计算机图形学实验报告
Ray Tracing

课程名称：　计算机图形学

实验名称：　Ray Tracing

学生学院：　泰山学堂

学生班级：2017 级计算机取向

学生学号：　201705301350

学生姓名：　宋建涛

提交日期：　2019.12.15

目录

实验目标

本实验的主要要求是实现光线跟踪（RayTracing）。光线跟踪是一个在二维屏幕上呈现三维图像的方法。我们首先来看一下百度百科给出的光线追踪的定义：

一个光线跟踪程序数学地确定和复制从一幅图像的光线的路线,但是方向相反(从眼睛返回原点).光线跟踪现在被广泛用于计算机游戏和动画,电视和 DVD 制作,电影产品中.许多厂商提供用于个人电脑的光线跟踪程序.在光线跟踪中,每一个光线的路径由多重直线组成,几乎总是包含从原点到场景的反射,折射和阴影效应.在动画中,每一束光线的直线部分的位置和方向总是在不断变化,因此每一条光线都要用一个数学方程式来表示,定义光线的空间路径为时间的函数.根据光线在到达屏幕前经过的场景中的目标的色素或颜色来分配给每一束光线一种颜色.屏幕上的每一个像素符合每一时刻可以回溯到源头的的每条光线.光线跟踪最先是由一个叫数学应用组的组织中的科学家在 20 世纪 60 年代发明的.这些科学家中的一些人变得对光线跟踪作为一种艺术感兴趣,成为绘画艺术家,并建立了一个动画摄影工作室,使用光线跟踪为电视和电影制作 3D 电脑肖像和动画.

因为本实验网络上的代码虽然很多，但是使用 openGL 的代码却不多，找到的唯一一个有参考价值的代码运行结果却是很好。为了能够获得一个良好的运行结果，也为了能够更加深入的了解 RayTracing 的原理和机制，我选择了一篇没有使用 openGL，而是选择未使用第三方代码库的博客参考，但是实现和提交的代码还是 openGL 运行的结果

实验概览

根据网上搜索的资料和实验要求我们可以将本实验分为以下几部分：

创建环境

在本次试验中我们要对三个球体和一个椎体进行渲染，那么我们最先要做的就是将这些物体绘制出来，GLUT 提供了圆形的绘制函数 glutSolidSphere(radius, x,y)，我们只要提供半径 radius 就可以在图中画出一个球体（圆形），但是 GLUT 没有提供直接绘制锥体的函数，所以我们只能通过绘制四个三角面片来组装成一个三棱锥，因为代码大量重复，这里我们仅仅给出画出一个球体和一个锥体的代码，并给出对应的附加材质的代码。

```
1.  void DrawCenterSphere(double radius) {
2.
3.      glutSolidSphere(radius, 100, 100);
4.
5.      GLfloat material_Ka[] = { 0, 0, 0, 1.0f };
6.      GLfloat material_Kd[] = { 0, 0, 0, 1.0f };
7.      GLfloat material_Ks[] = { 0.8f, 0.8f, 0.8f, 1.0f };
8.      GLfloat material_Se = 32.0f;
```

```
9.
10.     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, material_Ka);
11.     glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_Kd);
12.     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, material_Ks);
13.     glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, material_Se);
14. }
15.
16. void DrawTriangle() {
17.
18.     glBegin(GL_TRIANGLES);
19.
20.     GLfloat material_Ka[] = { 0.0f, 0.0f, 0.0f, 1.0f };
21.     GLfloat material_Kd[] = { 0.94, 0.75, 0.31, 1.0f };
22.     GLfloat material_Ks[] = { 0.2, 0.2f, 0.2f, 1.0f };
23.     GLfloat material_Se = 30.0f;
24.
25.     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, material_Ka);
26.     glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_Kd);
27.     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, material_Ks);
28.     glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, material_Se);
29.
30.     glNormal3f(2, 1, 2);
31.     glVertex3f(2.0, 0, -25.0);
32.     glVertex3f(7, -10, -25);
33.     glVertex3f(2, -10, -20);
34.
35.     glNormal3f(-2, 1, 2);
36.     glVertex3f(2.0, 0, -25.0);
37.     glVertex3f(2, -10, -20);
38.     glVertex3f(-3, -10, -25);
39.
40.     glNormal3f(0,0,1);
41.     glVertex3f(2.0, 0, -25.0);
42.     glVertex3f(-3, -10, -25);
43.     glVertex3f(7, -10, -25);
44.     glEnd();
45. }
```

