

SPPM 随机渐进光子映射渲染器报告

本项目实现了一个基于SPPM随机渐进光子映射算法的渲染器并完成了一系列效果。报告将按照如下的顺序展示相应地结果和实现：

- SPPM核心
- BVH和HashGrid加速
- 物体和基本图元
- 材质
- 相机
- Mesh
- 场景
- Bezier曲面
- 其他附加功能
- 效果图

其中附加功能将根据实现的对象在不同的部分进行穿插。

SPPM算法核心

这一部分在 `include/ShadeRender.hh` 的 `Camera` 类中完成了实现，实现的是随机渐进光子映射的算法核心，也就是SPPM一轮的两个部分：光线追踪和光子映射。首先进行光线追踪，将采样点 `HPoint` 结构体（定义于 `ShadeTraceRecord.hh`）中通过光线追踪投射到漫反射界面上，然后运行光子映射，将光子按照采样半径采样到采样点上，并最后根据采样点估计像素的颜色。这将包含两个部分：启动光线追踪和光子映射的 `Render` 和进行光线追踪/光子映射的 `Trace`，首先我们来看渲染的部分：

```
void renderScene(Scene &scene, int nround, int samps) {
    photonRand.resize(samps);
    for (auto i = 0; i < samps; ++i)
        photonRand[i].setState(rand(), rand());
    root = scene.buildScene();

    for (int nr = 0; nr < nround; ++nr) {
        printf("> doing round %d\n", nr);
        // trace eye rays and store measurement points
        for (int y = 0; y < h; y++) {
            fprintf(stderr, "\rHitPointPass %5.2f%%", 100.0 * y / (h - 1));
#pragma omp parallel for schedule(guided)
            for (int x = 0; x < w; x++) {
                int pixel_index = x + y * w;
                RandEngine &gen = pixelRand[pixel_index];
                auto u = double(x + gen()) / double(w);
                auto v = double(y + gen()) / double(h);
                trace(getRay(u, 1 - v, gen), 0, &grid[pixel_index],
                      Vector3d(0, 0, 0), Vector3d(1, 1, 1), gen);
            }
        }
        fprintf(stderr, "\n");
        fprintf(stderr, "> building hash grid\n");

        // build the hash table over the measurement points
        grid.buildHashGrid();
        for (int i = 0; i < samps; i++) {
```

```

        double p = 100. * (i + 1) / samps;
        fprintf(stderr, "\rPhotonPass %5.2f%%", p);
        int m = 1000 * i;
        Ray r;
        Vector3d f;
#pragma omp parallel for
        for (int j = 0; j < 1000; j++) {
            scene.generateRay(r, f, photonRand[i]);
            trace(r, 0, nullptr, f, Vector3d(1, 1, 1), photonRand[i]);
        }
    }
    fprintf(stderr, "\rfinish\n");

    // density estimation
    for (int i = 0; i < w * h; ++i) {
        c[i] =
            c[i] + clamp(Vector3d(grid[i].flux * (1.0 / (PI * grid[i].r2 *
samps * 1000.0))));
    }

    if (nr % 100 == 0 && nr > 0) {
        saveBackup(nr);
    }
}

for (int i = 0; i < w * h; ++i) {
    c[i] /= nrround;
}
}

```

以上的代码是渲染部分的核心，可以看到首先进行的是光线追踪，通过调用 `trace` 函数并提供对应的采样点 `&grid[pixel_index]` 来完成的，而上面的随机数是抖动采样的实现，通过抖动采样，可以完成最终图像上的抗锯齿效果（SPPM自带超采样），我为每个像素点都分配了一个随机数发生器来保证最后产生的结果尽可能不会出现奇怪的条纹。一旦完成了光线追踪的部分，就将会进行加速结构 `HashGrid` 的构建，这将会在下一个部分进行详细的介绍。在光子映射阶段，将会选择发射光子到场景中，并对采样点进行通量的累加的方法进行光子映射。最后，将每一轮的结果截断，然后评估多轮的平均结果，就可以得到最后的图像了。

接下来我们看 `trace` 函数，这个函数同时肩负了光子映射和光线追踪的责任，在接受一个 `HPoint` 的时候是光线追踪，而接受的是 `nullptr` 的时候则是光子映射。我们来看一下这个函数：

```

void trace(const Ray &r, int dpt, HPoint *m, const Vector3d &f1,
           const Vector3d &adj, RandEngine &gen) {
    dpt++;
    Hitrec h;
    if (!root->intersect(r, h, 1e-4) || (dpt >= 10))
        return;
    const Material *mat = h.mat;
    double t = h.t;
    Vector3d n = h.norm;
    Vector3d x = r.o + r.d * t, f = mat->query(h.uv);
    Vector3d nl = n.dot(r.d) < 0 ? n : n * -1;
    double p = f.x() > f.y() && f.x() > f.z() ? f.x()
                                                : f.y() > f.z() ? f.y()
                                                : f.z();
#define GETBRDF(prop) PRE_BRDFS[mat->brdf].prop

```

```

auto spec = GETBRDF(spec), diff = GETBRDF(diff), refr = GETBRDF(refr);
auto s = spec + diff + refr;
auto act = gen(); /* s;

if (diff > 0 && act <= diff && act > 0) {
    double r1 = 2. * PI * gen(), r2 = gen();
    Vector3d w = n1, u = ((fabs(w.x()) > .1 ? Vector3d(0, 1, 0)
                           : Vector3d(1, 0, 0)) %
                           w)
                           .normalized();

Vector3d v = w % u;

if (m) {
    m->f = f * adj;
    m->pos = x;
    m->norm = n;
} else {
    auto a = gen();
    if (a <= GETBRDF(rhos)) {
        double r2s = pow(r2, 1.0 / (GETBRDF(phongs) + 1));
        auto d = (u * cos(r1) * r2s + v * sin(r1) * r2s + w * sqrt(1 - r2))
                  .normalized();
        trace(Ray(x, d), dpt, m, (f * fl), adj, gen);
    } else {
        a -= GETBRDF(rhos);
        auto &hp = grid[x];
        for (auto hitpoint : hp) {
            Vector3d v = hitpoint->pos - x;
            if ((hitpoint->norm.dot(n) > 1e-3) &&
                (v.dot(v) <= hitpoint->r2)) {
                double g =
                    (hitpoint->n * ALPHA + ALPHA) / (hitpoint->n * ALPHA + 1.0);
                hitpoint->r2 = hitpoint->r2 * g;
                hitpoint->n++;
                hitpoint->flux =
                    (hitpoint->flux + (hitpoint->f * fl) * (1. / PI)) * g;
            }
        }
        if (a <= GETBRDF(rhoD)) {
            double r2s = sqrt(r2);
            auto d =
                (u * cos(r1) * r2s + v * sin(r1) * r2s + w * sqrt(1 - r2))
                  .normalized();
            trace(Ray(x, d), dpt, m, (f * fl) /* (1. / GETBRDF(rhoD))/, adj,
                                              gen);
        }
    }
}
act -= diff;

if (spec > 0 && act <= spec && act > 0) {
    trace(Ray(x, r.d - n * 2.0 * n.dot(r.d)), dpt, m, f * fl, f * adj, gen);
}
act -= spec;

if (refr > 0 && act <= refr && act > 0) {
    Ray lr(x, r.d - n * 2.0 * n.dot(r.d));
}

