真实感场景渲染

王焱 2017050024

一、球类

场景中全部物体均为球体，包括包围空间的墙面地面等也由超大半径球体模拟平面构成。球体由原点和半径刻画，有 emission 成员变量用来记录光照，color 为其颜色属性，refl 代表其反射属性，另外用 hastexture 记录该物体是否有纹理贴图，相应的在构造函数中加载出纹理。

```
struct Sphere
{
    double radius;
    Vec3 position, emission, color;
    Refl_t refl;
    bool hastexture = 0;
    int rows, cols;

    Mat texture;
    Sphere(double _r, Vec3 _p, Vec3 _e, Vec3 _c, Refl_t _refl,bool hastexture=0) :radius(_r), position(_p), emission(_e), color(_
    {
        if (hastexture)
        {
            texture = imread("m.jpg", 1);
            rows = texture.rows;
            cols = texture.cols;
        }

    }
```

二、求交

已知一束光线（一条射线，由起始点和方向表示），求它与该物体的交点。基本原理就是将射线的参数方程代入到球的函数中，求 t 的值。将(o+td)带入到球的方程$(p-c)^2=r^2$,据此求得 t。或者从几何关系上也可推出此求解式子。

```
double intersect(const Ray &ray) const
{
    Vec3 op = position - ray.o;
    double t, eps = 1e-4;
    double b = op.dot(ray.d), det = b * b + radius * radius - op.dot(op);
    if (det < 0) return 0;//无交点
    else det = sqrt(det);
    return (t = b - det) > eps ? t : ((t = b + det) > eps ? t : 0);//内部 外部
}
```

下面的函数为发射一条光线，通过和所有的物体求交，最终返回一个距离最近的交点，是其真实交点。

```
inline bool intersect(const Ray &ray, double &t, int &id) //光线和所有物体求交点
{
    double n = sizeof(spheres) / sizeof(Sphere), d, inf = t = 1e20;
    for (int i = int(n); i >= 0; i--)
    {
        if ((d = spheres[i].intersect(ray))!=0 && d < t)
        {
            t = d;
            id = i;
        }
    }
    return t < inf;
}
```

三、渲染

渲染的算法我才用的是 PathTracing。大致分为三步，首先求得交点，其次根据物体的反射属性求得反射方向，最后递归的发出下一个光线。代码中主要涉及到的是漫反射、折射和镜面反射。

```
if (obj.refl == DIFF) //漫反射
{
    double r1 = 2 * M_PI*drand(), r2 = drand(), r2s = sqrt(r2);
    Vec3 w = nl, u = ((fabs(w.x) > .1 ? Vec3(0, 1) : Vec3(1)).cross(w)).norm(), v = w.cross(u);
    Vec3 d = (u*cos(r1)*r2s + v * sin(r1)*r2s + w * sqrt(1 - r2)).norm();
    return obj.emission + f.mult(radiance(Ray(x, d), depth));
}

else
{
   Ray reflRay(x, r.d - n * 2 * n.dot(r.d));
   if (obj.refl == SPEC) // 镜面反射
       return obj.emission + f.mult(radiance(reflRay, depth));
   else//折射加反射
   {
       bool into = n.dot(nl) > 0;
       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)
           return obj.emission + f.mult(radiance(reflRay, depth));
       Vec3 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 ? (drand() < P ?
           radiance(reflRay, depth)*RP : radiance(Ray(x, tdir), depth)*TP) :
           radiance(reflRay, depth)*Re + radiance(Ray(x, tdir), depth)*Tr);
   }
}
```

