disney Principal BRDF`方式实现）

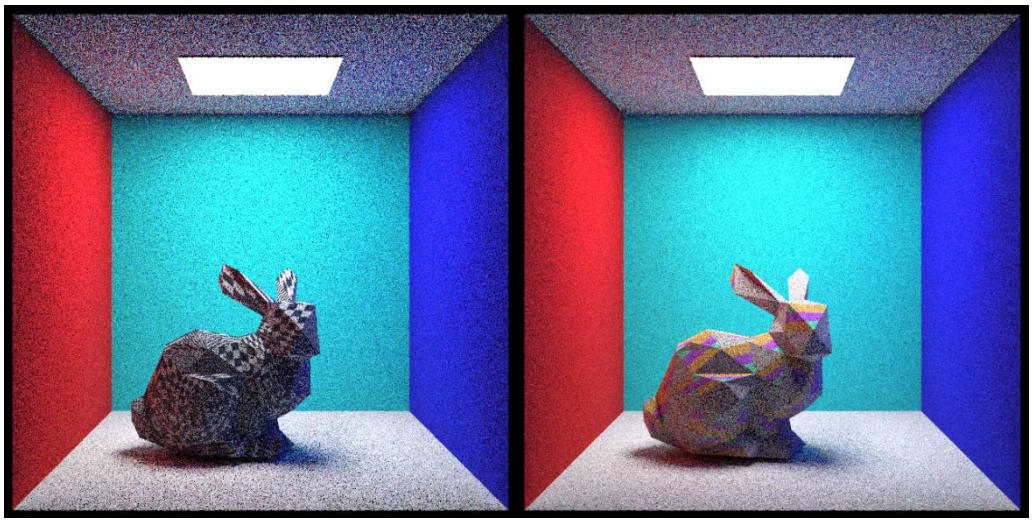
实现前后对比：



实现前　　实现后

注：实现前简单地使用漫反射和理想反射混合，事实证明效果不佳

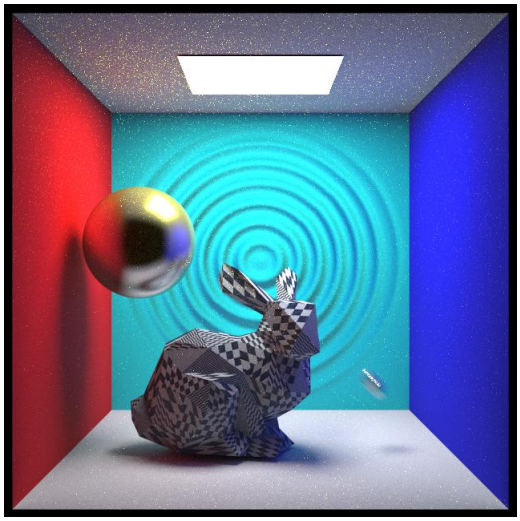
2. 纹理贴图



上面两张图只进行了 4spp，故仍存在较多噪点