```

```

        bool into = (n.dot(nl) > 0.0);
        double nc = 1.0, nt = GETBRDF(refN), nnt = into ? nc / nt : nt / nc,
               ddn = r.d.dot(nl), cos2t;

        // total internal reflection
        if ((cos2t = 1 - nnt * nnt * (1 - ddn * ddn)) < 0)
            return trace(lr, dpt, m, fl, adj, gen);

        Vector3d td =
            (r.d * nnt - n * ((into ? 1 : -1) * (ddn * nnt + sqrt(cos2t))))
                .normalized();
        double a = nt - nc, b = nt + nc, R0 = a * a / (b * b),
               c = 1 - (into ? -ddn : td.dot(n));
        double Re = R0 + (1 - R0) * c * c * c * c * c * c, P = Re;
        Ray rr(x, td);
        Vector3d fa = f * adj;
        if (m) {
            trace(lr, dpt, m, fl, fa * Re, gen);
            trace(rr, dpt, m, fl, fa * (1.0 - Re), gen);
        } else {
            (gen() < P) ? trace(lr, dpt, m, fl, fa, gen)
                          : trace(rr, dpt, m, fl, fa, gen);
        }
    }

    if (mat->brdf == BRDF::SCATTER) {
        trace(Ray(r.o, unitSphereRandom(gen)), dpt, m, fl, f, gen);
    }
}

```

简单来说就是根据Phong模型，我们对不同的BRDF需要采取不同的动作，而选择什么样的动作则根据俄罗斯轮盘赌来实现。比较需要注意的有两个地方，第一个地方是在漫反射的介质表面，如果是光线追踪阶段，就结束了算法，记录采样点的位置了，如果是光子映射，则根据Phong模型，有可能是吸收（给采样点累加通量），也有可能是漫反射。另一个需要注意的地方是在折射的地方采用了Schlick近似(r^5)来完成菲涅尔定律的复杂计算，同时采用轮盘赌的方式决定下一束光线是折射还是反射。这都让最后的结果得以最真实的方式呈现。

BVH和HashGrid加速

由于SPPM需要同时进行光线追踪和光子映射，所以需要对求交和采样的过程进行加速。其中求交加速选择了较为传统的BVH方式，而采样则是抛弃了KD-Tree而选择了HashGrid的方式，这是因为KD-Tree建立速度很慢，同时查询也比较慢，相比之下HashGrid在预先已经分配好充足的内存后建立速度极快，而且查询命中效率也是在 $O(1)$ 的，是极佳的选择。

BVH

要建立BVH首先需要实现包围盒。这就是AABB，在ShadeObject.hh中完成了实现，这个包围盒同时在HashGrid中用到了，所有有相关的方法，但是最核心的求交选择的是：

```

bool intersect(const Ray &r, double tMin) const {
    Vector3d dirfrac = r.id;
    double t1 = (min.x() - r.o.x()) * dirfrac.x();
    double t2 = (max.x() - r.o.x()) * dirfrac.x();
    double t3 = (min.y() - r.o.y()) * dirfrac.y();
    double t4 = (max.y() - r.o.y()) * dirfrac.y();
    double t5 = (min.z() - r.o.z()) * dirfrac.z();

```

```

        double t6 = (max.z() - r.o.z()) * dirfrac.z();
        double tmin = std::max(std::max(std::min(t1, t2), std::min(t3, t4)),
                               std::min(t5, t6));
        double tmax = std::min(std::min(std::max(t1, t2), std::max(t3, t4)),
                               std::max(t5, t6));
        if (tmax < 0)
            return false;
        if (tmin > tmax)
            return false;
        return true;
    }
}

```

也就是在每个面进行求交，然后得到 `tmin` 和 `tmax`，注意由于需要反复进行包围盒求交，所以需要反复用到光线方向的倒数，这就被直接保存在了光线中来进行加速了。

有了包围盒之后，可以为基类指定 `AABB boundingBox() const` 方法来获取对应物体的包围盒，从而用于构建BVH了。构建的方法很简单：

```

BVHNode(object **objs, int n, int axis = 0) : object(nullptr) {
    if (n == 1) {
        left = right = objs[0];
        box = objs[0]->boudingBox();
        return;
    } else {
        switch (axis) {
        case 0:
            std::sort(objs, objs + n, sortX);
            break;
        case 1:
            std::sort(objs, objs + n, sortY);
            break;
        case 2:
            std::sort(objs, objs + n, sortZ);
            break;
        }
        left = new BVHNode(objs, n / 2, (axis + 1) % 3);
        right = new BVHNode(objs + n / 2, n - n / 2, (axis + 1) % 3);
        box = left->boudingBox() + right->boudingBox();
    }
}

```

首先按照某一轴来进行排序，然后递归的构建子树，最后将两个子树的包围盒包起来作为自己节点的包围盒。而对BVH的求交也是类似方式进行的：

```

bool intersect(const Ray &ray, Hitrec &h,
               double tmin = 1e-4) const override {
    if (box.intersect(ray, tmin)) {
        if (left == right)
            return left->intersect(ray, h, tmin);
        Hitrec hLeft, hRight;
        bool isLeft = left->intersect(ray, hLeft, tmin);
        bool isRight = right->intersect(ray, hRight, tmin);
        if (isLeft && isRight)
            h = hLeft.t < hRight.t ? hLeft : hRight;
        else if (isLeft)
            h = hLeft;
    }
}

```

```

        else if (isRight)
            h = hRight;
        return isLeft || isRight;
    }
    return false;
}

