院系：数据科学与计算机　　专业：计算机类　　科目：计算机图形学
姓名：郑康泽　　学号：17341213

计算机图形学

一. 作业题目

使用GL_POINTS绘制固定轨道运动的小球

- 每个glVertex调用指明一个小球的球心位置
- 小球大小根据观察点距离变化（近大远小）
- 使用Phong Shading

二. 实现过程及代码

1. 顶点着色器
    1. 计算点在裁剪坐标的位置：

        利用代码`gl_Postion = gl_ModelViewProjectionMatrix * gl_Verter`，其中`gl_Position`表示的就是点在裁剪坐标系中的坐标，即`gl_Position.x`、`gl_Position.y`、`gl_Position.z`都$\in [0,1]$，`gl_ModelViewProjectionMatrix`是`Projection`矩阵右乘上`ModelView`矩阵得到的，`gl_Vertex`是点在世界坐标系的坐标。

    2. 实现近大远小的效果：

        利用`gl_PointSize`控制方形点区域渲染像素大小。一般相机的朝向是从z轴的正方向看向z轴的负方向，所以当`gl_Position.z`越小说明里相机越远，当`gl_Position.z`越大说明里相机越近。所以实现近大远小的代码为`gl_PointSize = radius * (1 + gl_Position.z)`，radius就是控制大小的因子，可以为任意合适的值，`1 + gl_Position.z`是为了防止出现负数；

        有一点要注意的是，着色器默认是不让修改`gl_PointSize`，所以需要利用`glEnable(GL_PROGRAM_POINT_SIZE)`修改着色器。

    3. 计算点在ModelView坐标中的坐标：

        这个坐标是为了在片元着色器计算Phong Shading要用到的，具体怎么用将在片元着色器中说明，在这里只给出计算的代码：`vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex`。

    4. 全部代码（为了简化读文件的过程，我直接写成字符串的形式）：

```
 const char* VertexCode = R"(
#version 120
uniform int radius;
varying vec4 vertex_in_modelview_space;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_PointSize = radius * (1 + gl_Position.z);
    gl_FrontColor = gl_Color;

    vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;
}
)";
```

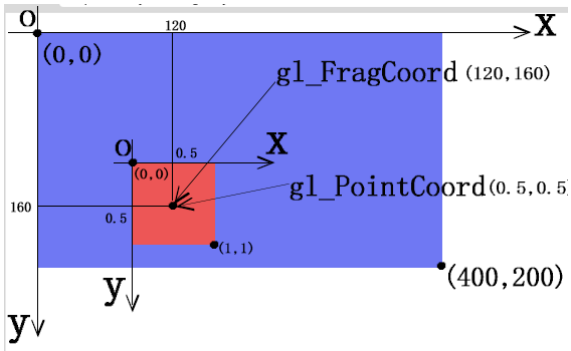
2. 片元着色器

    1. 通过丢弃部分片元来实现球的效果：

    首先介绍一下`gl_PointCoord`，它是渲染点片元坐标。根据这篇CSDN文章介绍，每个点会被渲染成方形区域，而`gl_PointCoord`就是描述这个方形区域的坐标系，其中坐标系的原点在方形的左上角，而方形的中心的坐标显然就是$(0.5, 0.5)$，并且这个中心点代表的就是顶点着色器的传过来的点，所以可以通过计算点片元上的点到中心点的距离来舍弃一些点，从而形成一个圆。具体代码如下：

```
float dist = distance(gl_PointCoord, vec2(0.5, 0.5));
    if (dist > 0.5)
        discard;
```

    做到这里，我有些迷茫，作业要求是要画球，现在只能画圆，这是可是差了一个维度的大问题。甚至我在想是不是CSDN文章写错了，我当初的想法是点应该被渲染成是一个立方体，然后我们要把它削成一个球。后来我也查阅了许多资料，始终没有找到我心中想的方法，后来请教了同学，得到了答案。片元着色器的功能就是渲染2D的，不存在3D的处理，那么如何使圆看起来了球，这就是光照的作用了，通过使圆上各部分光照的不同使圆看上去像球。就像中学美术课上的素描作业，通过在圆上加阴影，使2D的画有了3D的感觉。