光线

本实验的名称为光线追踪，那么光线的创建一定是一个重点，我们在这里给出光线的类定义和光线的绘制函数。

```
1.  class Light {
```

```
public:
    Vector position;
    Vector color;
    Light(Vector p, Vector c) : position(p), color(c) {}//float t, float d, float a, float s);

    bool isHit(Vector & rayVector, Vector& rayOrigin);
    Vector intersectPoints(const Vector &rayVector, const Vector &rayOrigin);
    Vector normalPoint(Vector hitPoint);
};
void DemoLight() {

    GLfloat light_pos[] = { 0, 9, -25, 1.0f };
    GLfloat light_Ka[] = { 0, 0, 0, 1.0f };
    GLfloat light_Kd[] = { 0.65, 0.65, 0.65, 1.0f };
    GLfloat light_Ks[] = { 0.7, 0.7, 0.7, 1.0f };

    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_Ka);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_Kd);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_Ks);

    GLfloat light_pos1[] = { 0,9,-15,1.0f };

    glLightfv(GL_LIGHT1, GL_POSITION, light_pos1);
    glLightfv(GL_LIGHT1, GL_AMBIENT, light_Ka);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light_Kd);
    glLightfv(GL_LIGHT1, GL_SPECULAR, light_Ks);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
    glEnable(GL_NORMALIZE);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}
```

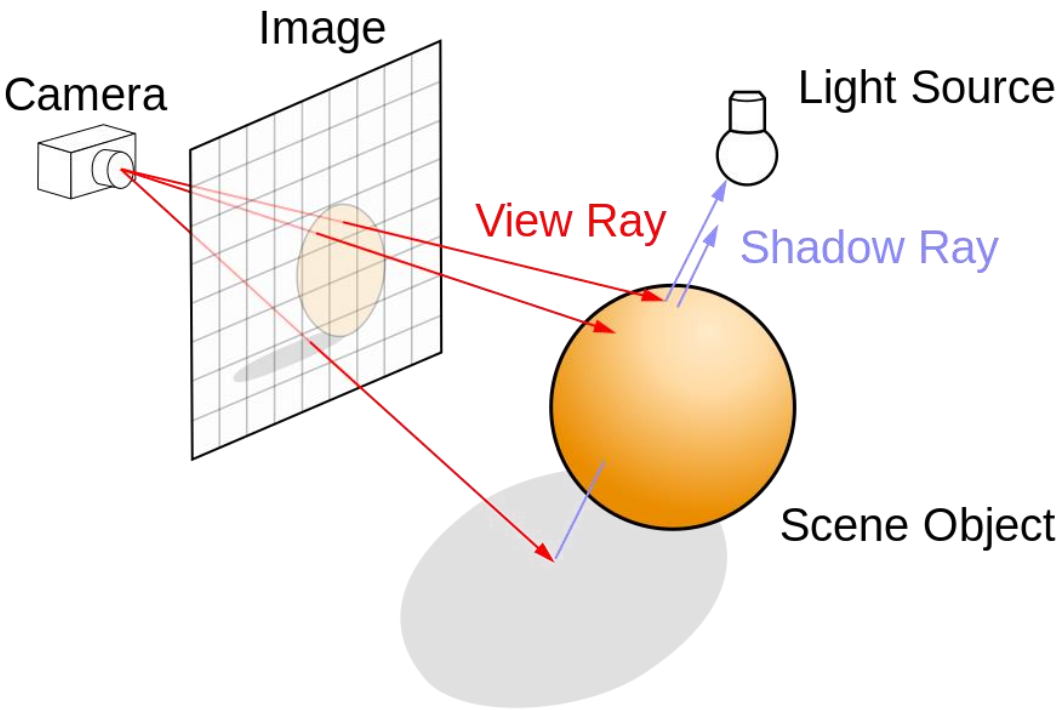
下边这段代码的中唯一重要的部分是一个允许你检测一个给定光线（源于 orig 射向 dir ）是否与我们的球体相交的函数。

```
struct Sphere {
    Vec3f center;
```

```
3.      float radius;
4.
5.      Sphere(const Vec3f &c, const float &r) : center(c), radius(r) {}
6.
7.      bool ray_intersect(const Vec3f &orig, const Vec3f &dir, float &t0) const
    {
8.          Vec3f L = center - orig;
9.          float tca = L*dir;
10.         float d2 = L*L - tca*tca;
11.         if (d2 > radius*radius) return false;
12.         float thc = sqrtf(radius*radius - d2);
13.         t0       = tca - thc;
14.         float t1 = tca + thc;
15.         if (t0 < 0) t0 = t1;
16.         if (t0 < 0) return false;
17.         return true;
18.     }
19. };
```

