图形学第三次作业报告

软73 沈冠霖 2017013569

1.使用方法

运行环境：

- Windows10
- VS2017
- OpenGL 4.6.0
- GLU工具库 1.2.2.0

运行方法：

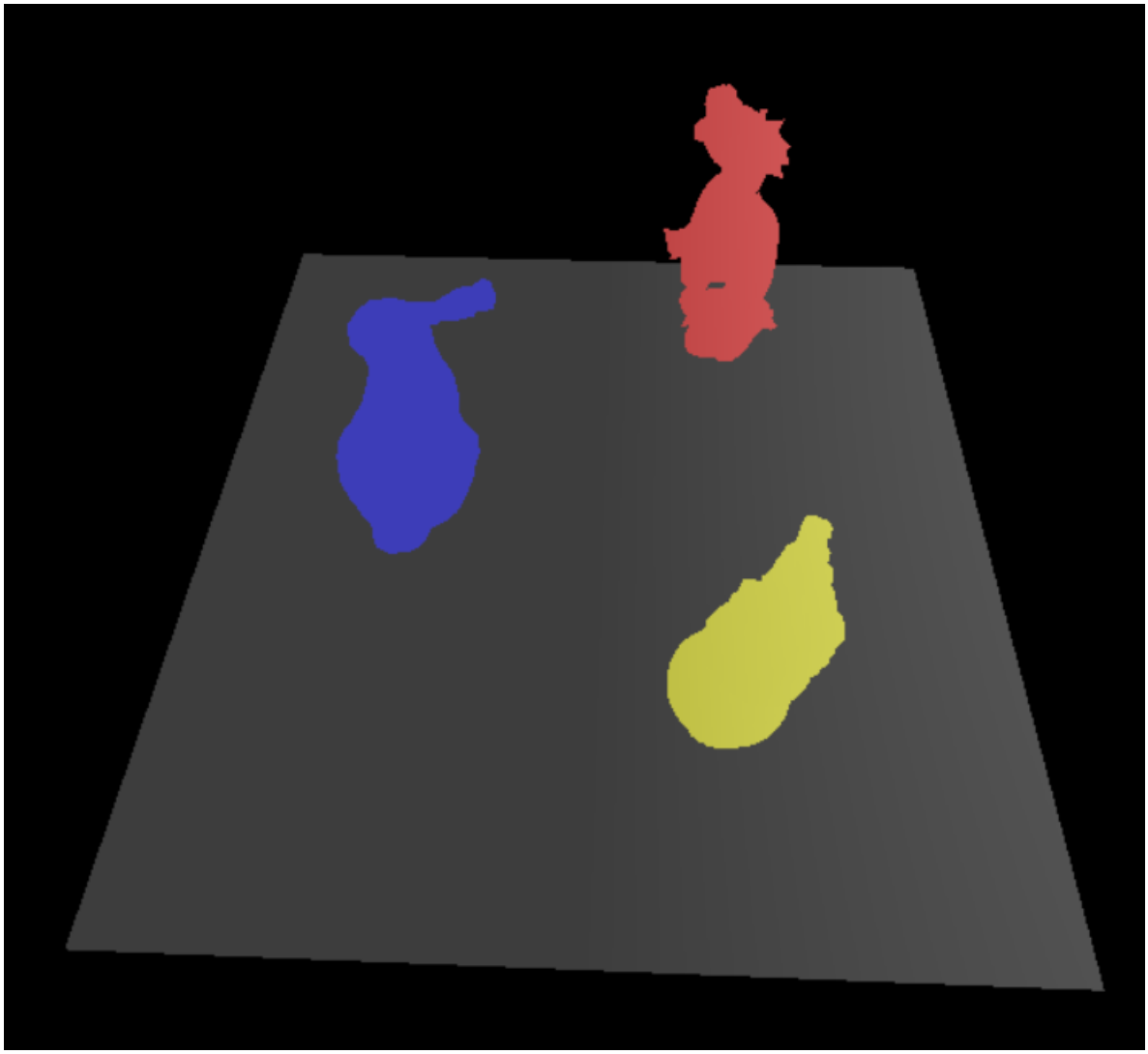
- 打开exe文件即可运行，选择1查看OpenGL的光照模型，选择2查看我自己实现的phong模型
- OpenGL光照模型和Phong模型都可以和作业2一样拖动鼠标/键盘变换视角
- 选择3使用基础光线追踪（较慢），选择4使用加速光线追踪，等待片刻之后生成光线追踪图片

