

高等计算机图形学课程作业：光线追踪渲染器

主算法

采用路径追踪算法。对于每个像素，从相机类获取多条射向场景的光线。对于每条光线，重复以下过程多次：求得在场景中的最近交点，根据交点所在表面的材质类型计算下一条光线的方向。渲染方程的蒙特卡洛估计形式：

$$L_o(x; w_o) = \frac{1}{N} \sum_{w_i} \frac{L_i(x; w_i) f(w; w_i \rightarrow w_o) \langle w_i, n(x) \rangle}{\text{pdf}(w_i)}$$

由于每次求交后只生成一条入射光线，在过程中只需要记录此前所有 $\frac{f(\cdot)\langle \cdot, \cdot \rangle}{\text{pdf}(\cdot)}$ 的乘积；为了降低方差，在入射光线有多种可能的情况下，正比于其BRDF随机生成。

加速

坐标轴对齐包围盒

判断光线和物体是否相交前，先判断光线是否与物体的坐标轴对齐包围盒

$B = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$ 相交。在解析场景时预处理好每个物体的包围盒，求交时转化为射线的每维分量与对应的一对平面求交，再将参数区间求并集。

代码实现见 code/include/object/object3d.hpp，19至59行。

```
struct BoundPlane {
    double coorMin, coorMax;

    inline query queryIntersectX(Ray r) {
        Vector3d p = r.getOrigin(), v = r.getDirection();
        if (v.x() == 0) {
            if (p.x() < coorMin || p.x() > coorMax) return std::make_pair(0, -1);
            return std::make_pair(-1e38, 1e38);
        }
        double t0 = (coorMin - p.x()) / v.x(), t1 = (coorMax - p.x()) / v.x();
        if (t0 > t1) return std::make_pair(t1, t0);
        return std::make_pair(t0, t1);
    }

    inline query queryIntersectY(Ray r) {...}

    inline query queryIntersectZ(Ray r) {...}

    inline void merge(const BoundPlane &b) {
        coorMin = std::min(coorMin, b.coorMin);
        coorMax = std::max(coorMax, b.coorMax);
    }
};
```

三维物体二叉树

将场景中的所有物体放在一个树形结构中。对于树的每个节点，选择 x, y, z 中的一维坐标，依照包围盒对应维度的最小值排序，将前一半和后一半分别划入两个子树中，最终每个三维物体被存放在这个二叉树的叶子节点。二叉树每个节点也需要维护包围盒，表示这个节点及其子树中所有三维物体的包围盒，递归求交时若与此包围盒无交，则不必递归进入子树。

代码实现见 code/include/object/object3d.hpp , 129至188行。

```
struct TreeNode {
    TreeNode *lc, *rc;
    BoundPlane planeX, planeY, planeZ;
    Object3D* obj;
} *rt = NULL;

void buildTree(TreeNode *&rt, std::vector <Object3D*> v, int dep) {
    rt = new TreeNode;
    rt->lc = rt->rc = NULL;
    if (v.size() == 1) {
        rt->planeX = v[0]->getBoundPlaneX();
        ...
        rt->obj = v[0];
        return;
    }

    std::vector <std::pair <double, int> > coorBuf;
    if (dep == 0) {
        for (int i = 0; i < v.size(); ++i)
            coorBuf.push_back(std::make_pair(v[i]->getBoundPlaneX().coorMin, i));
    } else if (dep == 1) {...}
    else {...}
    std::sort(coorBuf.begin(), coorBuf.end());

    std::vector <Object3D*> tempV;
    for (int i = 0; i < v.size() / 2; ++i)
        tempV.push_back(v[coorBuf[i].second]);
    buildTree(rt->lc, tempV, (dep + 1) % 3);

    tempV.clear();
    for (int i = v.size() / 2; i < v.size(); ++i)
        tempV.push_back(v[coorBuf[i].second]);
    buildTree(rt->rc, tempV, (dep + 1) % 3);

    rt->planeX = rt->lc->planeX; rt->planeX.merge(rt->rc->planeX);
    ...
    rt-> obj = NULL;
}

bool queryIntersect(TreeNode *rt, const Ray &r, Hit &h, const double &tmin, const bool &testLs = false)
{
    if (rt == NULL) return false;
    if (rt->obj != NULL) return rt->obj->intersect(r, h, tmin, testLs);
    query qX = rt->planeX.queryIntersectX(r),
    ...;
    double L = std::max(qX.first, std::max(qY.first, qZ.first)),
           U = std::min(qX.second, std::min(qY.second, qZ.second));
    if (L > U || U < 0 || L >= h.getT()) return false;
    bool ret = queryIntersect(rt->lc, r, h, tmin, testLs);
    if (testLs && ret) return true;
    ret |= queryIntersect(rt->rc, r, h, tmin, testLs);
    return ret;
}
```