相机

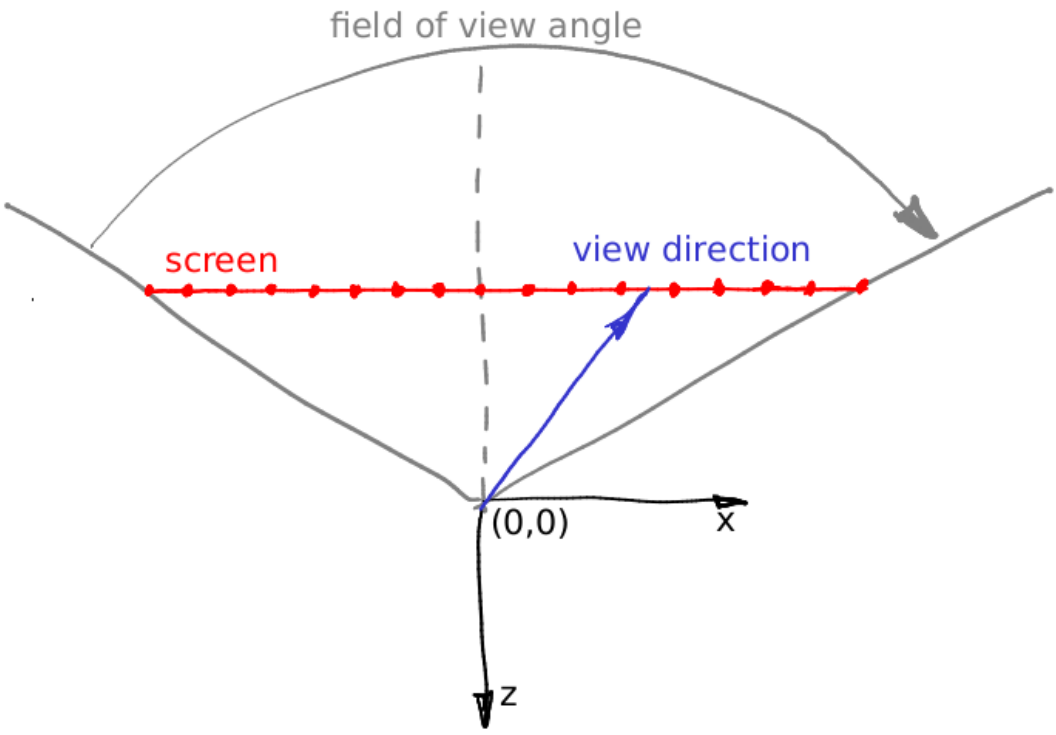
现在对于每个像素，我们生成一束从源头射出的穿过该像素的光线，之后检查该光线与球体是否相交:



如果没有和球体相交我们就用第一种颜色画出该像素，否则的话就用另外一种颜色画出这个像素。

下面，为了确定我们摄像机的参数，我们需要确定这么几个因素：图像宽度、图像高度、视场角、摄像机位置、观察方向（默认沿 z 轴负方向）。博客中给出了计算要追踪光线的初始方向的计算公式：

```
1.  float x =  (2*(i + 0.5)/(float)width  - 1)*tan(fov/2.)*width/(float)height;
2.  float y = -(2*(j + 0.5)/(float)height - 1)*tan(fov/2.);
```



由图片我们可以比较清晰的看出来式子的来历。摄像机放置在原点且面向 z 轴负方向。上面这幅图显示了从摄像机上方向下看的情形，y 轴指向屏幕外。

摄像头放置在原点， 场景被投影在位于 z=-1 平面的屏幕上。视场角决定了能显示在屏幕上的区域。在上图中屏幕宽度为 16 像素（译注：一段红线段为一个像素），你能计算出它在世界坐标系中的长度吗？很简单：让我们把焦点放在由红线、灰线以及灰色虚线组成的三角形上。很容易看出 tan(field of view / 2) = (screen width) * 0.5 / (screen-camera distance)。我们把屏幕放在离摄像机距离为 1 的地方，这样 (screen width) = 2 * tan(field of view / 2)。

