计算机图形学实验报告
Ray Casting

课程名称：　　计算机图形学

实验名称：　　Ray Casting

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：　　2019.11.30

目录

实验目标 3
实验概览 3
    创建体数据 3
    获得旋转矩阵 4
    体数据分类 6
    合成像素值 7
    显示图像 9
实验代码结构 9
实验结果和收获 10
    实验结果展示 10
    实验收获 10
实验源代码 11

实验目标

本实验的主要目的是实现光线投射算法，主要的要求有两个：

1. 从特定的角度投射光线
2. 实现光线与图元的相交效果

这个功能将能够对实现许多多边形的碰撞产生的光线效果起加速作用。光线投射主要用于如三维电脑游戏以及动画等实时模拟场合，在这些场合下，细节并不太重要或者是通过人为制造细节可以得到更好的计算效率。通常在需要多帧图像生成动画时就会出现这种情况。如果不使用其它的技巧，这种方法得到的物体表面通常看起来比较扁平，就好像场景中的物体都是经过光滑处理的糙面。

建好的几何模型从外部观察点逐点、逐线进行分析，就如同是从观察点投射出光线一样。当光线与物体交叉的时候，交叉点的颜色可以用几种不同的方法来计算。其中最简单的方法是用交叉点处物体的颜色表示该点的实际颜色；也可以用纹理映射的方法来确定；一种更加复杂的方法是仅仅根据照明因数变更颜色值，而无需考虑与模拟光源的关系。为了减少人为误差，可以对多条相邻方向的光线进行平均。

另外也可以对光的特性进行大致的模拟：简单计算从物体到观察点的光线。另外的一些计算涉及到从光源到物体的入射角，以及根据光源的强度计算像素的亮度值。此外还有一些模拟使用辐射着色算法绘制的照明结果，或者使用这两种信息的组合。

实验概览

要想完成本节实验，我们同样可以把整个大项目分为几个小项目：

1. 创建本实验要用到的三维物体
2. 获得旋转矩阵
3. 对体数据进行分类
4. 逐个合成像素值
5. 显示图像

创建体数据

在本次试验中，我们可以使用一些比较简单的模型来模拟 RayCasting，这里我们选择两个正方体，一个球体的组合，从外到内分别是正方体、球体、正方体，其中相邻的两个物体内切。

在这里我们封装两个函数来分别表示建立球体和正方体的过程，然后再封装一个函数来调用这两个函数来创建球体和正方体。在这里我们不能直接用 GLUT 当中建立球体和正方体模型的函数，因为在之后的 RayCasting 中我们要用到我们这里的数据，所以我们用一个数据结构 Data 来存储我们的体数据，Dim 存储体数据大小。

```
1.  //生成体数据
2.  void CreateVolume(int *Data, int *Dim)//data 代表体积数据，Dim 代表体数据大小
```

```
{
    CreateCube(0, 0, 0, 200, 100, Data, Dim);//大正方体
    CreateSphere(100, 100, 100, 80, 200, Data, Dim);//球体
    CreateCube(70, 70, 70, 60, 300, Data, Dim);//小正方体
}

//生成正方体数据
void CreateCube(int x, int y, int z, int side, int density, int *Data, int *Dim)//x,y,z 代表正方体左下角坐标，side 代表边长，density 代表对应的标量值
{
    int max_x = x + side, max_y = y + side, max_z = z + side;
    int Dimxy = Dim[0] * Dim[1];
    for (int k = z; k < max_z; k++){
        for (int j = y; j < max_y; j++){
            for (int i = x; i < max_x; i++){
                Data[k*Dimxy + j * Dim[0] + i] = density;
            }
        }
    }
}

//生成球体数据
void CreateSphere(int x, int y, int z, int radius, int density, int *Data, int *Dim)//x,y,z 代表正方体左下角坐标，side 代表边长，density 代表对应的标量值
{
    int radius2 = radius * radius;
    int Dimxy = Dim[0] * Dim[1];
    for (int k = 0; k < Dim[2]; k++){
        for (int j = 0; j < Dim[1]; j++){
            for (int i = 0; i < Dim[0]; i++){
                if ((i - x)*(i - x) + (j - y)*(j - y) + (k - z)*(k - z) <= radius2){
                    Data[k*Dimxy + j * Dim[0] + i] = density;
                }
            }
        }
    }
}
```

