光线追踪算法分析

1.求交

光线追踪主要的计算量来源于大量的求交计算。设 O 代表射线起点，D 方向 ，P 为圆上的点，C 为圆心，r 半径。球的方程为:(P - C)(P - C) = r * r ，直线的参数方程: p(t) = O + tD。

将直线方程代入后得 D^2t^2+2(O-C)Dt+(O-C)^2-r^2=0，随后利用一元二次方程求根公式，判断有无解，有两个解时，选择>0 且较小的 t。

求交的基本原理就是将射线的参数方程代入到圆的函数中，求 t 的值。

1）将 P(t) = O + tD 代入圆方程，会得到 t 的一元二次方程。

2）先求出 Vec op，op 是用球心 p 的坐标减去射线的起点 (O - C)。

3）b = op.dot(r.d)指代 ” D * (O - C) ”

4）求 det，这里要注意我们求的 b 和原理中的 b 差了两倍，所以可以直接用

```
double det = b * b - op.dot(op) + rad * rad;
```

如果 det<0 说明无解，直接 return 0。

否则求根号的 det

5）最终的解有一个或两个，可能在 t = b - det，或者 t = b + det 中，选择 t 大于 0 并且两个中较小的 t。

2.绘制

1）用 6 个很大的球体当做平面（DIFF 属性，只有漫反射），因为半径很大的话，你在近距离看起来，球面就很像一个平面。

作者这样做应该是为了避免去写平面求交，平面类等函数。

2）用 1 个球表示光源，就是 Lite，1 个 Mirr 球（完全反射），1 个 Glass 球（折射和反射都有）

遍历所有的球，求交点

```
inline bool intersect(const Ray &r, double &t, int &id) {
    double n = sizeof(spheres) / sizeof(Sphere), d, inf = t = 1e20;
    for (int i = int(n); i--;) {
        if ((d = spheres[i].intersect(r)) && d < t)
        {
            t = d;
            id = i;
        }
    }
    return t < inf;
}
```

此光线射出去，在所有的球体中求交点。

求出距离 camera 最近的交点，这就是待会要绘制在屏幕上的主要的点。

3.主函数说明

1）camera 的位置是在（50， 52， 295.6）， 往 z 轴的负方向看。

```
int main(int argc, char *argv[])
{
```

```
int w = 1024, h = 768, samps = argc == 2 ? atoi(argv[1]) / 4 : 10; // # samples
Ray cam(Vec(50, 52, 295.6), Vec(0, -0.042612, -1).norm()); // cam pos, dir
Vec cx = Vec(w*.5135 / h), cy = (cx.cross(cam.d)).norm()*.5135, r, *c = new Vec[w*h];
```

2）遍历每个像素点，用随机采样的方式求得要射出的光线的方向 d。

```
for (int y = 0; y<h; y++) {                                    // Loop over image rows
        fprintf(stderr, "\rRendering (%d spp) %5.2f%%", samps * 4, 100.*y / (h - 1));
        for (unsigned short x = 0, Xi[3] = { 0,0,y*y*y }; x<w; x++)      // Loop cols
            for (int sy = 0, i = (h - y - 1)*w + x; sy<2; sy++)          // 2x2 subpixel rows
                for (int sx = 0; sx<2; sx++, r = Vec()) {                // 2x2 subpixel cols
                    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);
                        Vec d = cx*(((sx + .5 + dx) / 2 + x) / w - .5) +
                            cy*(((sy + .5 + dy) / 2 + y) / h - .5) + cam.d;
                        r = r + radiance(Ray(cam.o + d * 140, d.norm()), 0, Xi)*(1. / samps);
                    } // Camera rays are pushed ^^^^^ forward to start in interior
                    c[i] = c[i] + Vec(clamp(r.x), clamp(r.y), clamp(r.z))*.25;
                }
    }
    FILE *f = fopen("image.ppm", "w");                 // Write image to PPM file.
    fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
    for (int i = 0; i<w*h; i++)
        fprintf(f, "%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
```