2.实现原理

我的场景包括一个下方边界和三个ply面片模型。

2.1 Phong模型

OpenGL的glut库自带光照模型，只要设置光源的各个属性和物体的对应属性就可以显示。



但是因为GLUT无法读取光照模型每个点的光照，我还是自己实现了光照模型---不开启OpenGL光源，带入phong模型的公式来计算每个面片的颜色用于绘制。

$$I = I_a K_a + I_p K_d (L \cdot N) + I_p K_s (R \cdot V)^n$$

其中I_a为环境光颜色$Ambient$, K_a为物体每个面片的$Ambient$属性

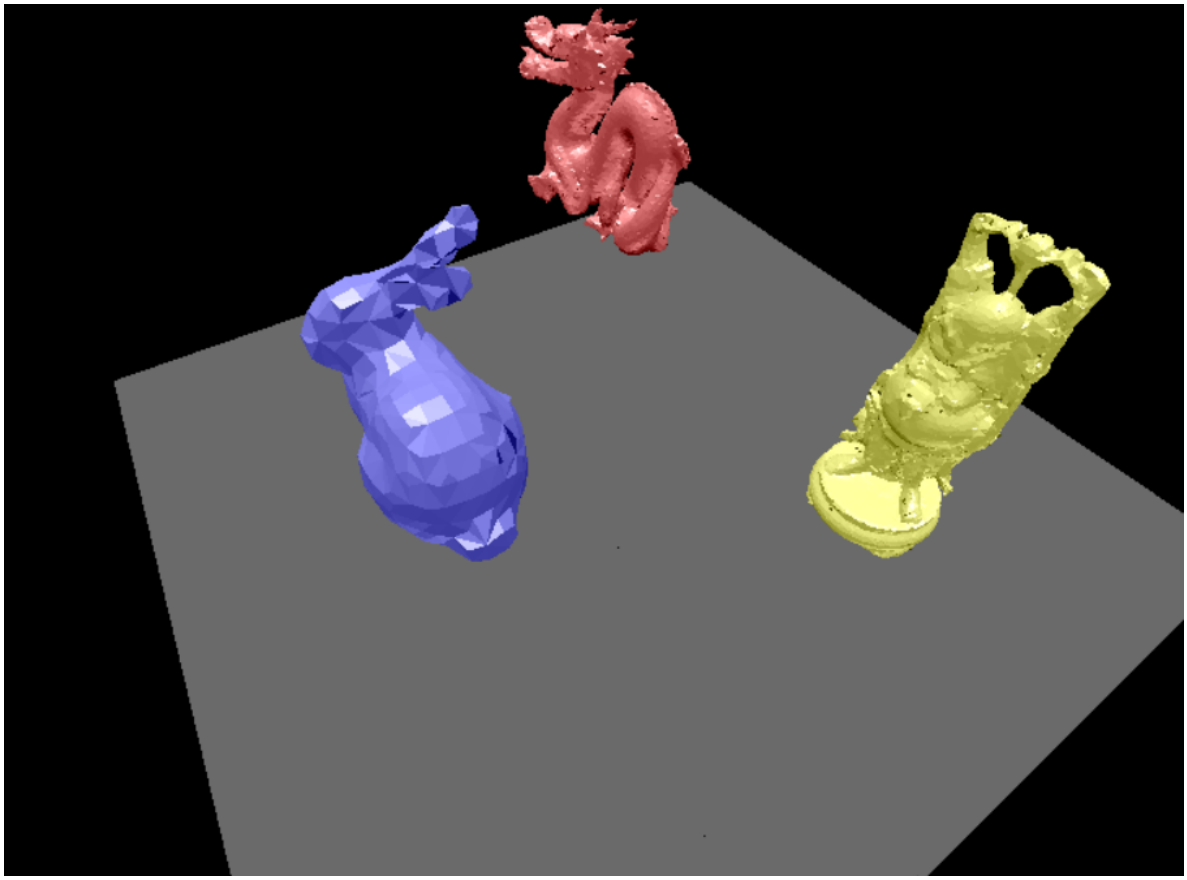
K_d为物体每个面片的$Diffuse$属性，K_s为物体每个面片的$Specular$属性