光源采样

在漫反射、塑料、粗糙塑料等表面，BRDF不是狄拉克函数。若依每个面积元的BRDF成正比采样入射光线，在光源占该点立体角较小的情况下，需要更多的采样次数才能获得噪点较少的画面。因此，在这些材质表面，增加在光源上采样的步骤：对于每个光源，分别在表面采样一点，计算该点对于材质表面的辐照度。对此主算法需要进行修改，当与光源相交时，如果上一次的光线在材质表面进行了光源采样，则不能重复累加该光源的光强。

假设光源面积为 A ，光源上任一点为 x' ，则积分中 $d\omega_i = \frac{\langle x', n(x') \rangle}{|x - x'|^2} dA$ ，蒙特卡洛估计项作对应修改。

代码实现见 code/src/main.cpp , 28至54行。

```
inline Vector3d lightSampling(Material* m, const Vector3d &x,
    Hit& hit, const Vector3d &wo, Sampler* sampler) {

    Vector3d illum = Vector3d::ZERO;
    for (auto l: lights) {
        Vector3d y, ny;
        double A;
        if (!l->getSample(x, y, ny, A, sampler))
            continue;
        Vector3d z = (y - x).normalized();
        double cosTh = Vector3d::dot(hit.getShadeNormal(), z);
        if (cosTh <= 0)
            continue;
        Vector3d color = m->getColor(wo, z, hit);
        if (color.length() < 1e-9) continue;
        double cosThP = Vector3d::dot(ny, -z);
        double dist = (x - y).length();
        Ray testRay(x, z);
        Hit testHit = Hit(dist + 1e-9, NULL, NULL, Vector3d::ZERO, Vector2d::ZERO,
                           true);
        if (baseGroup->intersect(testRay, testHit, 1e-9, true))
            continue;
        illum += l->getEmission() * color * cosTh * cosThP
            / (y - x).squaredLength() * A;
    }
    return illum;
}
```

附加功能

场景解析器

该渲染器使用 .xml 格式的场景描述文件，其语法参考Mitsuba渲染器，兼容简单的Mitsuba场景。支持功能较杂，在此不作具体叙述，可以参考 code\testcases 下的各场景文件制作新的场景。

代码实现见 code/include/parser.hpp 。

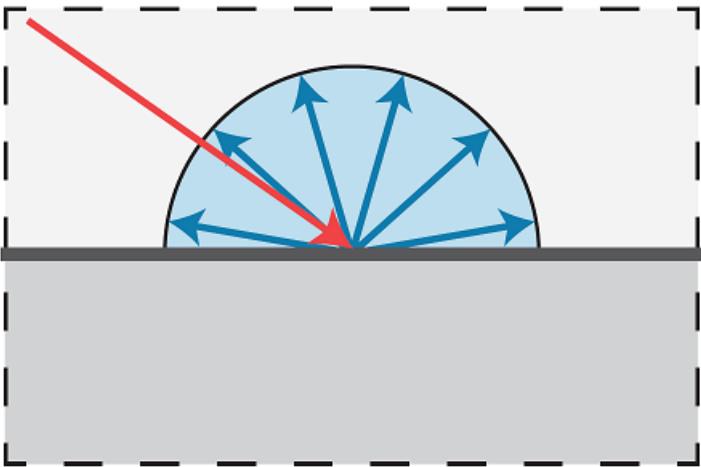
多种材质

除了基本的漫反射 (Diffuse) 、镜面反射 (Conductor::none) 、电介质 (Dielectric) 外，该渲染器还支持导体 (Conductor) 、粗糙导体 (RoughConductor) 、薄电介质 (ThinDielectric) 、塑料 (Plastic) 和粗糙塑料 (RoughPlastic) 。

Material 类提供了通用的 sampleBSDF 接口，用于材质表面采样；对于非狄拉克表面，还提供了 getColor 接口，用于在光源采样时计算光源对材质表面的颜色。