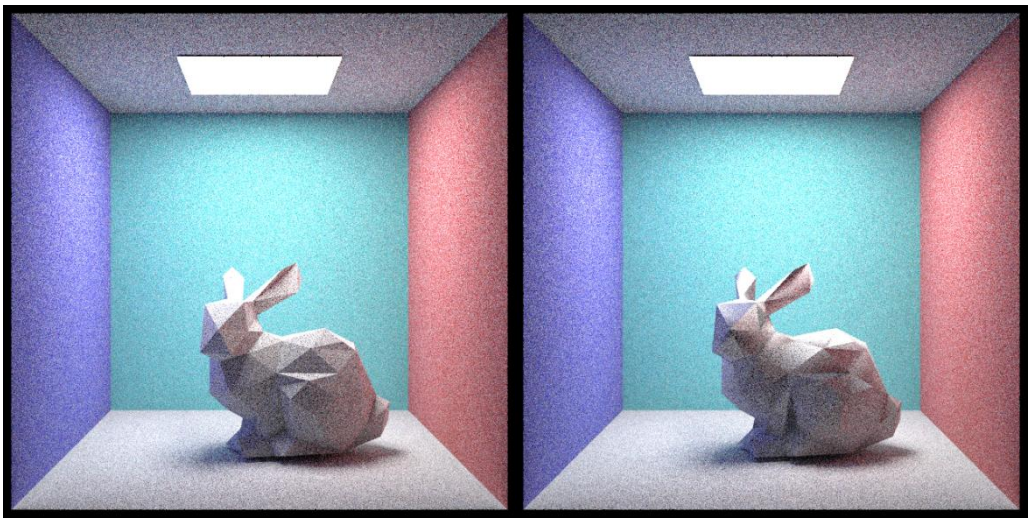
结果图中有更为清晰的效果

3．法线贴图



后面墙壁

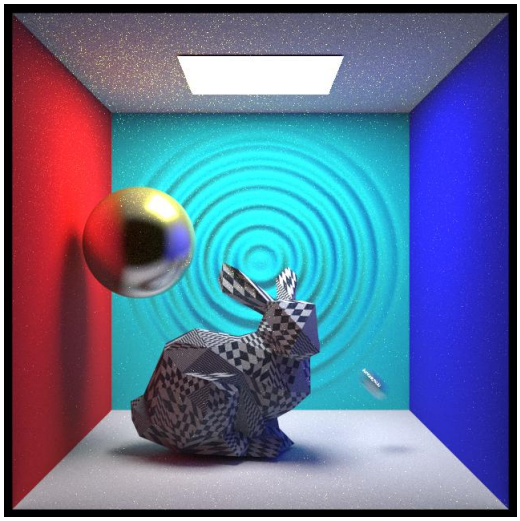
4．法向插值



实现前　　实现后

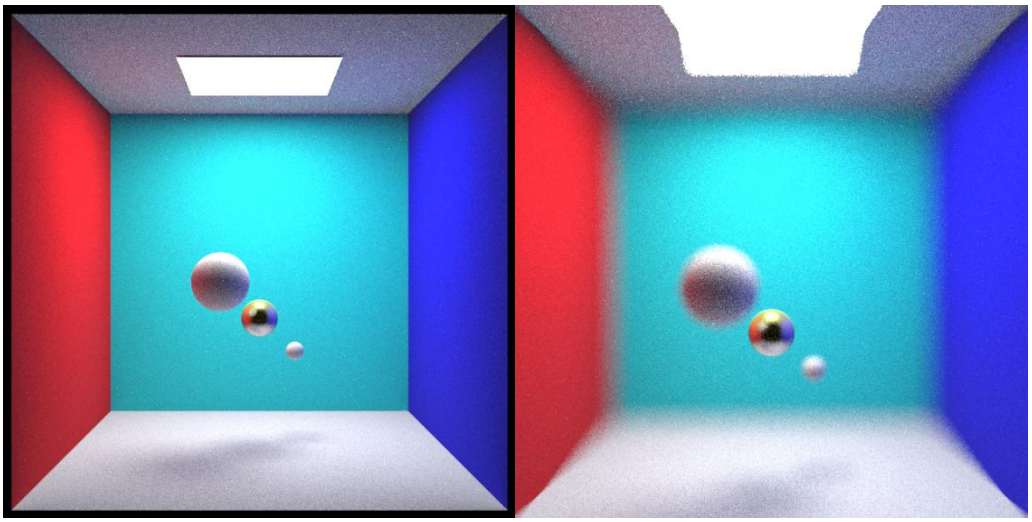
注：from legacy method（即使用 Disney Principal BRDF 前）