现在我们想把一个向量投射到距屏幕左边第十二个像素的中心处，也就是我们想计算出图中的蓝色向量该怎么做？从屏幕左边到向量箭头处的距离是多少？首先，这个距离是 12 + 0.5 像素。我们知道屏幕上 16 个像素对应于 2 * tan(fov/2) 个世界单位。也就是说向量的箭头处位于从左边数起第 (12+0.5)/16 * 2*tan(fov/2) 个世界单位处，或者说从屏幕和 z 轴交点处起向右 (12+0.5) * 2/16 * tan(fov/2) - tan(fov/2) 个世界单位。再屏幕纵横比的影响算上，最终就得到了计算光线方向的式子。

加入反射和折射

在光学中，反射和折射是总所周知的现象。反射和折射分向都是基于相交点处的法线和入射光线（主光线）的方向。为了计算折射方向，我们还需指定材料的折射率。

同样，我们也必须意识到像玻璃球这样的物体同时具有反射性和折射性的事实。我们需要为表面上的给定点计算两者的混合值。反射和折射具体值的混合取决于主光线（或观察方向）和物体的法线和折射率之间的夹角。有一个方程式精确地计算了每个应该如何混合，这个方程被称为菲涅耳方程。

反射/折射基本原理

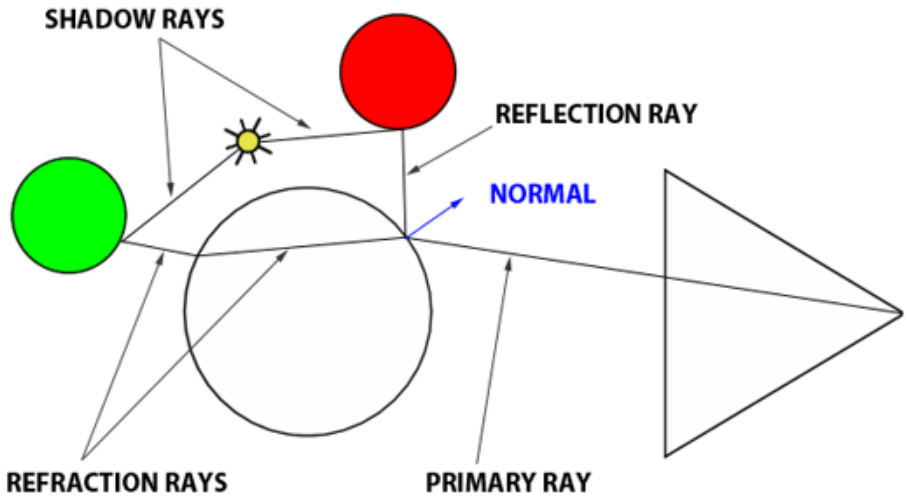
反射：为此，我们需要两个项：交点处的法线和主光线的方向。一旦我们获得了反射方向，我们就朝这个方向发射新的光线。我们假设反射光线撞击了红色球体，通过向光线投射阴影射线来找出到达红色球体上的那个点的光线多少。这会得到一种颜色（如果是阴影，则为黑色），然后乘以光强并返回到玻璃球的表面。

折射：注意，因为光线穿过玻璃球，所以它被认为是透射光线（光线从球体的一侧传播到另一侧）。为了计算透射方向，我们需要在知道击中点的法线，主射线方向和材料的折射率。

当光线进入并离开玻璃物体时，光线的方向会改变。每当介质发生变化时都会发生折射，而且两种介质具有不同的折射率。折射对光线有轻微弯曲的作用。这个过程就是让物体在透视时或在不同折射率的物体上出现偏移的原因。

现在让我们想象一下，当折射的光线离开玻璃球时，它会碰到一个绿色的球体。在那里，我们再次计算绿色球体和折射射线之间交点处的局部照明（通过拍摄阴影射线）。然后，将颜色（如果被遮挡，则为黑色）乘以光强并返回到玻璃球的表面。