    2. 计算Phong Shading

    根据课件给出Phong Shading的方法：

    > - 计算每个顶点的法向量
    > - 在多边形内部对顶点法向量进行插值
    > - 根据Phong反射模型计算多边形内部fragment颜色

    但是在这里是对点片元进行渲染，而不是对一个多边形的片元进行渲染，所以我觉得这里的Phong Shading就变成了Phong反射了。

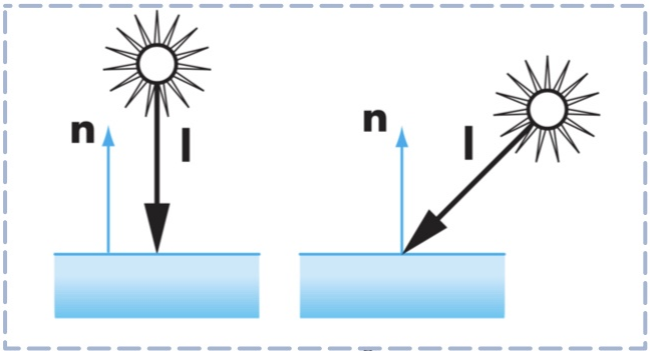
    Phong反射主要有三种反射成分，分别为环境光反射、漫反射、镜面反射。

- 环境光：场景中所有点处环境光$\mathbf{L}_a$一致。当环境光照射在物体表面时，一部分被表面吸收，另一部分被表面反射，反射部分强度由环境光反射系数k_a决定，即环境光的发射反射强度为$\mathbf{I}_a = k_a\mathbf{L}_a$。
- 漫反射光：$\mathbf{I}_d = k_d\mathbf{L}_d \cdot cos\theta$，其中$\theta$为照射点处法向量$\mathbf{n}$与光照角度$\mathbf{l}$的夹角，注意这里的两个向量都需要归一化。照射点的法向量求法为

  ```
  vec2 n_xy = gl_PointCoord - vec2(0.5, 0.5);
  vec3 n = normalize(vec3(n_xy, sqrt(0.5 * 0.5 - dot(n_xy, n_xy))));
    // 法向量
  ```

  `n_xy`就是圆上的点在`gl_PointCoord`上的坐标，而`sqrt(0.5 * 0.5 - dot(n_xy, n_xy))`就是该点在z轴的坐标，这样赋予了2维点在z轴上的坐标后，一个平面的圆就变成了半球，也就是我们看一个球的前半部分。知道了点在z轴上的坐标，那么该点的法向量就很简单了，就是点的增广坐标减去原点的增广坐标，即`vec3(n_xy, sqrt(0.5 * 0.5 - dot(n_xy, n_xy)))`。

  至于光照$\mathbf{l}$的球法，就是用光源的坐标减去球心的坐标，因为光源的坐标是在ModelView坐标中，所以我们也要用球心在ModelView中的坐标，这个坐标就是在顶点着色器中求得的`vertex_in_modelview_space`。

- 镜面反射：$\mathbf{I}_s = k_s\mathbf{L}_s \cdot max((\mathbf{r}\cdot\mathbf{v})^\alpha, 0)$，其中$\mathbf{r}$为反射方向，$\mathbf{v}$为观察方向，$\alpha$为高光系数。

  反射方向$\mathbf{r}$可以利用着色器的内建函数`reflect`函数，它的两个参数是光的入射方向和该点的法向量，不过这里的入射向量的求法应该是球心坐标减去光源坐标。观察方向的求法是球心坐标减去相机位置，但是我找了很久都没找到着色器中哪个**attribute**是用来描述相机位置，所以只能建一个**uniform**变量来传参。

  注意，当$\mathbf{r}\cdot\mathbf{n}$为负时镜面反射不起作用。

  当然，也可以用Blinn-Phong反射模型，公式为$\mathbf{I}_s = k_s\mathbf{L}_s \cdot max((\mathbf{n}\cdot\mathbf{h})^\alpha, 0)$，其中$\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{|\mathbf{l}+\mathbf{v}|}$。