5．运动模糊



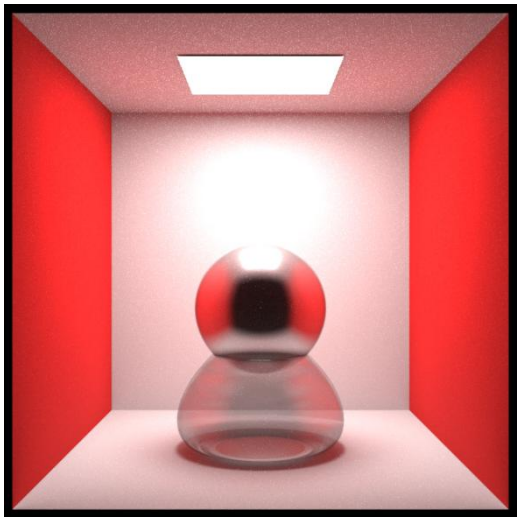
右侧小球

6．景深效果



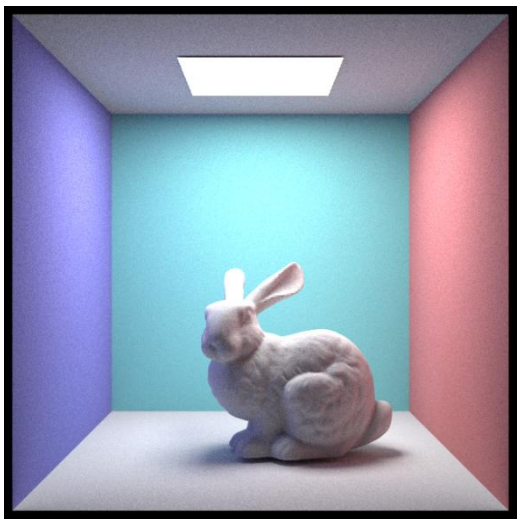
无景深　　有景深

7．参数曲面解析法求交



玻璃瓶

8. 复杂网格模型及其求交加速



使用 BVH Tree 进行加速

使用网络学堂上 bunny.fine.obj
共 70,580 个面片

速度对比：

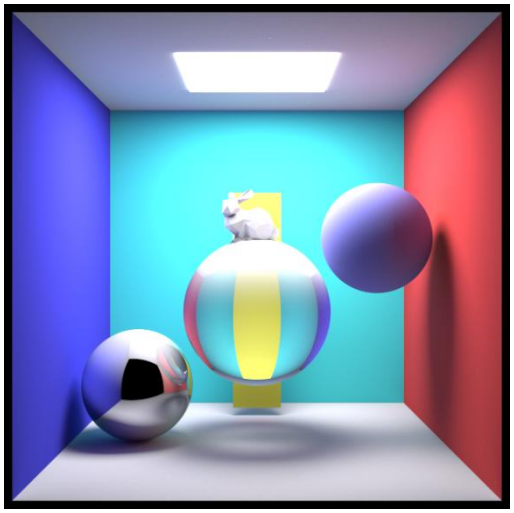
使用前	**使用后**
16spp 用时：14h28min22s	192spp 用时：9h 51min 48s

加速比：

17.61

验收前实现的功能：（按从基本到高阶的顺序）

1. 理想折射，反射，漫反射



2. 面光源（见上图）
3. NEE（见代码逻辑）
4. cosine-weighted 采样（见代码逻辑）
5. 抗锯齿（见代码逻辑）
6. openmp 加速（见代码逻辑）

原理和代码逻辑

宏观

原理部分

本次大作业归根结底只是在做一件事，就是用蒙特卡洛方法求解渲染方程：

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega L_i(p,\omega_i) f_r(p,\omega_i,\omega_r)(n\cdot\omega_i)\mathrm{d}\omega_i$$

实现时需要小心 pdf 的计算，尤其是实现复杂采样和 MIS 时

代码部分

验收前的即为`legacy method`，这里我通过设置参数影响下一次光线追踪的方向来近似模拟 glossy 的效果（效果不佳，后采用 Disney Principal BRDF 解决）。

具体来说，通过为材料设置 roughness 的参数，将光线应有的方向（折射，反射按照物理规律的方向）和一个随机的方向进行插值，则参数设的大，趋于漫反射；参数小则接近理想反射、折射。