```

首先对自己的包围盒进行求交，如果没有交点的话就直接剪枝了，否则对两个孩子进行求交，如果都命中就比较谁的t小，否则返回唯一发生相交的分支。

HashGrid

HashGrid是一种空间哈希网格，用于光子映射的采样加速，通过将采样点以哈希表的方式组织起来完成的。这在 `ShadeAccelerate.hh` 中完成了实现：

```

void buildHashGrid() {
    hpbbox.reset();
    for (auto &hp : hitpoints)
        hpbbox.fit(hp.pos);
    Vector3d ssize = hpbbox.max - hpbbox.min;
    double irad =
        ((ssize.x() + ssize.y() + ssize.z()) / 3.0) / ((w + h) / 2.0) * 2.0;
    hpbbox.reset();
    int vphoton = 0;
    for (auto &hp : hitpoints) {
        hp.r2 = irad * irad;
        hp.n = 0;
        hp.flux = Vector3d();
        vphoton++;
        hpbbox.fit(hp.pos - irad);
        hpbbox.fit(hp.pos + irad);
    }

    // make each grid cell two times larger than the initial radius
    hashS = 1.0 / (irad * 2.0);
    numHash = vphoton;

    // build the hash table
    for (uint32_t i = 0; i < numHash; i++)
        hashGrid[i].clear();
    for (auto &hp : hitpoints) {
        Vector3d BMin = ((hp.pos - irad) - hpbbox.min) * hashS;
        Vector3d BMax = ((hp.pos + irad) - hpbbox.min) * hashS;
        for (int iz = abs(int(BMin.z())); iz <= abs(int(BMax.z())); iz++) {
            for (int iy = abs(int(BMin.y())); iy <= abs(int(BMax.y())); iy++) {
                for (int ix = abs(int(BMin.x())); ix <= abs(int(BMax.x())); ix++) {
                    int hv = hash(ix, iy, iz);
                    hashGrid[hv].push_back(&hp);
                }
            }
        }
    }
}

```

在建立的第一步是确定初始半径，这个可以不用管，采用了一种启发式的方法而不是固定半径，从而适应了不同的场景尺度大小。之后则需要确定整体的大小（采样点+采样球），然后根据大小将x,y,z坐标离散化，并通过空间哈希函数分配到对应的链表里面。空间哈希函数的选取如下：

```
uint32_t hash(int ix, int iy, int iz) {
    return (uint32_t)((ix * 73856093) ^ (iy * 19349663) ^ (iz * 83492791)) %
        numHash;
}
```

这个函数是参考别人的实现的，采用了一系列的魔数，但是最后的效果不错。

物体和基本图元

基类

对于所有的物体来说都继承于这个基类，在 `shadeobject.hh` 中定义了基类的样子：

```
class Object {
    friend class Light;

protected:
    Material *mat;

public:
    Object(Material *mat) : mat(mat) {}

    virtual bool intersect(const Ray &ray, Hitrec &h, double tmin) const = 0;

    virtual AABB boundingBox() const = 0;

    virtual void emit(Ray &ray, Vector3d &f, RandEngine &rand) const {}

    virtual ~Object() {}

};
```

除了包含了大家都有的材质以外，还需要提供其他的几个功能：求交，求包围盒和发射光子的功能。其中发射光子是用于当物体作为灯的时候要发射光子用的。接下来介绍几个常见的基本图元，主要看求交函数。

球体

球体的求交是通过解方程计算判别式完成的：

```
bool intersect(const Ray &r, Hitrec &h, double tmin = 1e-4) const override {
    Vector3d op = p - r.o;
    double t, b = op.dot(r.d), det = b * b - op.dot(op) + rad * rad;
    if (det < 0) {
        return false;
    } else {
        det = sqrt(det);
    }
    if ((t = b - det) > tmin) {
        auto pos = r.o + r.d * t;
        auto norm = (pos - p).normalized();
        auto uv = mat->reqUV ? getUV(norm) : Vector3d();
        h.setHit(t, norm.dot(r.d) < 0 ? norm : -norm, uv, mat);
    }
}
```

```

        return true;
    } else if ((t = b + det) > tmin) {
        auto pos = r.o + r.d * t;
        auto norm = (pos - p).normalized();
        auto uv = mat->reqUV ? getUV(norm) : Vector3d();
        h.setHit(t, norm.dot(r.d) < 0 ? norm : -norm, uv, mat);
        return true;
    }
    return false;
}

```

大概就是将方程 $r = o + t \bullet d$ 带入到 $(x - o)^2 = r^2$ 中然后解二元一次方程得到两个 t 的可能，然后根据最后的判别式判断是相交还是没有相交。然后得到交点之后记录相关的数据。除了这一点之外，由于采用了UV贴图，还需要计算球体表面的UV坐标，这是通过球坐标系转化为UV坐标系完成的：

```

Vector3d getUV(const Vector3d &pos) const {
    double u = 1 - (atan2(pos.z(), pos.x()) + PI) / (2 * PI);
    double v = pos.y() * 0.5 + 0.5;
    return Vector3d(u, v, 0);
}

```

这个转化方法是固定且统一的，并没有什么好说的。

三角形

三角形的求交也是通过联立解方程得到的：

```

bool intersect(const Ray &r, Hitrec &h, double tmin) const override {
    auto orig = r.o, dir = r.d;
    auto v0 = vertices[0], v1 = vertices[1], v2 = vertices[2];
    vector3d e1 = v1 - v0, e2 = v2 - v0;
    auto p = dir % e2;
    auto det = e1.dot(p), abs_det = fabs(det);
    vector3d t = det > 0 ? orig - v0 : v0 - orig;
    if (abs_det < 1e-6)
        return false;
    auto udet = t.dot(p);
    if (udet < 0.0f || udet > abs_det)
        return false;
    auto q = t % e1;
    auto vdet = dir.dot(q);
    if (vdet < 0.0f || udet + vdet > abs_det)
        return false;
    auto time = fabs(e2.dot(q));
    time /= abs_det;
    if (time < tmin)
        return false;
    auto uv = mat->reqUV ? getUV(r.o + r.d * time) : Vector3d();
    h.setHit(time, norm, uv, mat);
    return true;
}

```

由于是线性方程组，首先可以通过克莱姆法则来判断是否存在这样的交点，然后得到局部坐标系的解，换算得到 t ，具体来说就是考虑方程：

$$o + d \bullet t = (1 - u - v)V_0 + uV_1 + vV_2$$

然后按照线性方程组的方法进行求解即可

接下来也需要计算UV坐标，计算的方法是：

```
vector3d getUV(const vector3d &pos) const {
    auto toP = (pos - vertices[1]);
    auto l = toP.norm();
    auto x = (vertices[0] - vertices[1]);
    auto y = (vertices[2] - vertices[1]);
    auto x1 = x.norm(), y1 = y.norm();
    auto u = toP.dot(x) / (x1 * x1);
    auto v = toP.dot(y) / (y1 * y1);
    return Vector3d(u, v, batchId);
}
```

这里选定了 V_1 作为基准的坐标，计算三角形作为一个仿射变换的逆变换得到原来的UV坐标即可。

法线插值的三角形

三角形的法线可以是通过三个顶点计算出来，也可以根据三个顶点的法线插值得到：

```
vector3d interplotNorm(const vector3d &p) const {
    auto a = vertices[0], b = vertices[1], c = vertices[2];
    auto alpha = ((p.x() - b.x()) * (c.y() - b.y()) +
                  (p.y() - b.y()) * (c.x() - b.x())) /
                 ((a.x() - b.x()) * (c.y() - b.y()) +
                  (a.y() - b.y()) * (c.x() - b.x()));
    auto beta = ((p.x() - c.x()) * (a.y() - c.y()) +
                 (p.y() - c.y()) * (a.x() - c.x())) /
                 ((b.x() - c.x()) * (a.y() - c.y()) +
                  (b.y() - c.y()) * (a.x() - c.x()));
    auto gamma = 1 - alpha - beta;
    return alpha * norm[0] + beta * norm[1] + gamma * norm[2];
}
```