应用菲涅尔方程

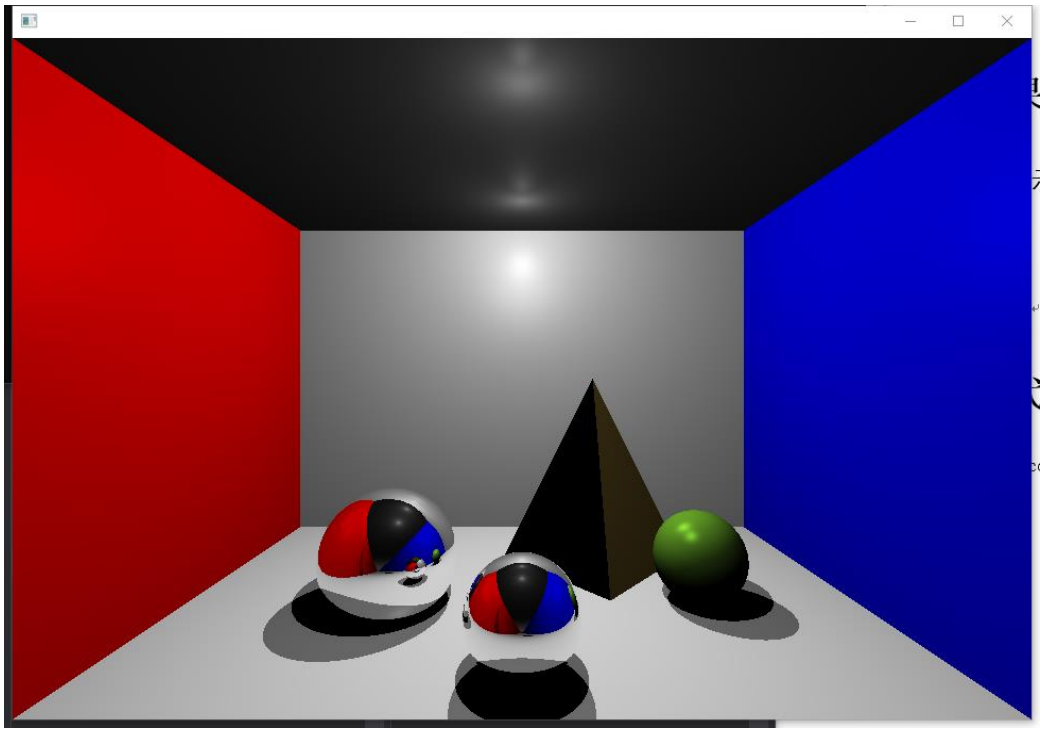


这种算法的美妙之处在于它是递归的。迄今为止，在我们研究过的情况下，反射光线照射到一个红色的、不透明的球体上，而折射光线照射到一个绿色的、不透明的和漫射的球体上。但是，我们会想象红色和绿色的球体也是玻璃球。为了找到由反射和折射光线返回的颜色，我们必须按照与原始玻璃球一起使用的红色和绿色球体的相同过程。

这是光线追踪算法的一个严重缺陷。想象一下，我们的相机是在一个只有反射面的盒子里。从理论上讲，光线被困住了，并且会持续不断地从箱子的墙壁反弹（或者直到你停止模拟）。出于这个原因，我们必须设置一个任意的限制值，从而防止光线相互作用导致的无限递归。每当光线反射或折射时，其深度都会增加。当光线深度大于最大递归深度时，我们就停止递归过程。

实验结果和收获

实验结果展示



反思和收获

本次实验真的是暴露出了我代码实现上的很大的问题，原理的理解还好，从 RayCasting 大作业结束之后抽空补了一下矩阵相乘的知识，这次咋推导公式的时候不再满头雾水，但是看明白不等于能够自己推导、也不等于能够用代码实现，这次的实验参考的博客很多而且很杂，从使用 openGL 的代码到不适用 openGL 的代码，再到 256 行的极简代码。这么多的代码阅读确实给了我很大的启发，理解原理的过程主要是靠没有使用

openGL 的代码，但是在阅读和实现过程中发现这个没有使用 openGL，于是在自己实现的过程中又主要参考了其他的代码。关于 GLUT 的使用还是要长时间的摸索，一个学期使用边角料的时间来学习是远远不够的。

实验源代码

https://github.com/yuemos/Computer-Graphics.git