结果效果不佳（见报告开始对比）

验收后的部分加入 BRDF 的计算，同时实现多个效果。

微观

下面分点描述：

1. Whitted-Style

没有使用蒙特卡罗方法估计，投射一条光线，打到漫反射就返回，依赖冯模型计算颜色（经验模型）。其他材质按物理规律改变光线方向，可以达到镜面、折射等效果。

2. 面光源

打到光源就返回颜色（实现 nee 之前），和普通面片无区别。到 nee 时，因为直接光照部分已经在前面算过了，直接返回即可（即间接光照的贡献程度为 `0`）。

值得注意的是对光源的采样，应使用合适的均匀点采样方法。

`include/shape.h`

```
// Light Sample for Next Event Estimation
LightSampleResult sampleLight() const {
    float r1 = randf();
    float r2 = randf();
    LightSampleResult res;
    //利用重心坐标系的三角形随机均匀点采样
    res.origin = (1.0f - sqrt(r1)) * p1 + (sqrt(r1) * (1.0f - r2)) * p2 + (sqrt(r1) * r2) * p3;
    res.color = material.color;
    res.normal = material.normal;
    res.pdf = 1.0f / (0.5f * length(cross(p2 - p1, p3 - p1)));
    res.erate = material.erate;
    return res;
}
```

3. NEE

本质上就是同时对光源进行采样，实现的时候要注意将光源的 `pdf`。

Sampling the Light

Then we can rewrite the rendering equation as

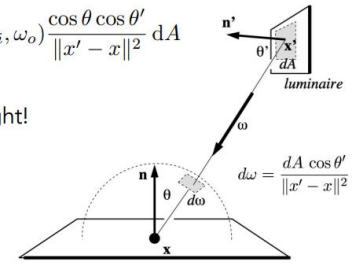
$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\cos\theta\,\mathrm{d}\omega_i$$
$$= \int_A L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\frac{\cos\theta\cos\theta'}{\|x'-x\|^2}\,\mathrm{d}A$$

Now an integration on the light!

Monte Carlo integration:

"f(x)": everything inside

Pdf: 1 / A



对应代码：

Include/renderer.h

```
    HitResult shadowRes = shoot(shapes, shadowRay);
    if (abs(shadowRes.distance - distance) < 0.01f && shadowRes.material.isEmissive) {
        float cosTheta = dot(shadowRay.direction, normal);
        float cosTheta2 = dot(shadowRay.direction, lsr.normal);
        if (cosTheta * cosTheta2 > 0) {
            float G = cosTheta * cosTheta2 / (distance * distance);
            float pdf = lsr.pdf;
            vec3 fr;
            if(legacy) {
                // for legacy method, we think every thing as
                // a mix of ambient and reflect(except reflect part)
                // we use where the ray goes to simulate the material
                // and we think every thing goes on well with Lambert
                fr = hitColor * PI_INV;
            } else {
                fr = BRDF_Evaluate(-indirect, material.normal, shadowRay.direction, material, hitColor);
            }
            float weight = 1.0 / pdf;
            color += fr * G * weight * lsr.erate;
        }
    }
}
```

4. cosine-weighted 采样

推导在注释中

Include/renderer.h

```
if(legacy) {
    // legacy方案不是蒙特卡洛，只是用改变方向的方式模仿BRDF的效果
    // 1/pdf is always 2*PI thanks to our living in a 3D world
    // Lambert漫反射取rou为 1
    // fr = rou / PI
    // cos-weighted importance sampling pdf = cos / PI
    // so, fr * cos / pdf = rou
    if (r < res.material.specularRate) {
        vec3 ref = normalize(reflect(ray.direction, res.material.normal));
        nextRay.direction = ref;
        color += pathTracingNEE(shapes, nextRay, depth + 1, lights, res.material, res.hitColor) / P;
    } else if (res.material.specularRate <= r && r <= res.material.refractRate) {
        vec3 ref = normalize(refract(ray.direction, res.material.normal, float(res.material.refractRate)));
        nextRay.direction = mix(ref, -nextRay.direction, res.material.refractRoughness);
        color += pathTracingNEE(shapes, nextRay, depth + 1, lights, res.material, res.hitColor) / P;
    } else {
        vec3 srcColor = res.hitColor;
        vec3 ptColor = pathTracingNEE(shapes, nextRay, depth + 1, lights, res.material, res.hitColor);
        color += ptColor * srcColor / P;
    }
}
```