在给定三个顶点的法向之后，可以计算相交点的重心坐标，然后根据重心坐标插值得到法向。这里重心坐标的计算采用了公式法的方法直接计算得到。加入了法向插值和没有加入法向插值的效果会在后面进行展示。

空间变换

和之前的小作业中实现的一样，实现了对物体的空间变换的求交：

```
bool intersect(const Ray &ray, Hitrec &h, double tmin) const override {
    auto trOrg = transPoint(trans, ray.o);
    auto trDir = transDir(trans, ray.d).normalized();
    bool inter = baseObject->intersect(Ray(trOrg, trDir), h, tmin);
    if (inter)
        h.setHit(h.t, transDir(trans.transpose(), h.norm).normalized(), h.uv,
                 h.mat);
    return inter;
}
```

这就使得我们可以包装物体来完成物体的空间中的变换了。

材质

之前提到过实现了UV贴图，这是在材质中实现的。材质包括了两部分，BRDF和颜色。这都是在 `shadeMaterial.hh` 中实现的。

BRDF

记录了Phong模型所需要的所有BRDF参数，从而可以模拟不同的情况：

```
struct BRDF {
    double spec, diff, refr, rhoD, rhoS, phongS, refN;

    BRDF() {}

    constexpr BRDF(double spec, double diff, double refr, double rhoD,
                  double rhoS, double phongS, double refN)
        : spec(spec), diff(diff), refr(refr), rhoD(rhoD), rhoS(rhoS),
          phongS(phongS), refN(refN) {}

    enum DefaultBRDF {
        DIFFUSE,
        MIRROR,
        GLASS,
        LIGHT,
        MARBLE,
        FLOOR,
        WALL,
        DESK,
        STANFORD_MODEL,
        WATER,
        TEAPOT,
        TRUEGLASS,
        SCATTER,
        METAL,
    };
};
```

其中提供了一系列的预设的BRDF可以调用。比如METAL材质就模拟了带大多数反射和一部分散射的界面。

颜色

有两种颜色可以选取，通过调用统一的接口来得到，即 `vector3d query(const vector3d& pos)`，通过UV坐标来获得颜色值。

常量颜色始终返回相同的颜色，而贴图则通过读入的PPM图片来计算UV坐标值对应的像素颜色。注意到传入的 `pos` 实际上是一个三维的点，而UV只用到了其中两维，所以第三维可以用来干嘛的，也就是指定了 `batchId`，我要求一个贴图必须是方形的，如果是多张贴图的话，可以竖着拼接起来，然后通过 `batchId` 指定用的是其中哪一张贴图，方便了比如贴盒子的时候就可以把几个面的贴图合成一张了。

```

vector3d query(const Vector3d &point) const override {
    int batchId = point.z();
    auto x = int(point.x() * w);
    auto y = int(point.y() * w);

    x = x < 0 ? 0 : x >= w ? w - 1 : x;
    y = y < 0 ? 0 : y >= w ? w - 1 : y;
    auto clr = data[x + y * w + batchId * w * w];
    return clr;
}

```

相机

相关代码位于 `ShadeRender.hh` 中的 `Camera` 类

除了普通的相机功能之外，还实现了模拟真实相机的功能，也就是实现了光圈和焦距，这实现起来是比较简单的，因为考虑一个透镜成像的系统，我们不需要真实的模拟一个透镜，而只需要把一个圈放在前面就好了，然后按照圈来随机扰动像素发射的光线，就形成了聚焦和光圈虚化的效果。具体的代码实现如下：

```

Ray getRay(double s, double t, RandEngine &rand) {
    auto rd = lensR * unitSphereRandom(rand);
    auto offset = _u * rd.x() + _v * rd.y();
    return Ray(
        pos + offset,
        (lowerLeft + s * horizon + t * vertical - pos - offset).normalized());
}

```

其中 `lensR` 是光圈决定的，注意到要经过光圈的圆盘的扰动；而发射光线的初始位置是焦距决定的：

```

void configCamera(const Vector3d &pos, const Vector3d &lookAt,
                  const Vector3d &up, double fov, double aperture = 0,
                  double focusDisk = 10.0) {
    lensR = aperture / 2;
    auto ratio = double(w) / double(h);
    this->pos = pos;
    auto theta = fov * PI / 180;
    auto halfH = tan(theta / 2);
    auto halfW = ratio * halfH;
    _w = (pos - lookAt).normalized();
    _u = up.cross(_w).normalized();
    _v = _w.cross(_u).normalized();
    lowerLeft = pos - halfW * focusDisk * _u - halfH * focusDisk * _v -
                focusDisk * _w;
    horizon = 2 * halfW * _u * focusDisk;
    vertical = 2 * halfH * _v * focusDisk;
}

```

Mesh

总共实现了三种 `Mesh`：普通的 `Mesh`，法线插值 `Mesh` 和 Bezier 曲线 `Mesh(plt)`，分别进行介绍，具体代码都位于 `ShadeMesh.hh` 中：

Mesh

简单来说就是一堆的三角形构建了一个BVH:

```
Mesh(const char *filename, const Vector3d &pos, const Vector3d &scale,
      const Quaterniond &rot, Material *mat, bool obj = true)
: Object(mat) {
    if (obj) {
        auto f = fopen(filename, "r");
        char leading;
        double t1, t2, t3;
        int i1, i2, i3;
        std::vector<Vector3d> vertices;
        int id = 0;
        while (fscanf(f, "%c", &leading) != EOF) {
            // printf("scanned %c", leading);
            if (leading == 'v') {
                fscanf(f, "%lf %lf %lf", &t1, &t2, &t3);
                fgetc(f);
                vertices.push_back(scale * (rot * Vector3d(t1, t2, t3)) + pos);
            } else if (leading == 'f') {
                fscanf(f, "%d %d %d", &i1, &i2, &i3);
                fgetc(f);
                faces.push_back(new Triangle(vertices[i1 - 1], vertices[i2 - 1],
                                              vertices[i3 - 1], mat, id++));
            } else
                break;
        }
        nfakes = faces.size();
        printf("> Import '%s' :: %d Faces\n", filename, nfakes);
        root = new BVHNode(faces.data(), nfakes);
    } else {
        std::ifstream f(filename);
        f >> nfakes;
        Array<Vector3d> vertices;
        for (int _ = 0; _ < nfakes; ++_) {
            vertices.clear();
            int n, m;
            double a, b, c;
            f >> n >> m;
            for (int i = 0; i <= n; ++i) {
                for (int j = 0; j <= m; ++j) {
                    f >> a >> b >> c;
                    vertices.push_back(scale * (rot * Vector3d(a, b, c)) + pos);
                }
            }
            faces.push_back(
                new Beziersurface(n, m, vertices.data(), pos, scale, mat));
        }
        root = new BVHNode(faces.data(), nfakes);
        printf("> Import '%s' :: %d Faces\n", filename, nfakes);
    }
}
```