获得旋转矩阵

为了获得我们的从图像空间到物体空间的旋转矩阵，我们需要两个函数来辅助，分别是表示向量叉乘和向量归一化的函数。

```
//向量叉乘（c=a X b)
void CrossProd(float *c, float *a, float *b)//c:输出向量,a:输入向量,b:输入向量
{
    float x, y, z;
    x = a[1] * b[2] - b[1] * a[2];
    y = a[2] * b[0] - b[2] * a[0];
    z = a[0] * b[1] - b[0] * a[1];
    c[0] = x;
    c[1] = y;
    c[2] = z;
}

//向量归一化
void Normalize(float *norm, float *a)//norm:归一化结果,a:输入向量
{
    float len = sqrt(a[0] * a[0] + a[1] * a[1] + a[2] * a[2]);
    norm[0] = a[0] / len;
    norm[1] = a[1] / len;
    norm[2] = a[2] / len;
}
```

之后我们就可以开始计算旋转矩阵了。在这里我们首先要表示出图像空间的基向量，我们知道有三个基向量就可以表示出空间中的所有向量，这里我们用三个数组 XX[3]、YY[3]、ZZ[3]来存储这三个向量，计算 z 轴我们可以用我们视点的坐标减去观察物体的坐标来计算：

```
ZZ[0] = eye[0] - center[0];
ZZ[1] = eye[1] - center[1];
ZZ[2] = eye[2] - center[2];
```

之后我们通过向量叉乘求得我们的 x 坐标和 y 坐标：

```
CrossProd(XX, up, ZZ);
CrossProd(YY, ZZ, XX);
```

到目前为止三个基向量我们都已经求得，但是作为基向量我们希望它的模长为 1，所以我们用向量归一化函数来规范化我们的基向量：

```
Normalize(XX, XX);
Normalize(YY, YY);
Normalize(ZZ, ZZ);
```

最后我们将我们计算得到的向量填入矩阵中就得到了我们的旋转矩阵：

```
//由图像空间基向量构成旋转矩阵
```

```
R[0] = XX[0]; R[1] = YY[0]; R[2] = ZZ[0];
R[3] = XX[1]; R[4] = YY[1]; R[5] = ZZ[1];
R[6] = XX[2]; R[7] = YY[2]; R[8] = ZZ[2];
```

我们想要得到的旋转矩阵 R 计算完成，在之后的计算中我们就可以利用旋转矩阵计算从图像空间到物体空间的转换。

体数据分类

在第一步建立完提数据之后我们就要开始对体数据进行分类了。

在这里，我们将原始体数据的标量值映射为颜色和不透明度，主要操作就是将之前生成的数据分为三类：大正方体、球体、小正方体。并且我们将大正方体用白色表示，球体用红色表示，小正方体用黄色表示。

```
//数据分类
void Classify(float* CData, int *Data, int *Dim)//CData:分类后体数据
{
    int *LinePS = Data;
    float *LinePD = CData;
    //将原始体数据的标量值映射为颜色和不透明度
    //这里处理的比较简单，直接将之前生成的数据分三类：大正方体白色、球体红色、小正方体黄色
    for (int k = 0; k < Dim[2]; k++){
        for (int j = 0; j < Dim[1]; j++){
            for (int i = 0; i < Dim[0]; i++){
                if (LinePS[0] <= 100){
                    //白色
                    LinePD[0] = 1.0;
                    LinePD[1] = 1.0;
                    LinePD[2] = 1.0;
                    LinePD[3] = 0.005;
                }
                else if (LinePS[0] <= 200){
                    //红色
                    LinePD[0] = 1.0;
                    LinePD[1] = 0.0;
                    LinePD[2] = 0.0;
                    LinePD[3] = 0.015;
                }
                else{
                    //黄色
                    LinePD[0] = 1.0;
                    LinePD[1] = 1.0;
                    LinePD[2] = 0.0;
```

```
                    LinePD[3] = 0.02;
                }
                LinePS++;
                LinePD += 4;
            }
        }
    }
}
```

合成像素值

在这里我们对图片的每一个像素点合成像素值，首先我们要将图片上的像素点遍历一遍，并且对每一个像素点进行处理：

```
float *LinePS = Image;
for (int j = 0; j < HEIGTH; j++)//逐个合成像素值
{
    for (int i = 0; i < WIDTH; i++)
    {
        Composite(LinePS, i, j, CData, Dim, R, T);
        LinePS += 4;
    }
}
```