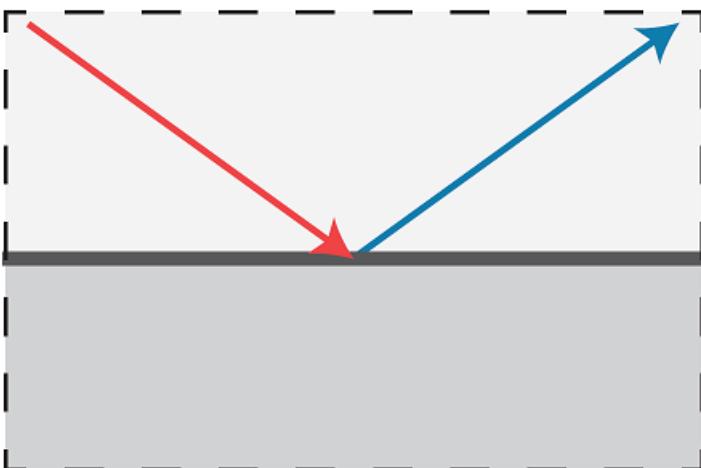
代码实现见 code/include/bsdf 下的对应文件。

漫反射



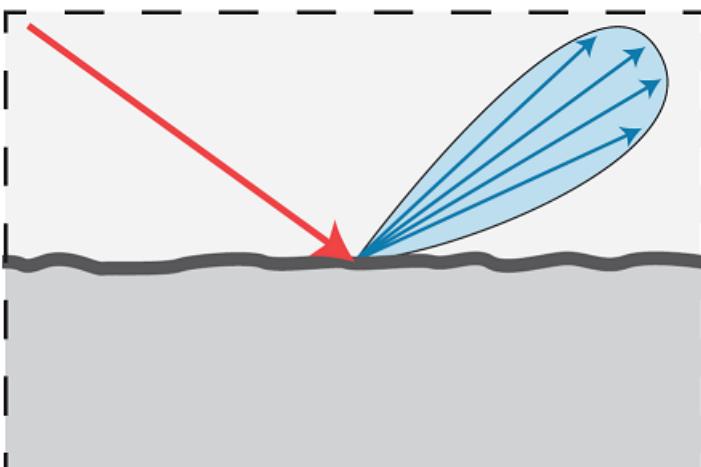
入射光线在法线所在的半球面采样。实现中使用了重要性采样，即在半球上按照极角的余弦值为分布函数采样，这样与渲染方程中的内积项抵消，可以减少辐照度的方差。

导体



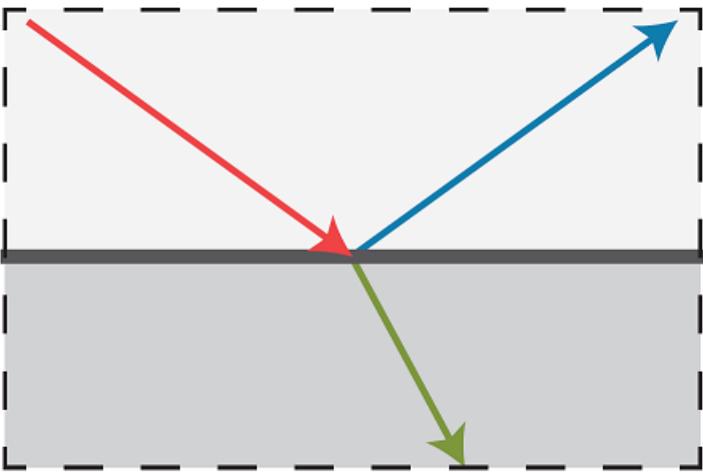
入射光线为出射光线关于法线的共面镜像，其反射率（即光强）由夹角、折射率 (η) 和吸收系数 (k) 根据导体的菲涅尔公式计算得出。镜面反射材质可以通过设 $\eta = 0, k = 1$ 来模拟。

粗糙导体



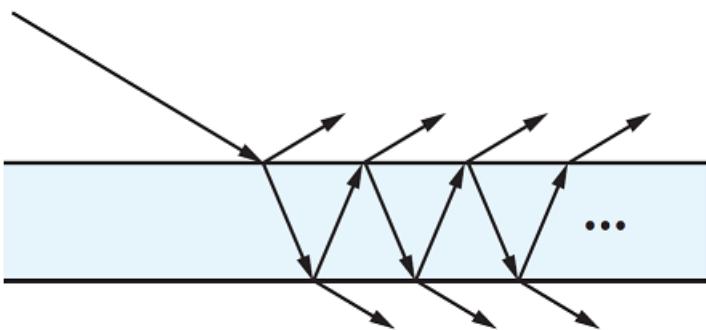
粗糙导体的表面由大量微小的起伏微表面模拟，这些微表面与导体同材质。入射光线的采样本质上是微表面采样，这里使用了GGX分布重要性采样 (<https://schuttejoe.github.io/post/ggximportancesamplingpart2/>)，其概率密度函数则从知乎 (<https://zhuanlan.zhihu.com/p/20119162>) 获得。