其中`isObj`这个参数决定了面片是贝塞尔曲面还是三角形。构建好三角形树之后就相当如BVH的一个包装。

法向插值Mesh

这需要一些技巧，计算每个三角形面片的法向，通过这些法向插值得出来顶点的法向，然后构建法向插值三角形的树：

```
InterpolateMesh(const char *filename, const Vector3d &pos,
                const Vector3d &scale, const Quaterniond &rot, Material *mat)
: Object(mat) {
    struct VerticeNorm {
        Vector3d vertice;
        Vector3d norm;
        int n;

        VerticeNorm(const Vector3d &v) : vertice(v), norm(0, 0, 0), n(0) {}

        void addNorm(const Vector3d &nm) {
            n++;
            norm += nm;
        }

        Vector3d getNorm() const { return norm / double(n); }
    };

    struct TriIndex {
        int i1, i2, i3, id;
        TriIndex(int i1, int i2, int i3, int id)
            : i1(i1), i2(i2), i3(i3), id(id) {}
    };

    auto f = fopen(filename, "r");
    char leading;
    double t1, t2, t3;
    int i1, i2, i3;
    std::vector<VerticeNorm> vertices;
    std::vector<TriIndex> tris;
    int id = 0;
    while (fscanf(f, "%c", &leading) != EOF) {
        // printf("scanned %c", leading);
        if (leading == 'v') {
            fscanf(f, "%lf %lf %lf", &t1, &t2, &t3);
            fgetc(f);
            vertices.push_back(
                VerticeNorm(scale * (rot * Vector3d(t1, t2, t3)) + pos));
        } else if (leading == 'f') {
            fscanf(f, "%d %d %d", &i1, &i2, &i3);
            fgetc(f);
            auto a = vertices[i1 - 1].vertice, b = vertices[i2 - 1].vertice,
                 c = vertices[i3 - 1].vertice;
            Vector3d v0v1 = b - a;
            Vector3d v0v2 = c - a;
            auto norm = (v0v1 % v0v2).normalized();
            vertices[i1 - 1].addNorm(norm);
            vertices[i2 - 1].addNorm(norm);
            vertices[i3 - 1].addNorm(norm);
            tris.push_back(TriIndex(i1, i2, i3, id++));
        } else
            break;
    }
}
```

```

        for (auto t : tris) {
            auto v1 = vertices[t.i1 - 1];
            auto v2 = vertices[t.i2 - 1];
            auto v3 = vertices[t.i3 - 1];
            faces.push_back(new InterpolateTriangle(
                v1.vertex, v2.vertex, v3.vertex, v1.getNorm(), v2.getNorm(),
                v3.getNorm(), mat, t.id));
        }
        nfaces = faces.size();
        printf("> Import '%s' :: %d Faces\n", filename, nfaces);
        root = new BVHNode(faces.data(), nfaces);
    }
}

```

可以看到最开始先计算得到了每个顶点的法向，然后根据这些记录来构建法相插值的三角形，然后建立BVH，最终也相当于BVH的一个包装。

场景

场景的描述位于 `ShadeRender.hh` 中的 `Scene` 类，是BVH的包装，通过对一系列的物体建立BVH来构建场景：

```

class Scene {
    Array<Object *> objects;
    Array<Object *> lights;
    Array<Material *> materials;
    BVHNode *root;

    Vector3d lorg;

public:
    Scene() : root(nullptr) {}

    BVHNode *buildScene() {
        if (root)
            delete root;
        root = new BVHNode(objects.data(), objects.size());
        return root;
    }

    void addObject(Object *obj) { objects.push_back(obj); }

    void addLight(Object *light) {
        objects.push_back(light);
        lights.push_back(light);
    }

    size_t objectSize() const { return objects.size(); }
    size_t lightSize() const { return lights.size(); }

    void configLight(const Vector3d &l) { lorg = l; }

    void generateRay(Ray &pr, Vector3d &f, RandEngine &gen) {
        lights[0] -> emit(pr, f, gen);
        f *= (4 * PI);
    }

    ~Scene() {
}

```

```

        for (auto i : objects)
            delete i;
        for (auto i : materials)
            delete i;
        if (root)
            delete root;
    }
};


```

而具体的读入场景配置文件的代码位于 `shadeMain.cc` 里面，代码比较长就不粘贴在这里了，根据不同的标签来合成不同的物体，插入到场景中。

Bezier曲面

Bezier曲面的求交是一个难点，最终采用了牛顿迭代的方法进行了求交，相关的代码位于 `shadeMesh.hh` 中的 `BezierSurface` 类中。

```

bool intersect(const Ray &ray, Hitrec &h, double tmin) const override {
    double mint = 1e100;
    double resu, resv;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j) {
            Vector3d min = Vector3d(1e100, 1e100, 1e100);
            Vector3d max = min * -1;

            for (int _i = i - 1; _i <= i; ++_i)
                for (int _j = j - 1; _j <= j; ++_j) {
                    Vector3d p = this->p[_i][_j];
                    min = boxMin(min, p);
                    max = boxMax(max, p);
                }

            double t = -1e100;
            if (!(ray.o >= min && ray.o <= max)) { // outside
                if (fabs(ray.d.x()) > 0)
                    t = std::max(t, std::min((min.x() - ray.o.x()) / ray.d.x(),
                                              (max.x() - ray.o.x()) / ray.d.x()));
                if (fabs(ray.d.y()) > 0)
                    t = std::max(t, std::min((min.y() - ray.o.y()) / ray.d.y(),
                                              (max.y() - ray.o.y()) / ray.d.y()));
                if (fabs(ray.d.z()) > 0)
                    t = std::max(t, std::min((min.z() - ray.o.z()) / ray.d.z(),
                                              (max.z() - ray.o.z()) / ray.d.z()));
            }
            if (t < 0)
                continue;
            Vector3d pp = ray.o + ray.d * t;
            if (!(pp >= min && pp <= max))
                continue;
        } else
            t = 0;

        for (int __ = 0; __ < 10; ++__)
            {
                double u = random(), v = random();
                auto x = Vector3d(t, u, v);
                double lambda = 1;
                double last = norm2(F(x, ray));
                for (int _ = 0; _ < 10; ++_) {