5. 抗锯齿

在 render 的循环里，通过对光线方向增加小扰动完成抗锯齿。

Include/renderer.h

```
//from smallpt
//tent filter
double r1 = 2.0 * randf();
r1 = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
double r2 = 2.0 * randf();
r2 = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);

//抗锯齿
x += (sx + 0.5 + r1) / double(width);
y -= (sy + 0.5 + r2) / double(height);;

Ray ray;
camera.castRay(vec2(x, y), ray);
```

6．openmp 加速

调库即可

7．Glossy

基于 Disney BRDF Principle 实现

$$f(\boldsymbol{l},\boldsymbol{v}) = \text{diffuse} + \frac{D(\theta_h)F(\theta_d)G(\theta_l,\theta_v)}{4\cos\theta_l\cos\theta_v}$$

其中，漫反射如下计算：

$$f_d = \frac{baseColor}{\pi}\left(1+(F_{D90}-1)(1-\cos\theta_l)^5\right)\left(1+(F_{D90}-1)(1-\cos\theta_v)^5\right)$$

where

$$F_{D90} = 0.5 + 2roughness\cos^2\theta_d$$

对于 D 项法线分布项：

$$D_{\text{GTR}} = c/(\alpha^2\cos^2\theta_h + \sin^2\theta_h)^\gamma$$

alpha 为粗糙度

其余两项更为复杂，请参见论文。

实现方法基本上就是把文章里的公式打一遍，代码位于 include/material.h 里

8. 纹理贴图

纹理贴图的思路十分简单，就是在渲染过程中间加了一层，本来要返回材质颜色的，现在直接返回对应纹理的颜色。

实现上，主要分为两块，一部分是计算交点的 uv 坐标，另一部分是从图片读取 uv 坐标的值。

9. 法向贴图

和纹理贴图基本一致，只用知道存在一个转换关系，因为图片的分量只能在[0，1]，而向量可以在[-1,1]。

10. 法向插值

按照交点的 uv 坐标取合适的法向。在构造三角形面片时应该存一个法向量数组，然后每一个面片加入时，对应的顶点法向量也加上此向量， normalize 以后即可达到顶点的法向量。

Include/mesh.h

L183

```
if(smooth) {
    vec3 norm = normalize(
            cross(vertices[trig[1]] - vertices[trig[0]], vertices[trig[2]] - vertices[trig[0]]));
    n[trig[0]] += norm;
    n[trig[1]] += norm;
    n[trig[2]] += norm;

    t.push_back(Triangle(vertices[trig[0]], vertices[trig[1]], vertices[trig[2]], m, vec3(3.0f, 3.0f, 3.0f), DIFF,
                         trig[0], trig[1], trig[2], smooth));
}
```

L202

```
if(smooth) {
    vec3 norm = normalize(
            cross(vertices[trig[1]] - vertices[trig[0]], vertices[trig[2]] - vertices[trig[0]]));
    n[trig[0]] += norm;
    n[trig[1]] += norm;
    n[trig[2]] += norm;

    t.push_back(Triangle(vertices[trig[0]], vertices[trig[1]], vertices[trig[2]], m, vec3(3.0f, 3.0f, 3.0f), DIFF,
                    trig[0], trig[1], trig[2], smooth));
}
```

11. 运动模糊

本质上是模拟快门，给每条光线加上时间参数就行了，物体求交时只用注意光线是和当前时刻物体求交的即可。

Include/shape.h

L171

```
vec3 get_O(float time){
    if(time0 == time1) return O1;
    return O1 + (time - time0) / (time1 - time0) * (O_prime - O1);
}
```

将正常球的 get_O()进行改写就可以实现。

12. 景深

本质上是模拟凸透镜成像，使用 1/u + 1/v = 1/f 即可算出像点的位置，然后从凸透镜上的点打到像点的那一束光就是需要投射的光线。剩下的部分与普通的光线追踪并无二至。