L为面片中心到光源的向量，因为光源在y轴无穷远，$L = (0, 1, 0)$

N为面片法向量，R为反射光线向量，$R = 2N(N \cdot L) - L$

V为视线方向，也就是$OpenGL$相机到视点中心的单位向量

结果如下：



2.2 基本光线追踪

光线追踪的伪代码如下：

```
Color RayTracing(start, direction, depth, weight)
{
    if(depth > MaxDepth || weight < MinWeight)
    {
        return black;
    }
    else
    {
        intersection = GetIntersection(); //求最近的交点
        if(intersection == NULL)
        {
            return black;
        }
        I_l = PhongModel(intersection); //Phong模型计算交点局部光强
        reflection = GetReflection(start, direction, intersection);//计算反射方向
        refraction = GetRefraction(start, direction, intersection);//计算折射方向
        I_s = RayTracing(intersection, reflection, depth + 1, weight * K_s);
        I_t = RayTracing(intersection, refraction, depth + 1, weight * K_t);
        return I_l + K_sI_s + K_tI_t;
    }
}
```

其中主要任务有5个：生成初始光线、求交、求反射方向、求折射方向、求局部光强。求局部光强用的是之前Phong模型的公式，这里不再赘述，只介绍其他四个部分。

2.2.1 生成初始光线

我们设相机位置到视角中心的向量为z轴，计算和z轴垂直的x，y轴，用x，y轴张成的平面作为视窗平面，然后计算视窗平面每个像素点在3D坐标系的坐标，以此为初始光线起点，以z轴方向为初始光线方向，得到的结果就是这个像素点的颜色。

2.2.2 求交

$$光线参数方程：start + t * direction$$

和边界求交很简单，只需要求光线所在直线和y=0平面的交点的t，如果交点在边框内且t>0，即可获得交点；

和面片求交也类似，先求光线所在直线和面片所在平面的交点的t，然后代入得到交点，通过累计角度法判断交点是否在面片内。如果交点在面片内且t>0，即可获得交点。

在基本光线追踪里，我的整体流程是，让光线和边界和每个面片都求交，找到t最小的合法交点。

2.2.3 求反射方向

求反射方向很简单，先求得面片/边界的法向量（0,1,0），然后求出光线方向的法向分量和切向分量，反射方向的法向分量取反，切向分量不变。

```
float dist = sqrt(Direction * m.Faces[i].Norm);
Point norm_speed = m.Faces[i].Norm * dist;
Point tangent_speed = Direction - norm_speed;
Point new_direction = tangent_speed - norm_speed;
```

2.2.4 求折射方向

折射稍微复杂一点：

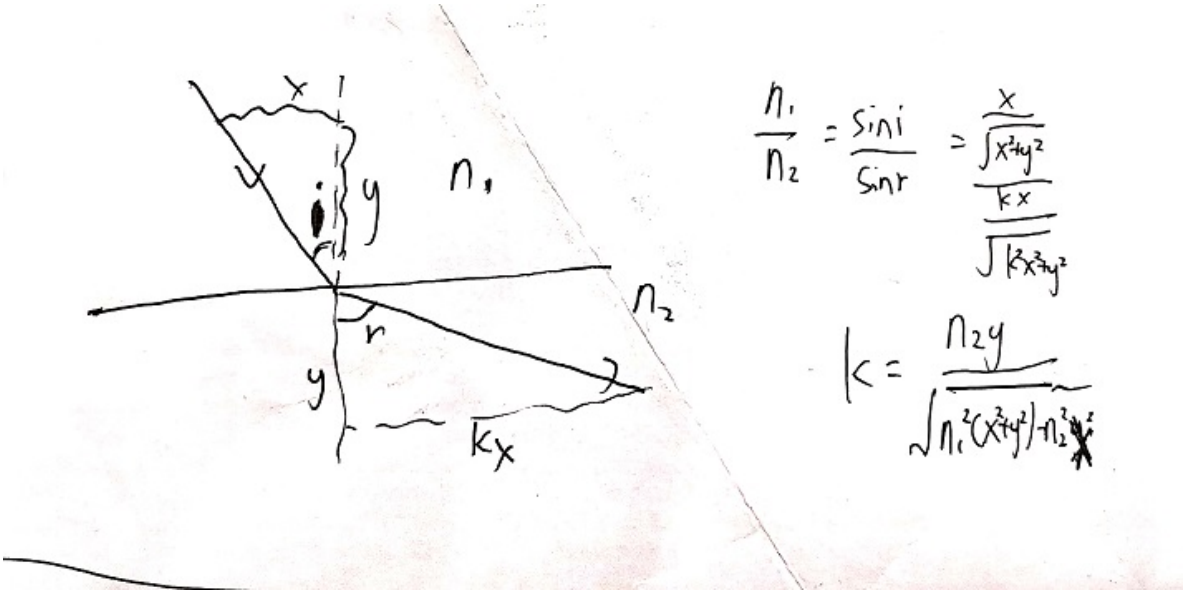
$$假设入射环境折射率为n_1,出射环境折射率为n_2；入射角为i，折射角为r$$