电介质



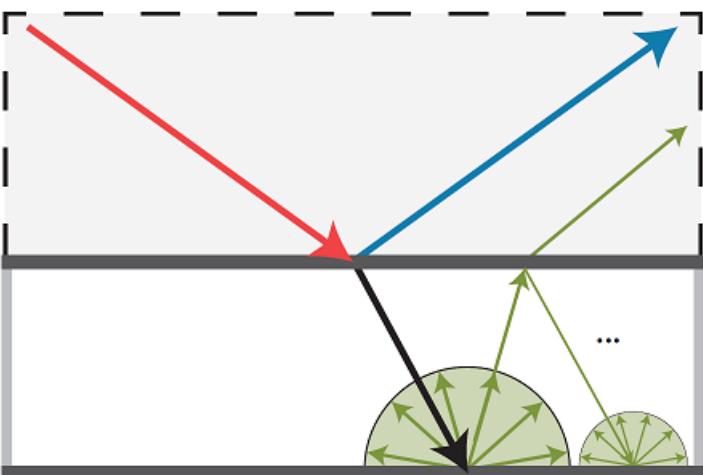
电介质即折射材质，反射率 r 和折射方向由夹角、材质两侧的折射率（ η_i, η_t ）由电介质的菲涅尔公式计算得出。采样时以 r 的概率反射、 $1 - r$ 的概率折射，若入射光为折射，根据能量和辐照度的关系，需要将辐照度乘以 $\left(\frac{\eta_t}{\eta_i}\right)^2$ 。

薄电介质



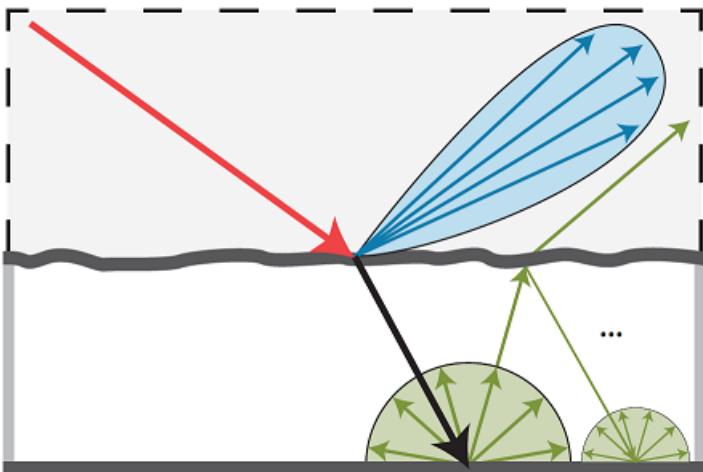
薄电介质模型假设电介质为非常薄的一层，两侧都视为其外部。在该假设下，可以用很简单的数学模型刻画在电介质层间的多次反射。由于电介质层很薄，所有的入射光都聚在出射点。若同材质的电介质有反射率 r ，则从出射光同侧射入的概率为 $\frac{2r}{1-r}$ ，其余均为从材质另一侧射入。辐照度不需再乘以系数。

塑料



塑料模型假设在漫反射面外层存在非常薄的电介质涂层。同样，所有的入射光都聚在出射点。该模型的实现基于论文 (<https://hal.inria.fr/hal-01386157/document>)。对于漫反射部分，需要知道在半球面上的入射光最后能折射的比例，这里采用大量随机取平均的做法简单求解近似值，但理论上应该存在很好的积分解。

粗糙塑料



粗糙塑料模型即将塑料的电介质表层改为微表面。其漫反射项与塑料一致，只需在 `getColor` 函数中添加镜面反射项。镜面反射部分和粗糙导体实现方法相同。在BSDF采样中，等概率选择漫反射和镜面反射，概率密度函数取平均。

景深

将简单的小孔成像相机改为透镜成像相机，接受场景文件给定的光圈半径和成像距离（透镜到焦平面的距离）。对于原本图片上的一个像素点 p ，求射线 $o + t(o - p)$ 与焦平面的交点 x 。在光圈上均匀随机采样得到点 o' ，最终使用射线 $o' + t(x - o')$ 渲染。

代码实现见 `code/include/camera.hpp`，63至66行。

```
inline Ray generateRay(const Vector2d &point, Sampler* sampler) {
    Vector3d dir;
    if (fovAxis == "y")
        dir = ...
    else if (fovAxis == "x")
        dir = ...
    double t = distance / Vector3d::dot(dir, direction);
    Vector3d x = center + t * dir;
    Vector2d dc = sampler->sampleDiskUniform();
    Vector3d nc = center + aperture * dc.x() * up + aperture * dc.y() * horizontal;
    return Ray(nc, (x - nc).normalized());
}
```

软阴影

路径追踪算法是真实感渲染算法，在面积光源下自然有软阴影。

抗锯齿

在主算法中，每个像素点的颜色是其 2×2 子像素颜色的平均，实现了简单的抗锯齿。场景文件指定的 `sampleCount` 为该像素整体的采样次数，故每个子像素使用 $\frac{1}{4}sampleCount$ 次采样。

代码实现见 `code/src/main.cpp`，124至135行。

```

for (int sy = -1; sy < 2; sy += 2)
    for (int sx = -1; sx < 2; sx += 2) {
        Vector3d r(0);
        for (int s = 0; s < samps; ++s) {
            double r1 = 2 * erand48(Xi), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
            double r2 = 2 * erand48(Xi), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);
            Ray camRay = camera->generateRay(Vector2d(x + 0.25 * sx + 0.5 * dx + 0.5, y + 0.25 * sy +
0.5 * dy + 0.5), sampler);
            r = r + rayTracing(camRay, sampler);
        }
        r = r / samps;
        finalColor = finalColor + Vector3d(clamp(r.x()), clamp(r.y()), clamp(r.z())) / 4;
    }
}