```

```

        x = x - d(x, ray);
        double cost = norm2(F(x, ray));
        if (!(norm2(x) <= 1e3)) {
            x.y() = x.z() = 1e100;
            break;
        }
        if (last - cost < 1e-8)
            break;
        last = cost;
    }
    t = x.x();
    u = x.y();
    v = x.z();
    if (0 <= u && u <= 1 && 0 <= v && v <= 1 &&
        norm2(F(x, ray)) < 1e-5) {
        mint = std::min(mint, t);
        resu = u;
        resv = v;
        break;
    }
}
if (mint < 1e100) {
    Vector3d du(0, 0, 0), dv(0, 0, 0);
    for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= m; ++j) {
            du = du - p[i][j] * dB(n, i, resu) * B(m, j, resv);
            dv = dv - p[i][j] * B(n, i, resu) * dB(m, j, resv);
        }
    h.setHit(mint, du.cross(dv), Vector3d(), mat);
    return true;
} else
    return false;
}

```

求交是针对四个控制点组成的小区域分别进行的，首先将收敛点选择包围盒最近的一点，然后开始梯度下降，如果最后得到交点的话，根据Bezier曲面的公式获得法向和坐标。但实际操作中，由于下降收敛速度太慢，并没有获得比较好结果，只有一张比较模糊的图。

其他附加功能

体积雾

具体实现在 `shadeObject.hh` 的 `ConstantMedium` 类中。这也是一个套壳类，通过这个类可以实现不同物体形状的雾化效果，同时可以调节雾的颜色和透明度。具体实现的原理很简单，首先和其中的物体求交，假设在 t 发生了相交，然后考虑设置 $t_{min} = t + \epsilon$ 并再次求交，如果还发生相交的话，说明之前的光线是从外部射入的，那么可以根据两次的相交时间 t_1 和 t_2 计算出光线在物体内穿过的距离，并根据距离折算出光线穿过物体的概率： $p = e^{-\rho \times d}$ ，然后就可以随机是穿过物体还是发生吸收，这样就会得到雾化效果了。具体实现如下：

```

bool intersect(const Ray &ray, Hitrec &h, double tmin) const override {
    Hitrec rec1, rec2;
    if (baseObject->intersect(ray, rec1, tmin)) {
        if (baseObject->intersect(ray, rec2, rec1.t + 0.0001)) {
            double t1 = max(rec1.t, tmin);
            double t2 = rec2.t;

```

```

    if (t1 >= t2)
        return false;
    t1 = max(t1, 0.0);

    double distInside = (t2 - t1) * ray.d.norm();
    double pass = exp(-distInside * density);
    if (engine() > pass) {
        h.setHit(t1, rec1.norm, Vector3d(), mat);
        return true;
    } else {
        return false;
    }
}
}

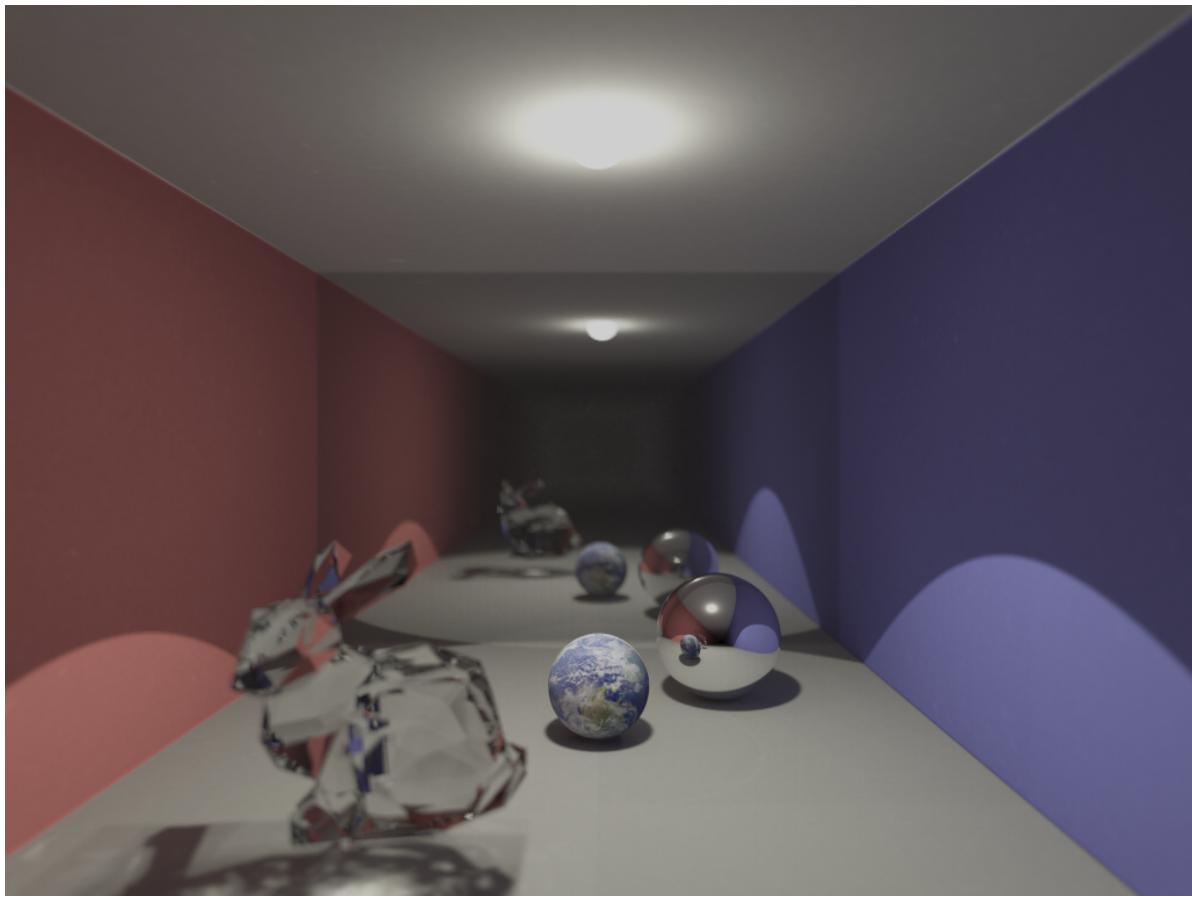
return false;
};

};

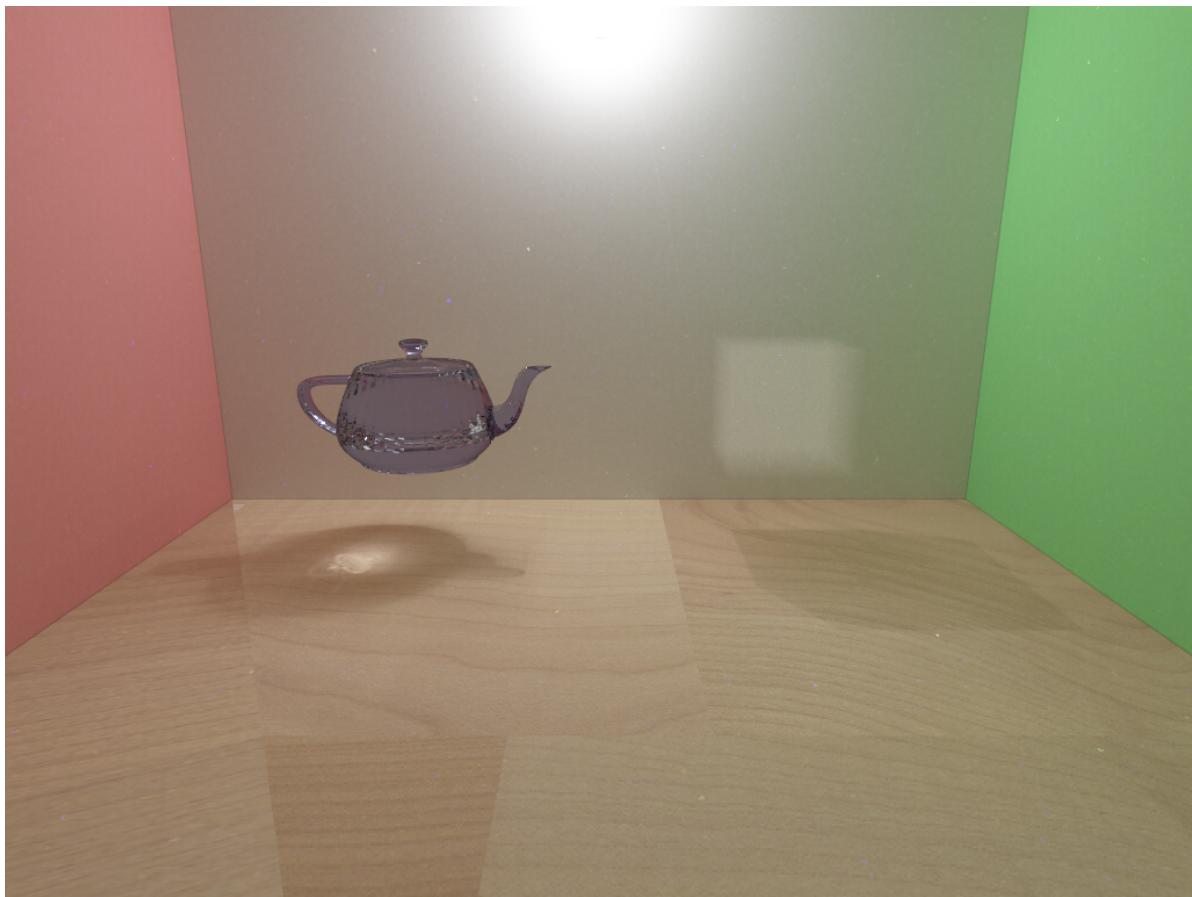