Include/camera.h

L 82

```
ThinLensCamera(const vec3& position, const vec3& forward = vec3(0,0,-1), float focalLength = 2.9f, float FOV = 0.5f * PI,
               float fNumber = 22.0f, float t0 = 0, float t1 = 0): Camera(position, forward, t0, t1) {
    // compute focal length from FOV
    if(FOV == 0.0f) {
        this->focalLength = focalLength;
    } else {
        this->focalLength = 1.0f / std::tan(0.5f * FOV);
    }
    // compute lens radius from F-number 光圈数
    lensRadius = 2.0f * focalLength / fNumber;

    b = 10000.0f; //物距
    a = 1.0f / (1.0f / focalLength - 1.0f / b); // 像距 1/u + 1/v = 1/f, u = a, v = b
}


// focus at specified position
void focus(const vec3& p) {
    // b should be much larger than a for a normal camera
    // with means a may change little when b changes
    // So, we can calculate this way
    b = dot(p - position, forward) - a; // position is where the eye is
    a = 1.0f / (1.0f / focalLength - 1.0f / b);
}


bool castRay(const vec2& uv, Ray& ray) const override {
    float t = time0 + randf() * (time1 - time0);

    const vec3 sensorPos = position + uv[0] * right + uv[1] * up;
    const vec3 lensCenter = position + a * forward;

    // sample point on lens
    float pdf_area;
    const vec2 pLens2D = sampleDisk(lensRadius, pdf_area);
    const vec3 pLens = lensCenter + pLens2D[0] * right + pLens2D[1] * up;
    vec3 sensorToLens = normalize(pLens - sensorPos);

    // find intersection point with object plane
    const vec3 sensorToLensCenter = normalize(lensCenter - sensorPos);
    const vec3 pObject = sensorPos + ((a + b) / dot(sensorToLensCenter, forward)) * sensorToLensCenter;

    ray = Ray(pLens, normalize(pObject - pLens), t);
    return true;
}
```

13. 参数曲面解析法求交

使用 Newton 迭代法对曲面进行求交，牛顿法实现相对容易，但精度可能存在缺陷。Newton 法的思想是一步步逼近答案，到一定精度就终止。由于函数较长，展示最为关键的地方，即判断是否产生交点。

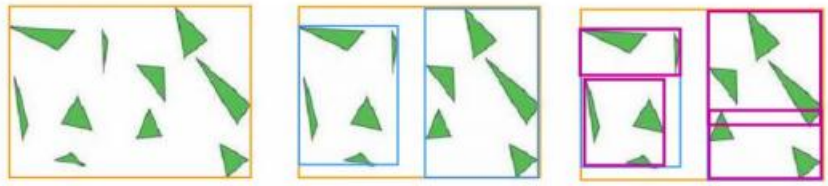
Include/revsurface.h

L42

```
int i;
for (i = 0; i < NEWTON_STEPS; ++i) {
    // Newton's method
    // 求解核心
    if (rou < 0.0) rou += 2 * PI;
    if (rou >= 2 * PI) rou = fmod(rou, 2 * PI);

    if (mu <= 0) mu = FLT_EPSILON;
    if (mu >= 1) mu = 1.0 - FLT_EPSILON;

    point = getPoint(rou, mu, drou, dmu);

    vec3 f = r.startPoint + r.direction * t - point;
    float dist = glm::length(f);
    normal = cross(dmu, drou);
    if (dist < NEWTON_EPS) break;

    float D = dot(r.direction, normal);
    // 迭代
    t -= dot(dmu, cross(drou, f)) / D;
    mu -= dot(r.direction, cross(drou, f)) / D;
    rou += dot(r.direction, cross(dmu, f)) / D;
}
```

14. 复杂网格模型及其求交加速

使用了课上介绍的 Bounding Volume Hierarchy 方式来进行构建。



在数据结构课程中实现过类似数据结构

在 include/bvh.h 中

加速主要是包围盒避免了一些三角形被计算，节约了求交时间。

实测结果为快 17 倍左右。

参考代码

1. Smallpt

参考 smallpt 实现了俄罗斯轮盘赌、抗锯齿、tent filter 以及 cos weighted 采样的功能

2. 往届学长代码

https://github.com/Guangxuan-Xiao/THU-Computer-Graphics-2020 by Guangxuan Xiao

参考此仓库实现了参数曲面解析求交。

3. disney.brdf 即官方实现

结果