在每一个像素点计算过程的内部，我们需要存储下采样步长、累计颜色值、投射光线的起点和方向等信息。

因为我们采用平行投影，所谓哦们在土星空间中投射光线的方向是(0,0,-1)，起点是(x0,y0,0),然后我们利用之前计算出的旋转矩阵 R 将光纤描述转换到物体空间中：

```
pos[0] = x0; pos[1] = y0; pos[2] = 0;
//将光线描述转换到物体空间
dir[0] = -R[2]; dir[1] = -R[5]; dir[2] = -R[8];//光线方向在物体空间的表达
MatrixmulVec(pos, R, pos);//旋转
pos[0] += T[0];//平移
pos[1] += T[1];
pos[2] += T[2];
```

我们看到在上边的计算过程中，我们用到了向量和矩阵相乘的计算函数 MatrixmulVec()，下边我们给出计算函数的定义：

```
//矩阵与向量乘积(c=a*b)
void MatrixmulVec(float *c, float *a, float *b)//c:输出向量,a:输入向量,b:输入向量
{
```

```
    float x, y, z;
    x = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
    y = a[3] * b[0] + a[4] * b[1] + a[5] * b[2];
    z = a[6] * b[0] + a[7] * b[1] + a[8] * b[2];
    c[0] = x;
    c[1] = y;
    c[2] = z;
}
```

之后我们需要判断一下光线与包围盒是否相交，这里我们使用的就是课堂上讲的将面无限扩展，然后判断点的位置，具体代码见最后一节实验源代码。

之后，当光线没有射出包围盒并且累计不透明度没有超过 1 时，我们持续利用三线性差值获得采样点的颜色和不透明度，然后使用从前到后的合成公式合成颜色和不透明度，然后对下一个采样点进行处理。

当光线射出包围盒并且累计不透明度超过 1 时，我们对当前像素点的计算就结束了，我们将计算结果存入 rgba 数组并且返回，

```
while (CheckinBox(samplepos, Dim) && cumcolor[3] < 1)//当光线射出包围盒或累计不透明度超过 1 时中止合成
    {
        TrInterpolation(samplecolor, samplepos, CData, Dim);//三线性插值获得采样点处的颜色及不透明度
        //合成颜色及不透明度,采用的是从前到后的合成公式
        cumcolor[0] += samplecolor[0] * samplecolor[3] * (1 - cumcolor[3]);//R
        cumcolor[1] += samplecolor[1] * samplecolor[3] * (1 - cumcolor[3]);//G
        cumcolor[2] += samplecolor[2] * samplecolor[3] * (1 - cumcolor[3]);//B
        cumcolor[3] += samplecolor[3] * (1 - cumcolor[3]);              //A

        //下一个采样点
        samplepos[0] += dir[0] * stepsize;
        samplepos[1] += dir[1] * stepsize;
        samplepos[2] += dir[2] * stepsize;
    }
    rgba[0] = cumcolor[0];
    rgba[1] = cumcolor[1];
    rgba[2] = cumcolor[2];
    rgba[3] = cumcolor[3];
    return;
}
```

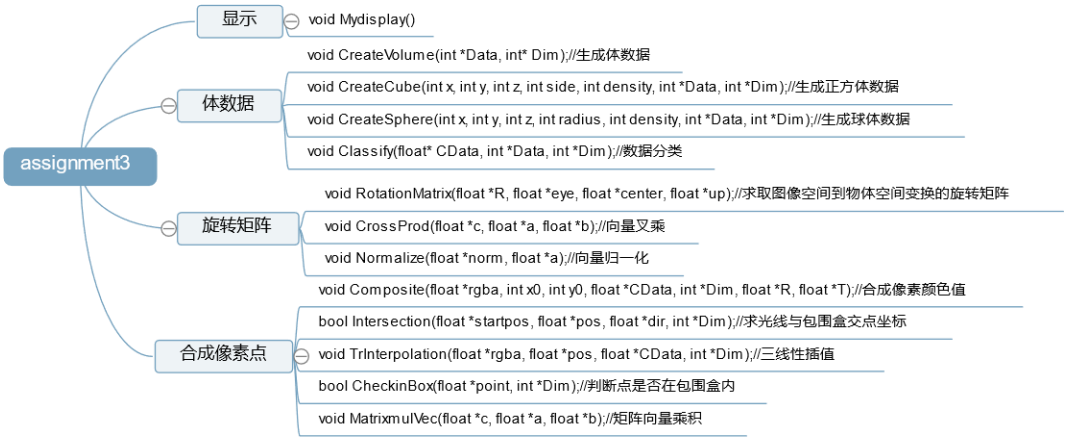
当以上的部分全部执行完毕后，我们对于每一个像素点都计算出了它的 RGB 值，RayCasting 方法运行完毕。