```

顶点法向、贴图和凹凸贴图

由 .obj 文件给定三角形各顶点的顶点法向 vn_i 和纹理坐标 vt_i 。对于光线与三角形 ABC 相交的点 p , 求出其重心坐标 (α, β) , 通过加权平均得到着色法向 $sn = \alpha vn_A + \beta vn_B + (1 - \alpha - \beta)vn_C$ 和纹理坐标 $(x, y) = \alpha vt_A + \beta vt_B + (1 - \alpha - \beta)vt_C$ 。

贴图由场景文件给出, 在 $w \times h$ 的贴图中读取像素 $(\lfloor xw \rfloor, \lfloor yh \rfloor)$ 的值作为该交点的颜色。如果还给定了凹凸贴图, 则根据该像素点关于 x 和 y 的两个中心差分计算法向的扰动量。在此之前需要对每个三角形预处理两个在三角形平面中的向量 p_u 和 p_v 用作扰动, 根据课本上的公式 $n' = n + n \times (dup_u - dvp_v)$ 计算得到新的法向。

顶点法向代码实现见 code/include/object/triangle.hpp , 第102、107行。

```

h.setShadeNormal(((alpha * va + beta * vb + (size - alpha - beta) * vc) / size).normalized());

```

贴图代码实现见 code/include/texture/texture.cpp , 24至38行。

```

inline void Texture::albedo(Hit &hit) {
    if (use_uniform) {
        hit.setColor(uniform);
        return;
    }

    int w = bitmap->w, h = bitmap->h;
    Vector2d texCoor = hit.getTexCoor();
    int x = (int(texCoor.x() * w) % w + w) % w,
        y = (int(texCoor.y() * h) % h + h) % h;
    int r = bitmap->data[(y * bitmap->w + x) * bitmap->bpp],
        g = bitmap->data[(y * bitmap->w + x) * bitmap->bpp + 1],
        b = bitmap->data[(y * bitmap->w + x) * bitmap->bpp + 2];
    hit.setColor(Vector3d(r / 256.0, g / 256.0, b / 256.0));
}

```

凹凸贴图代码实现见 code/include/texture/bump.hpp , 36至47行。

```

double xr = texCoor.x() * w - x, yr = texCoor.y() * h - y;
double dfdx1 = dfdx[x][y],
       dfdx2 = dfdx[x][y + 1],
       dfdy1 = dfdy[x][y],
       dfdy2 = dfdy[x - 1][y];
double bu = xr * dfdx2 + (1 - xr) * dfdx1,
      bv = (1 - yr) * dfdy2 + yr * dfdy1;
Vector3d pu, pv, n, shadeN = hit.getShadeNormal();
hit.getTangent(pu, pv);
n = (shadeN + Vector3d::cross(shadeN, bu * pv - bv * pu)).normalized();
if (Vector3d::dot(n, shadeN) < 0) n = -n;
hit.setShadeNormal(n);

```

样条曲线旋转体解析法求交

设直线参数为 t , 样条曲线旋转体的参数为 s 和 θ , 得到非线性方程组

$$F(t, s, \theta) = \begin{bmatrix} p_x + tv_x - f(s)_x \cos \theta \\ p_y + tv_y - f(s)_y \\ p_z + tv_z - f(s)_x \sin \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

计算其雅可比矩阵得到

$$J_F = \begin{bmatrix} v_x & -\cos \theta \cdot [f(s)_x]' & \sin \theta \cdot f(s)_x \\ v_y & -[f(s)_y]' & 0 \\ v_z & -\sin \theta \cdot [f(s)_x]' & -\cos \theta \cdot f(s)_x \end{bmatrix}.$$

于是由牛顿迭代法可以得到 $(t_{k+1}, s_{k+1}, \theta_{k+1}) = (t_k, s_k, \theta_k) - J_F^{-1}F$ 。在实现中我采用了类似机器学习中学习速率的概念。考虑到样条函数在定义区间以外 $(-\infty, 0) \cup (1, +\infty)$ 会剧烈波动, 我还在求值时特判了这种情况, 并用直线连接了0和1处, 斜率分别为0和1处的导数, 这样也可以避免 s 远离定义区间。