$$求得折射面法向量，将入射光方向分解为切向和法向，其欧氏距离分别为x和y$$

$$假设折射后切向法向的欧式距离分别为kx和y，则有：$$

$$\frac{n_1}{n_2} = \frac{sin(i)}{sin(r)} = \frac{\frac{x}{\sqrt{x^2+y^2}}}{\frac{kx}{\sqrt{k^2x^2+y^2}}}$$

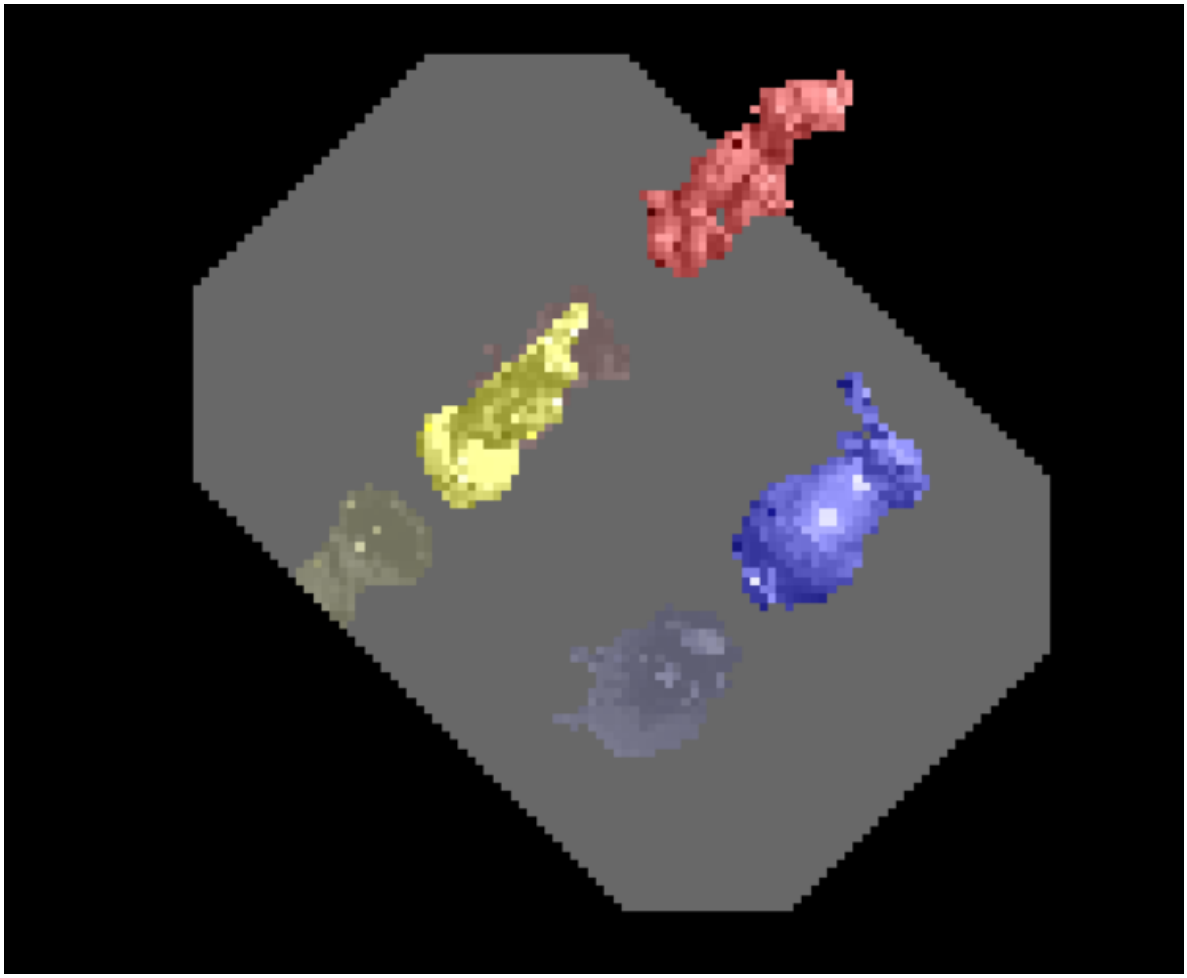
光路图如下：



$$带入求得 k = \frac{n_2 y}{\sqrt{n_1^2(x^2+y^2) - n_2^2 x^2}}$$

$$最终折射光线的单位方向向量为 \frac{k\overrightarrow{x} + \overrightarrow{y}}{\sqrt{k^2 x^2 + y^2}}$$

2.2.5 效果



因为基本光线追踪的速度很慢，我们只能渲染100*100的图片，但是也可以看出，我们能正确显示下方边界和三个面片模型，也能显示三个模型的反射情况，说明模型实现正确。

2.3 光线追踪加速

我们基本光线追踪的速度很慢，主要原因是求交：我们每个光线都和所有面片求交，但是其实需要求交的面片很少。

2.3.1 包围盒八叉树的递归建立

我的光线追踪加速方法结合了八叉树加速算法和包围盒加速算法。我对每个面片模型求了包围盒，然后在包围盒内部建立八叉树，让八叉树每个节点存储其x，y，z坐标上下界及其内部的面片（只要面片有一个点在八叉树节点内部就算）。

为了保证效果良好，我们保证八叉树深度不大于4，而且每个节点都能存储至少5个面片，这样能保证没有面片穿过八叉树节点，而且八叉树递归查找的开销不大。

建立八叉树的伪代码如下：

```
void BuildOctTree()
{
    if(depth >= MaxDepth || mesh_num <= MinNum) return;
    for(int i = 1; i <= 8; i ++)
    {
        Son[i] = BuildSon(i);
        Son[i].meshes = AssignMeshes(Son[i]);
        Son[i].BuildOctTree();
    }
}
```

2.3.2 优化后光线和面片模型的求交算法

在求交的时候，我们采用深度优先搜索的方法，查找所有和光线相交的八叉树叶子节点，将这些节点所有的面片加入求交列表。其中如果一个八叉树节点（非叶子结点或者叶子结点都是）和光线不相交，就不再向下搜索。