4.光线追踪递归说明

_Vector radiance：实现了光线跟踪处理流程，该函数中进行了递归调用。光线跟踪递归过程终止条件是光线与环境中任何物体均不相交，或交于纯漫射面、被跟踪光线返回的光亮度值对像素颜色的贡献很小、已递归到给定深度。该函数传入两个参数，一个是射线的引用，一个是递归的深度。

首先求出射线相交物体的距离以及与射线相交物体的 id，如果没有相交，则返回一个 emission(0,0,0)的向量。如果相交，求出物体被击中的那个点，并计算法向量 normal，normal_real 并进行向量单位化。然后判断递归是否达到给定深度，深度大于 100 就结束。深度大于 5 时，从 0-1 随机一个浮点数与 RGB 颜色分量中的最大值 P 进行比较，如果随机的数小于 P，就返回当前的颜色值。否则就根据球体的材质类型，进行反射折射等计算。其中漫反射取随机数以及 w、u、v 三个正交向量求出一个随机的漫反射光线，并继续迭代。镜面反射则直接求出反射光的角度。反射加折射首先判断 normal 和 normal_real 是否为同一方向，然后计算折射率和入射角余弦，进行菲涅尔折射反射等计算，最后返回颜色值，使用了轮盘赌的算法进行递归调用

设定好递归出口（depth 的值），对每个球体与光线求交，并使得法向量与 ray._direct 呈钝角（法向量指向球体外）。

1） 判断是否相交，求交点，求表面法向

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi) {
    double t;                                          // distance to intersection
    int id = 0;                                        // id of intersected object
```

```
if (!intersect(r, t, id))
    return Vec(); // if miss, return black
const Sphere &obj = spheres[id];            // the hit object
Vec x = r.o + r.d*t, n = (x - obj.position).norm(); // calculate vector n,球面法向量
Vec nl = n.dot(r.d) < 0 ? n : n*-1, f = obj.color;
double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
if (++depth>5||!p)
    if (erand48(Xi)<p)
        f = f*(1 / p);
    else
        return obj.emission;
```

2）漫反射（DIFF）

如果材质是漫反射，那么就随机生成一个方向进行漫反射。

利用法线向量 w 与向量(0,1,0)或(1,0,0)进行叉乘运算得到向量 u,随后 w 与 u 进行叉乘得到向量 v,利用叉乘运算的方向得到了一组标准正交基 w,u,v。利用随机函数 drand48（）得到两个随机数 r1,r2，通过二者的运算得到 3 个坐标，进而得到在标准正交基 w,u,v 下的一个随机向量 direct,即求得了一个随机的漫反射光线从而继续递归。

```
if (obj.refl == DIFF) {                     // Ideal DIFFUSE reflection
    double r1 = 2 * M_PI*erand48(Xi), r2 = erand48(Xi), r2s = sqrt(r2);
    Vec w = nl, u = ((fabs(w.x)>.1 ? Vec(0, 1) : Vec(1)).cross(w)).norm(), v = w.cross(u);    //w，v，u 为正交基
    Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1 - r2)).norm();
    return obj.emission + f.mult(radiance(Ray(x, d), depth, Xi));
}
```

3）镜面反射（材质为 SPEC）

计算镜面反射的方向，然后继续递归

由于漫反射和镜面反射都遵循反射规律，因此根据反射定律计算出反射光的方向，进而继续递归。

```
else if (obj.refl == SPEC)                  // Ideal SPECULAR reflection
    return obj.emission + f.mult(radiance(Ray(x, r.d - n * 2 * n.dot(r.d)), depth, Xi));
```

4）反射和折射（材质为 REFR）

玻璃材质，有一部分光进行反射，有一部分光进行折射。

这里用到了轮盘赌方法。

首先计算出相对折射率，由公式 n1sinn1 = n2 sinn2 可以计算出折射角的正弦值，同时根据入射光线的方向，法线方向以及折射的角度可以计算出折射方向从而生成折射光线；根据菲涅尔近似等式，可计算出菲涅尔反射和折射所占的比例（Fr+Fe = 1），从而继续递归。