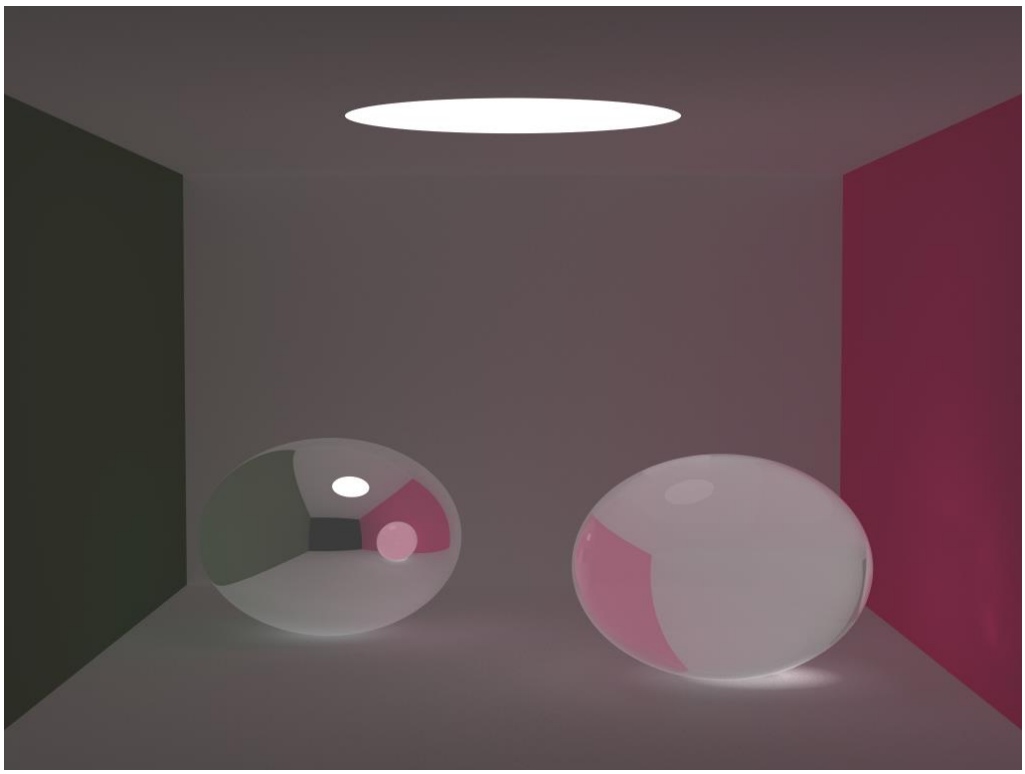
四、贴图

在球类的定义的 getcolor 函数中，当有纹理属性的时候，根据坐标映射返回一个加载的纹理上对应的点的颜色。

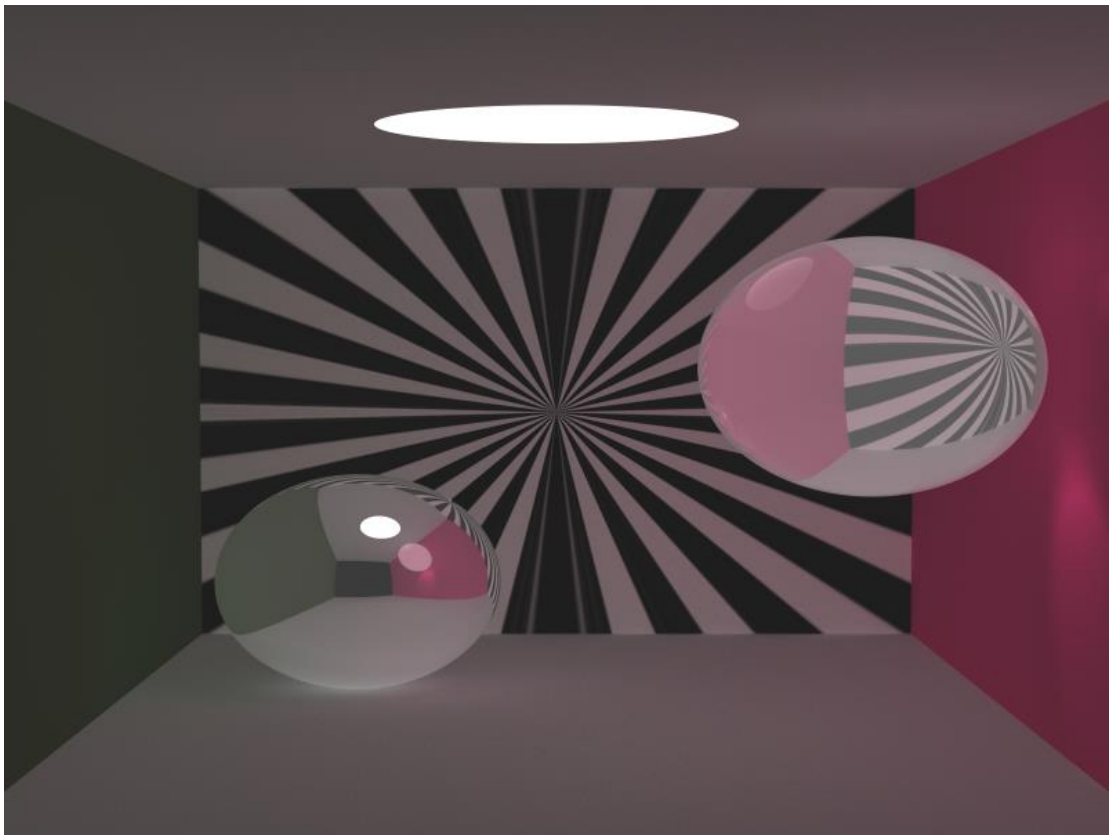
```
Vec3 getcolor(double x,double y,double z)const
{
    if (!hastexture)return color;
    x = abs(x - position.x); y = abs(y - position.y); z = abs(z - position.z);
    Vec3b color;
    if (x == 0 & y == 0)
        color = texture.at<Vec3b>(0, 0);
    else
        color=texture.at<Vec3b>(rows *asin(z/radius) / (2 * M_PI),cols *atan(y / x) / (2 * M_PI));
    return Vec3(color[0], color[1], color[2])*(0.00392);
}
```