求交的伪代码如下：

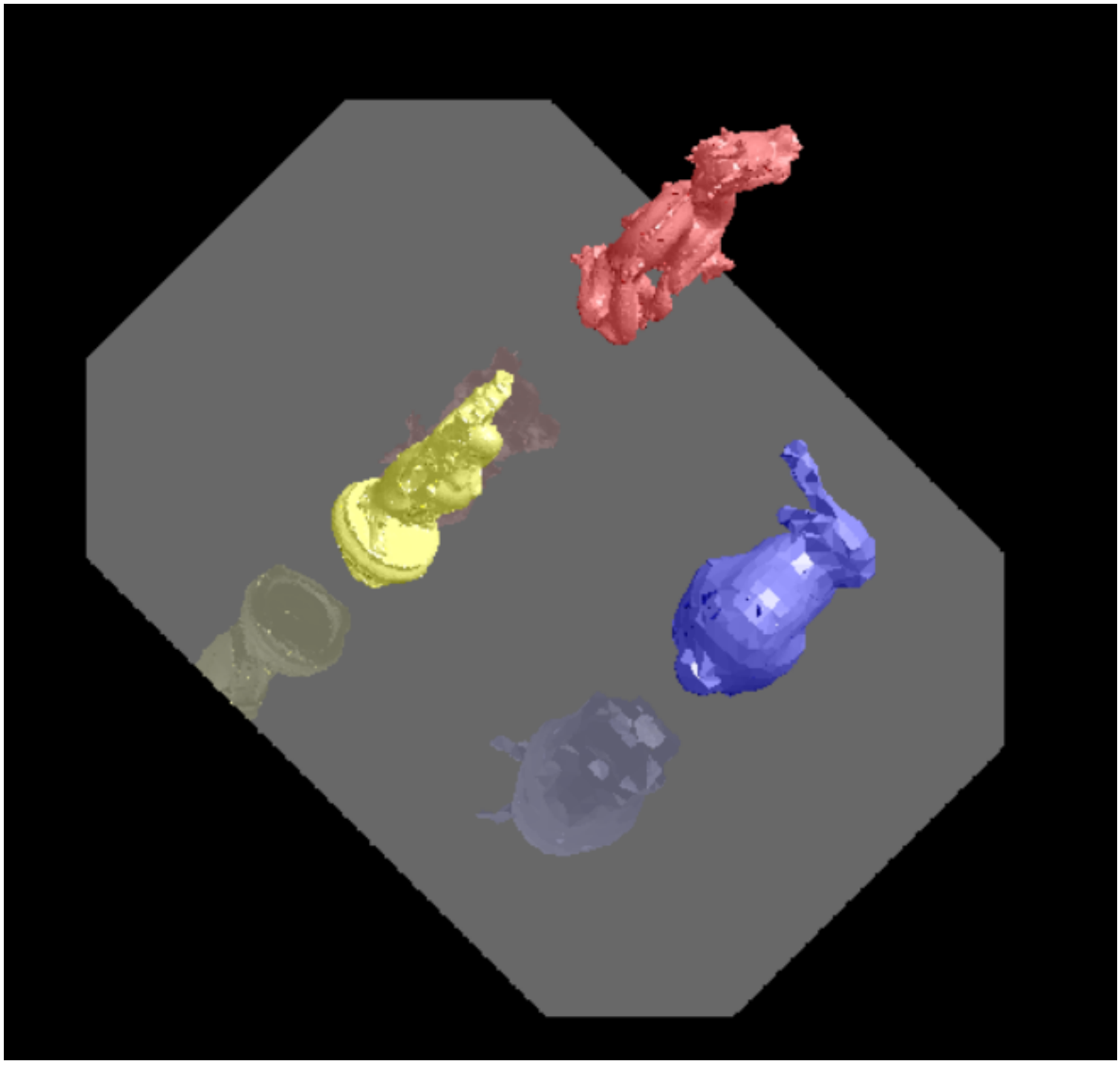
```
    /*
    描述：递归找包围盒里需要比较的面片
    参数：包围盒指针
    返回：在vector里添加面片和id
    */
    void GetPossibleMeshRecursive(vector<TriangleMesh>& mesh_list, BoundingBox*
m)
    {
        if (m == NULL) return;
        float t = GetIntersection(m);
        if (t <= 0) return; //如果和这个节点不相交，直接舍去（包括叶子或者非叶子，如果舍去
的是深度不大的节点，则会大大提高效率

        //非叶子结点递归深度优先搜索
        if (!m.IsLeafNode())
        {
            for (int i = 0; i < 8; i++)
            {
                GetPossibleMeshRecursive(mesh_list, m->Sons[i]);
            }
        }

        //叶子结点则直接加入面片
        else
        {
            for (int i = 0; i < m->MeshList.size(); i++)
            {