```
Ray reflRay(x, r.d - n * 2 * n.dot(r.d));         // Ideal dielectric REFRACTION  由平行四边形的方法求得反射光的 direction
bool into = n.dot(nl)>0;                          // Ray from outside going in?
double nc = 1, nt = 1.5, nnt = into ? nc / nt : nt / nc, ddn = r.d.dot(nl), cos2t;
if ((cos2t = 1 - nnt*nnt*(1 - ddn*ddn))<0)        // Total internal reflection
    return obj.emission + f.mult(radiance(reflRay, depth, Xi));
Vec tdir = (r.d*nnt - n*((into ? 1 : -1)*(ddn*nnt + sqrt(cos2t)))).norm();
double a = nt - nc, b = nt + nc, R0 = a*a / (b*b), c = 1 - (into ? -ddn : tdir.dot(n));
```

```
double Re = R0 + (1 - R0)*c*c*c*c*c, Tr = 1 - Re, P = .25 + .5*Re, RP = Re / P, TP = Tr / (1 - P);
return obj.emission + f.mult(depth>2 ? (erand48(Xi)<P ?      // Russian roulette
    radiance(reflRay, depth, Xi)*RP : radiance(Ray(x, tdir), depth, Xi)*TP) :
    radiance(reflRay, depth, Xi)*Re + radiance(Ray(x, tdir), depth, Xi)*Tr);
```

5.场景说明

0.5135 设置的是相机的视角大小，即该值越大视角越大，进而视锥体越胖。Sx,sy 是像素方格的四个顶点，本方法是遍历每个像素点，用随机采样的方式求得要射出的光线的方向 d。(sx + .5 + dx) / 2 这个的值得范围是[-0.25, 0.75]。这个值主要是为了在随机采样时，对 x 进行偏移。我们如果把他忽略不计。那么(((sx + .5 + dx) / 2 + x) / w - .5)的值其实是在[-0.5, 0.5]的。

关于为何要把 d 乘以 140 加到摄像机原点的位置，那是因为摄像机原点落在了“front”这堵墙的外面，如果不加的话，所有的光在发出时都会直接打到这堵墙上，直接返回了墙的颜色。

注：模型已经确定的参数：

空间视点：（x_e，y_e, z_e）

视见距离：D =140*cosθ　　（θ=0.5135）

视线方向：eyedir（x_d，y_d, z_d） =（0，0.042612，-1）；

视线上方：eyeup（x_u，y_u, z_u） =（1，0，0）与 eyedir 叉乘结果

屏中央：　opoint（x_o，y_o, z_o） ＝ eyedir+D*eyedir;

6.坐标系说明：

从相机坐标系到图像坐标系，属于透视投影关系，从 3D 转换到 2D。

计算出比例系数 u 即可计算出投影点的位置坐标。

比例系数为：u = (0.0 - eyePos._z) / (A._z - eyePos._z);

注意：此时投影点 p 的单位还是 mm，并不是 pixel，需要进一步转换到像素坐标系。

像素坐标系和图像坐标系都在成像平面上，只是各自的原点和度量单位不一样。图像坐标系的原点为相机光轴与成像平面的交点，通常情况下是成像平面的中点。图像坐标系的单位是 mm。原点是图像左上角，而像素坐标系的单位是 pixel，我们平常描述一个像素点都是几行几列。所以这二者之间的转换如下：其中 dx 和 dy 表示每一列和每一行分别代表多少 mm，即 1pixel=dx mm

进行归一化

```
Lw = (width) / (140 * sin(0.5135));
Lh = (height) / (140 * sin(0.5135));
```

注意：目前实际是绘制在距 eyePos 点 140 的球面上，而不是在平面上，所以要进行一定的比例调整。

```
Lw = Lw* 0.5135 / sin(0.5135); //弦长比弧长
Lh = Lh* 0.5135 / sin(0.5135);//弦长比弧长
x[i+1] = (int)(((b._x - eyePos._x) * u + eyePos._x) * Lw + 0.5) + width / 2;
y[i+1] = height / 2 - (int)(((b._y - eyePos._y) * u + eyePos._y) * Lh + 0.5);
```

因为像素坐标系左上角为（0,0）点，而图像坐标系图像中心为原点，所以要 width/2、height/2 进行转化。