代码实现见 `code/include/object/revsurface.hpp`, 71至99行。

```

double eta = 0.1;
for (int i = 0; i < NewtonSteps; ++i) {
    CurvePoint fs = pCurve->evaluate(s);
    Vector3d F(p.x() - fs.V.x() * cos(theta),
                p.y() - fs.V.y(),
                p.z() - fs.V.x() * sin(theta));
    if (F.length() < eps) {
        if (t >= tmin && t < h.getT()) {
            if (testLs) return true;
            Vector3d n(-fs.T.y() * cos(theta), fs.T.x(), -fs.T.y() * sin(theta));
            n.normalize();
            if (Vector3d::dot(r.getDirection(), n) < 0)
                h.set(t, this, material, n, Vector2d::ZERO, true);
            else
                h.set(t, this, material, -n, Vector2d::ZERO, false);
            return true;
        }
        return false;
    }

    Matrix3d JF(r.getDirection().x(), -cos(theta) * fs.T.x(), sin(theta) * fs.V.x(),
                 r.getDirection().y(), -fs.T.y(), 0,
                 r.getDirection().z(), -sin(theta) * fs.T.x(), -cos(theta) * fs.V.x());
    Vector3d delta = JF.inverse() * F;
    t -= eta * delta.x(); s -= eta * delta.y(); theta -= eta * delta.z();
    eta *= 0.99;
    if (isnan(t) || isnan(s) || isnan(theta))
        return false;
}

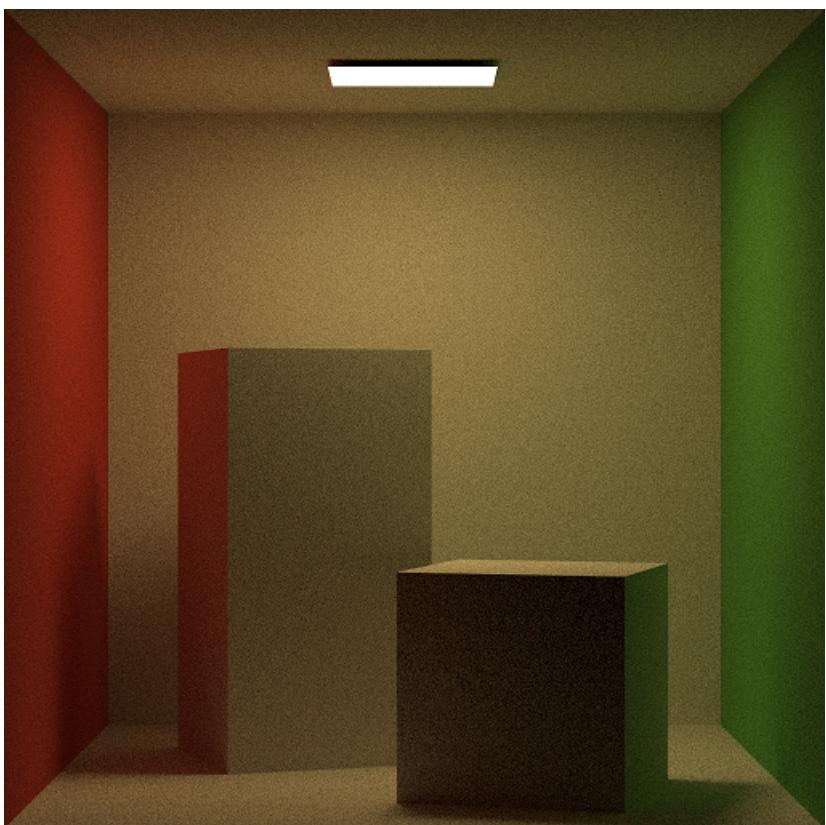
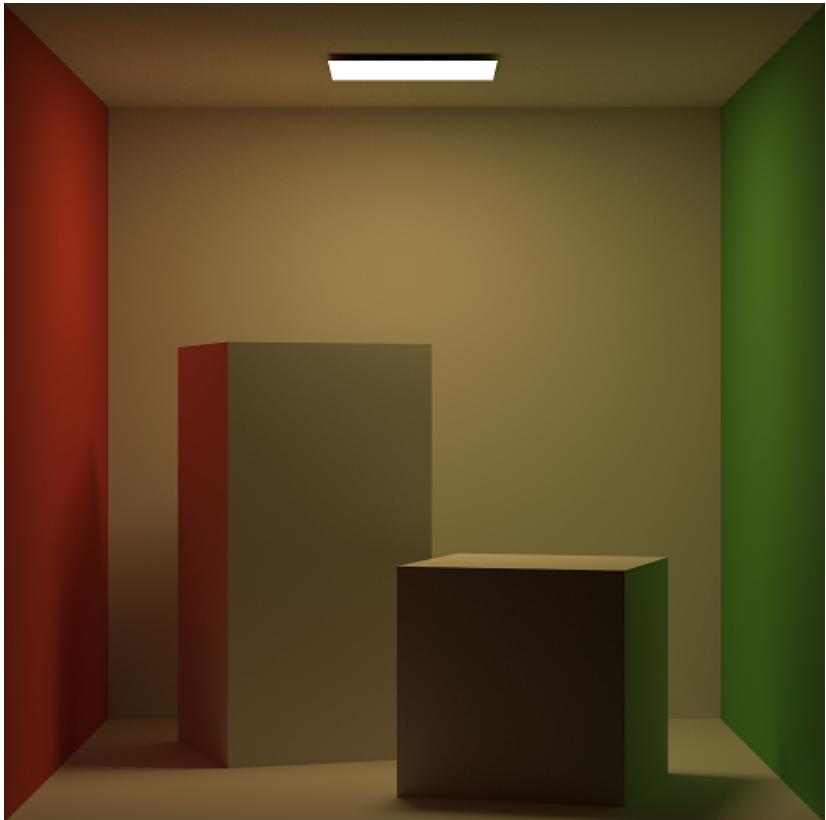
```