                mesh_list.push_back(m->MeshList[i]);
            }
        }
    }
```

其中，和包围盒求交方法也很简单，只需要求光线和包围盒对应6个平面的交点和对应的t，取交点在包围盒上且最小的t。

2.3.3 效果



可以看出，加速后和加速前效果并没有太明显的区别，说明加速算法正确。

同时，因为加速算法的有效性，在人可以忍受的等待时间内，原先只能迭代3层，生成100 * 100的低清晰度图片。现在可以迭代六七层，生成500 * 500的图片了。这样还改进了光线追踪的效果。

3.时间比较

我在Release环境下，迭代深度为3，生成图片为100*100情况下测试三次，结果如下：

运行次数/算法	基本光线追踪	加速光线追踪
1	209.843s	0.417s
2	205.703s	0.407s
3	200.555s	0.408s
平均	205.367s	0.411s

加速了499.676倍

可以看出，光线追踪的加速算法十分有效，成功将算法加速了近500倍。

4.总结和感想

这次作业，我深入理解了Phong模型、光线追踪算法和其加速，实现了光线和三角面片和包围盒的求交算法、以及反射折射光线的求解算法。除此之外我还了解了.ply格式的面片格式并且实现了其读取算法，通过实现八叉树优化锻炼了数据结构编程，提高了面向对象开发能力，收获很多。

但是这次作业实现也并非完美。因为算力和时间限制，我的模型过于简单，只有三个面片模型，没有纹理映射等进一步提高真实感的功能。同时，辐射度方法等其他的真实感绘制方法我也没有实现，希望之后能有机会更深入学习。