```

最终效果

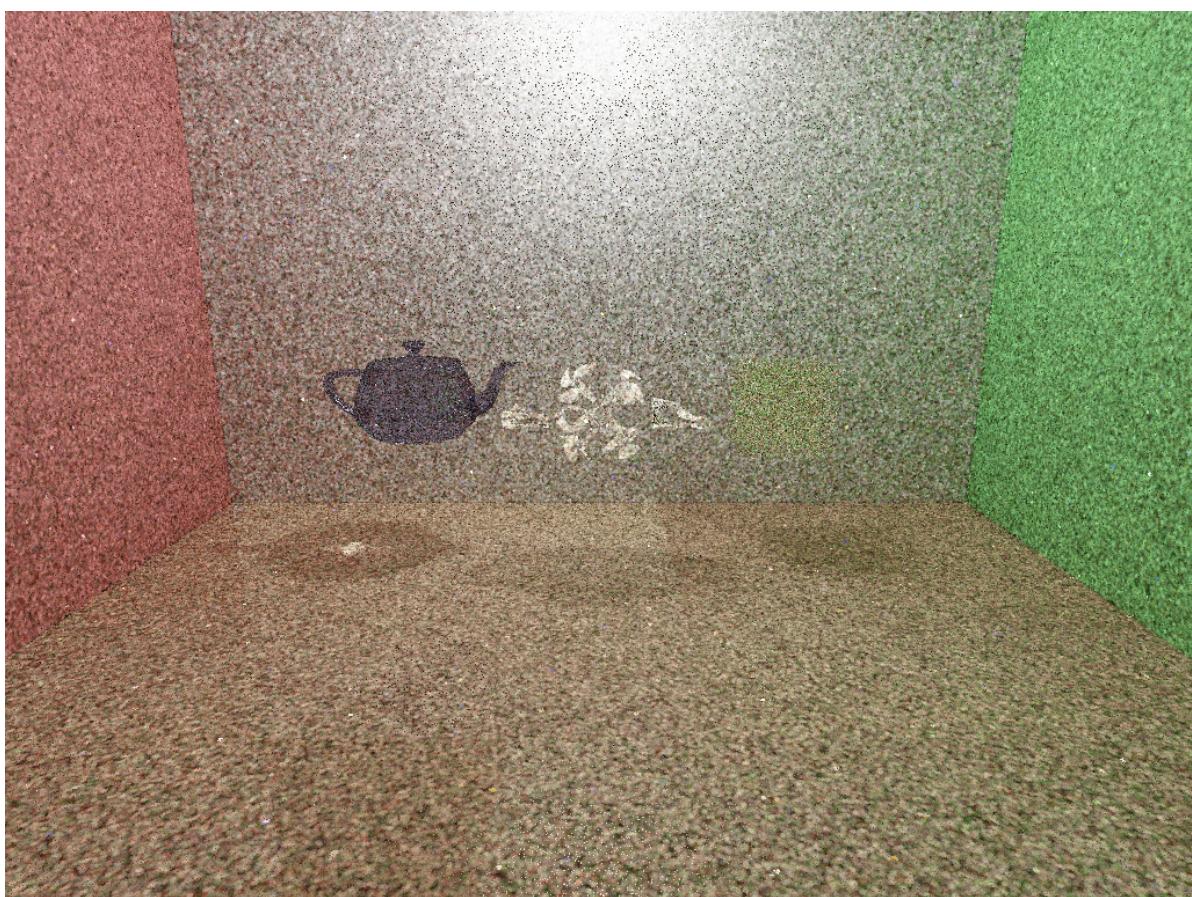
- 模拟相机效果，纹理映射，基本材质，面光源



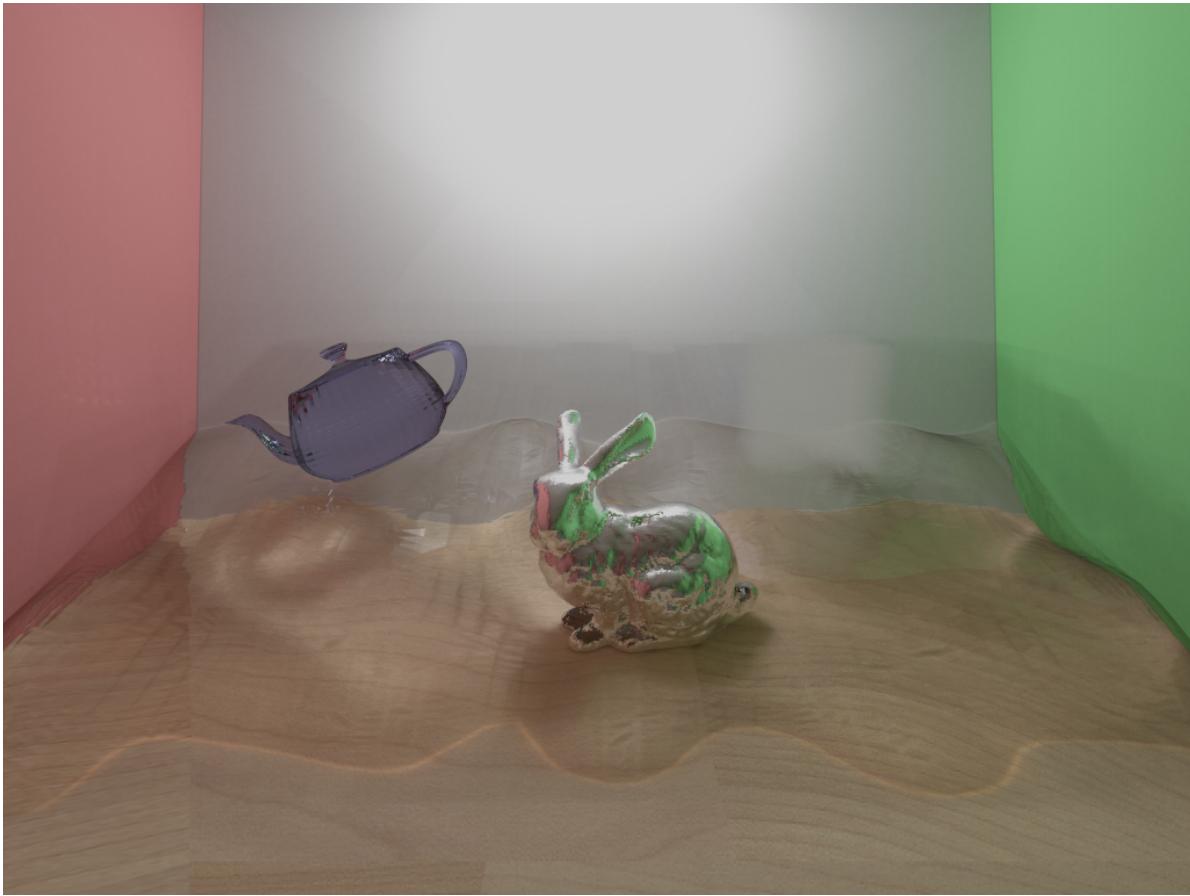
- 强烈的焦散效果



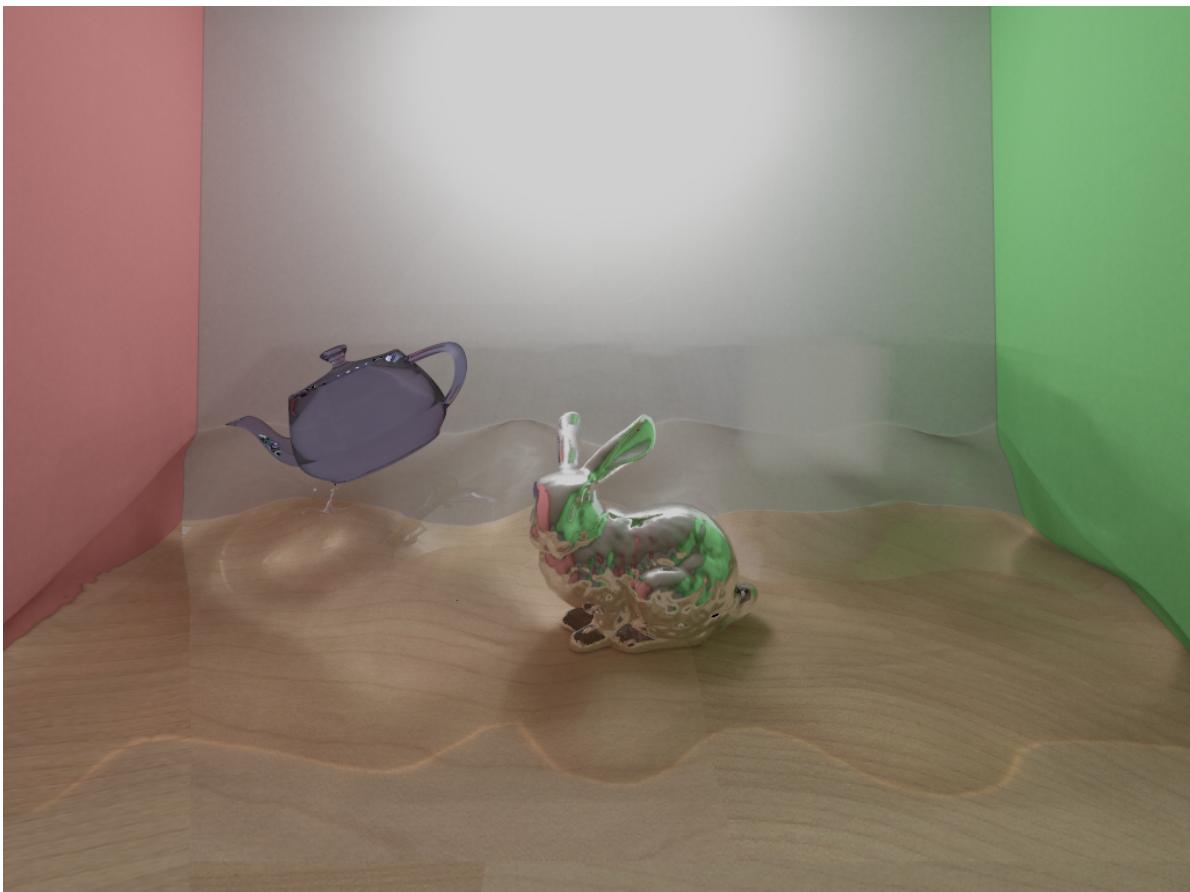
- 贝塞尔曲面求交测试



- 复杂网格，体积雾化效果和纹理贴图，空间变换



- 法向插值（请和上图进行比较）



文件树说明

`include` 文件夹存在了大多数的源文件

`include/Eigen` 是Eigen开源线性代数库

`src` 是main函数的地方

`scene` 是场景配置文件

`tex` 是纹理

`poly` 是各种网格数据

`pics` 是效果图片

`progressive` 里面有一张收敛过程的`gif`可以看看

`.clang-format` 是Clang-Format格式化配置

`compile_flags.txt`是clangd的配置

`Makefile` 是 `Makefile`

请注意：编译的时候我自己使用的是`clang++`，如果没有的话可以到`Makefile`里面修改成`g++`

别的

没了