显示图像

每一个点的像素值都计算结束了，将结果显示出来就边的简单多了，我们可以调用 GLUT 的函数直接来显示计算结果即可。

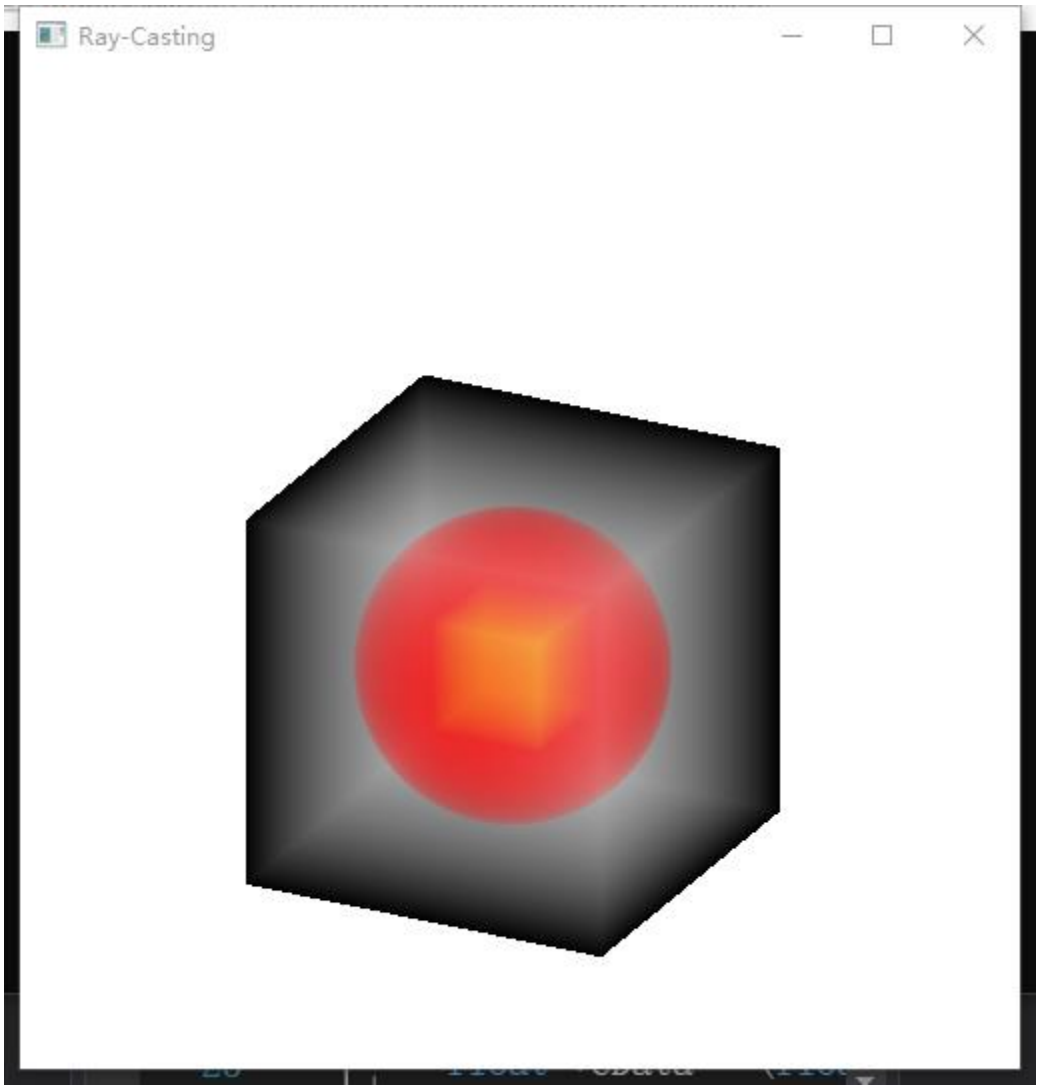
```
1. void Mydisplay()
2. {
3.     glClear(GL_COLOR_BUFFER_BIT);
4.     glDrawPixels(WIDTH, HEIGTH, GL_RGBA, GL_FLOAT, Image);//使用 OpenGL 的绘图函数
5.     glFlush();
6. }
```

实验代码结构



实验结果和收获

实验结果展示



实验收获

通过本次实验，我了解了一些关于 GLUT 的使用技巧，虽然对 RayCasting 的实验原理有了比较清晰的理解，但是在代码实现的过程中还是遇到了很多问题，包括公式推导和像素点值计算的过程中，数学相对来说还是一个短板，我还是需要多补习一下数学的相关知识，尤其是矩阵计算必须要尽快复习一下。

实验源代码

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