五、成果展示

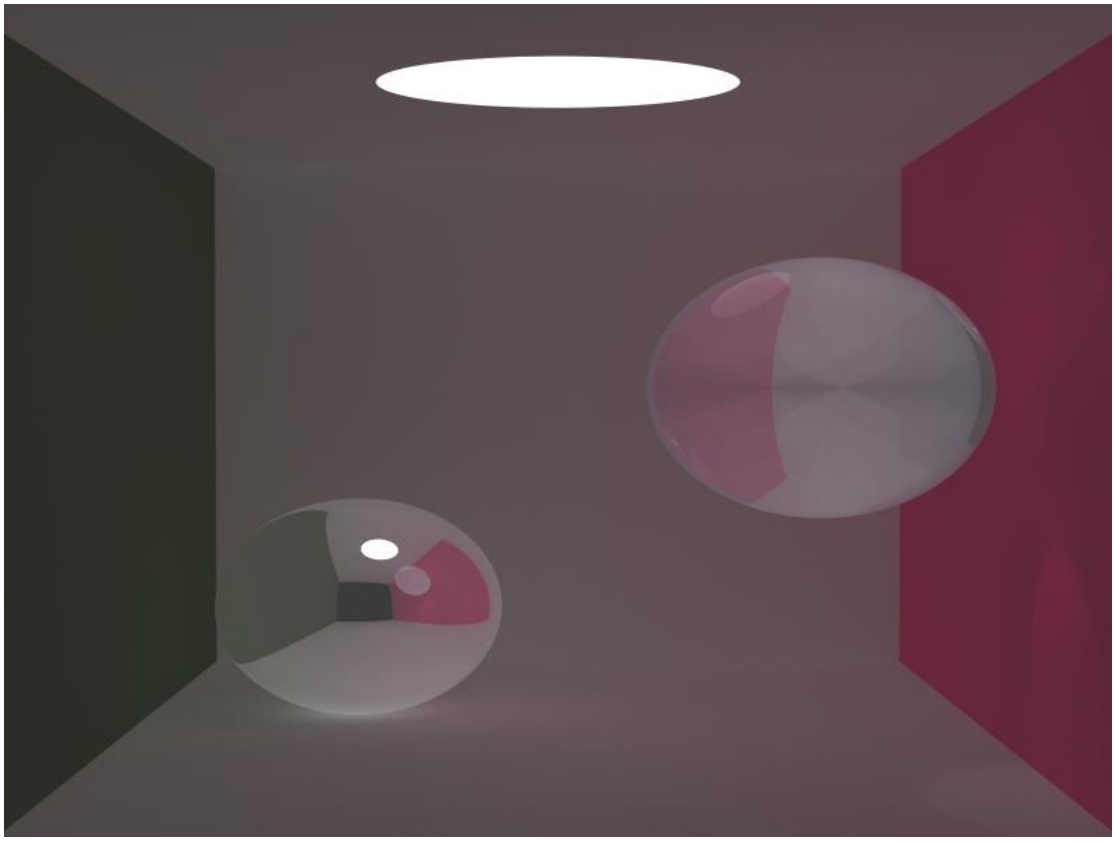
第一张图为初步实现的 PathTracing 效果图，第二张图是加上了二维纹理映射，将二维纹理映射到球体表面，第三张图是再次调整场景及相机参数后的效果图。



PathTracing 效果图



纹理映射效果图



重新调整参数后