成果

高清图在 gallery 下。

软阴影、光源采样和渗色

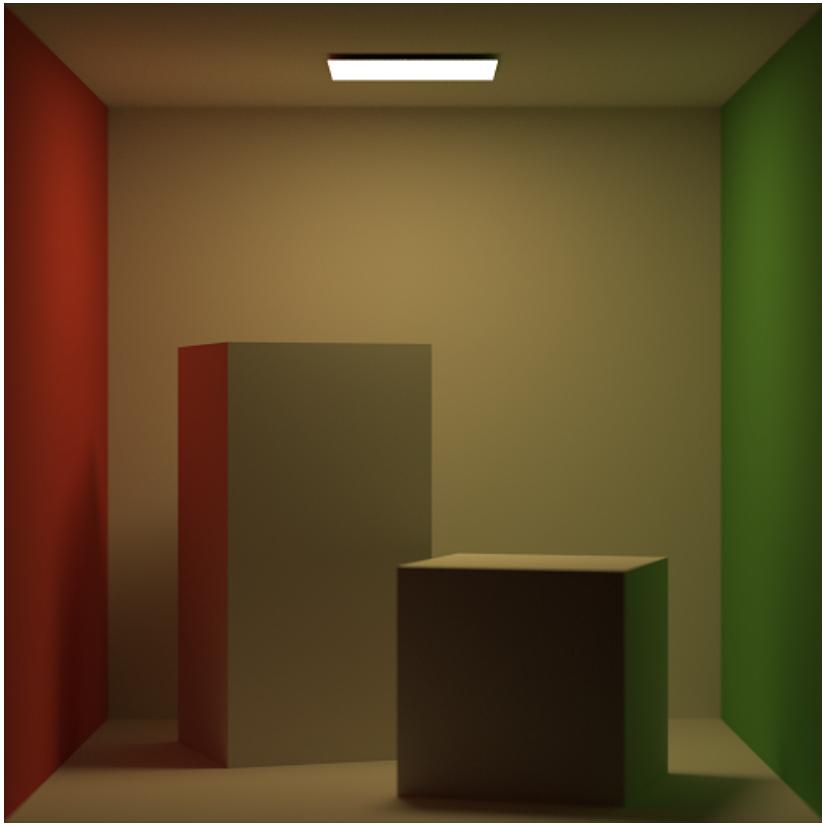
下面两张图同样采用1000spp渲染，时间相差不到10，而前者采用了光源采样，相对后者噪点显著减少。



两个盒子的材质均为浅色漫反射，可以反映出墙壁的颜色。

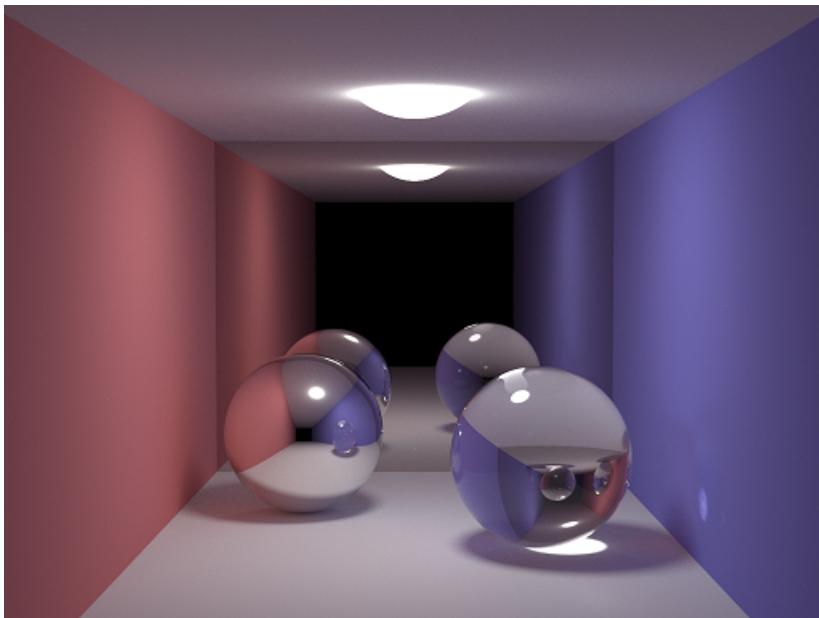
景深

下图添加了景深效果，可以看出焦平面大致在高盒子的前表面上，因此矮盒子和背景墙的墙角均变模糊。



导体和电介质

下图中，前面的球为完美电介质，后面的球为银材质，背景墙为钼材质。导体的折射率和吸收系数参考了Tungsten (<https://github.com/tunabrain/tungsten/blob/master/src/core/bsdfs/ComplexIorData.hpp>)。



样条曲线旋转体和薄电介质

下图中的两个杯子均用样条曲线旋转体制绘，左边为完美电介质，右边为薄电介质，可以看出薄电介质能模拟出中空的效果，且表面有多层反射。



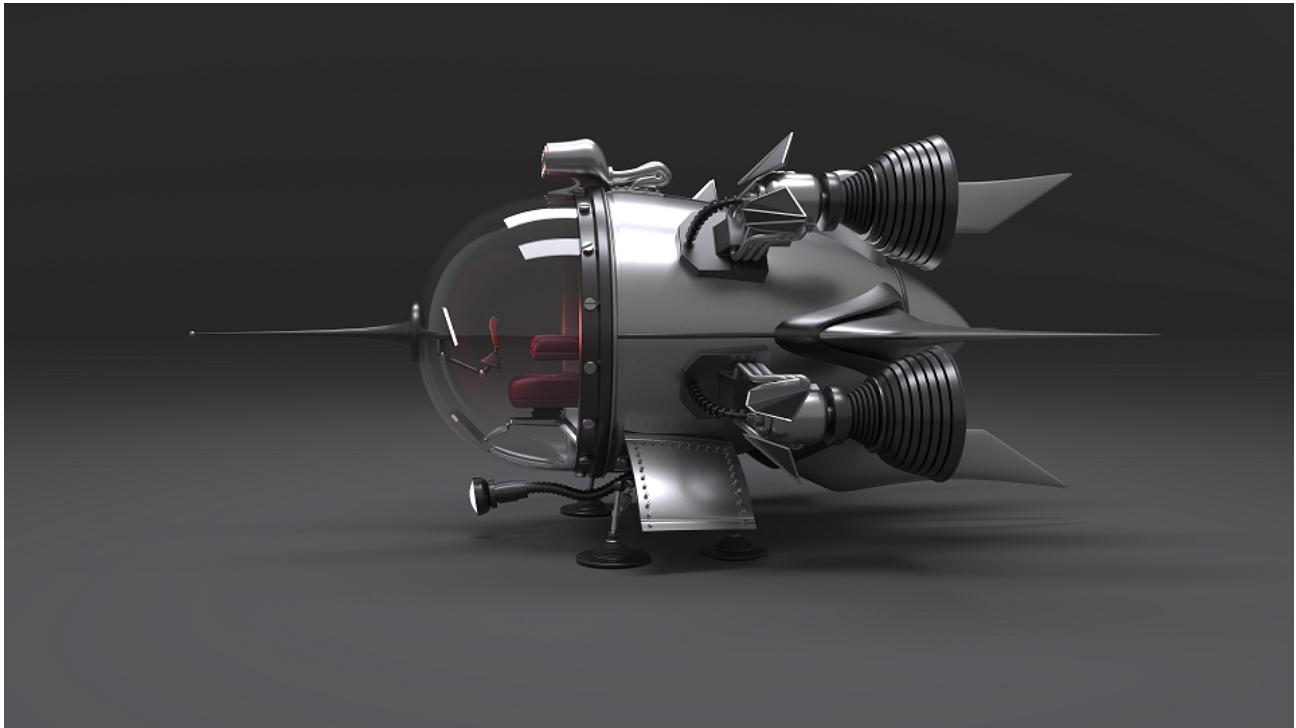
顶点法向、塑料和粗糙塑料

咖啡机的全部组件均在 .obj 文件中给出，顶点法向使得咖啡机看起来比较光滑。橙色杯体为塑料材质，黑色杯体和把手为粗糙塑料材质，反光效果各不相同。



粗糙导体

飞船的船身为粗糙导体，发动机的倒影变得模糊。



加速、几乎所有的材质、贴图和凹凸贴图

该场景有约150万个三角形，各种加速算法使得收敛效果较好。厨房中几乎涉及了所有的材质，毛巾、书籍、胡萝卜和收音机上均有贴图，面包箱表面为凹凸贴图。



后续

可以关注我的GitHub仓库 (<https://github.com/zhourunlong/ComputerGraphics>)，后续可能会有细节上的修正和优化。