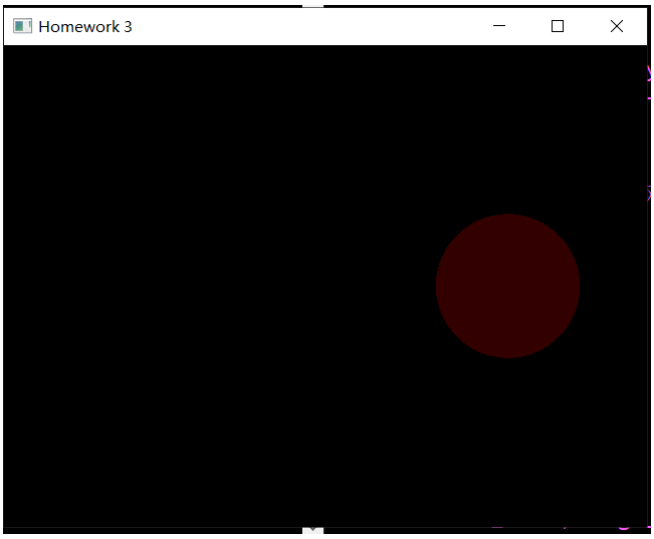
3. 全部代码（为了简化读文件的过程，我直接写成字符串的形式）：

```
    const char* FragmentCode = R"(
#version 120
uniform vec3 camera_pos;
varying vec4 vertex_in_modelview_space;
void main()
{
    float dist = distance(gl_PointCoord, vec2(0.5, 0.5));
    if (dist > 0.5)
        discard;

 vec2 n_xy = gl_PointCoord - vec2(0.5, 0.5);
```

```
    vec3 n = normalize(vec3(n_xy, sqrt(0.5 * 0.5 - dot(n_xy, n_xy))));
  // 法向量
    vec3 l = normalize(vec3(gl_LightSource[0].position -
vertex_in_modelview_space));   // 入射方向

    vec3 r = reflect(-l, n);  // 反射方向
    vec3 v = camera_pos - vertex_in_modelview_space.xyz;     // 观察方向
    vec3 h = normalize(v + l);

    vec4 ambient = gl_LightSource[0].ambient;
    vec4 diffuse = gl_LightSource[0].diffuse;
    vec4 specular = gl_LightSource[0].specular;

    float diffuse_term = clamp(dot(n, l), 0.0, 1.0);
    float specular_term = max(pow(dot(n, h), 500), 0);

    if (dot(n, l) < 0)
        gl_FragColor = (ambient + diffuse * diffuse_term) * gl_Color;
    else
        gl_FragColor = (ambient + 0.7 * diffuse * diffuse_term +
specular * specular_term) * gl_Color;
}
)";
```

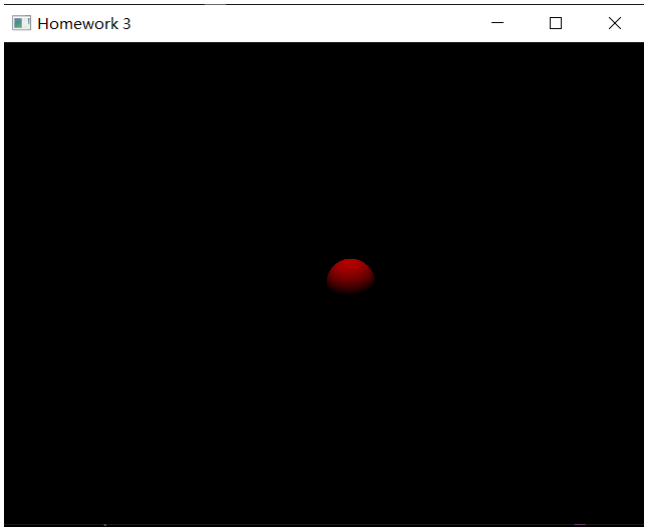
三. 过程截图

1. 只有环境光反射的效果



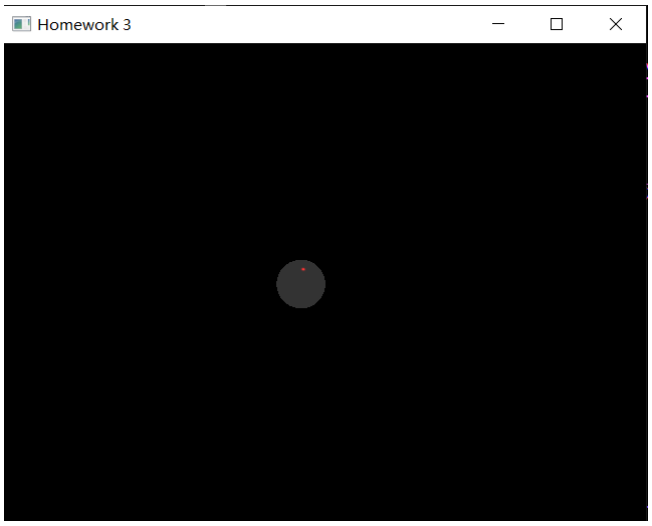
只有环境光的情况下，一个圆是看不出球的感觉的，并且只有环境光的情况下，场景挺暗的。

2. 只有漫反射反射的效果



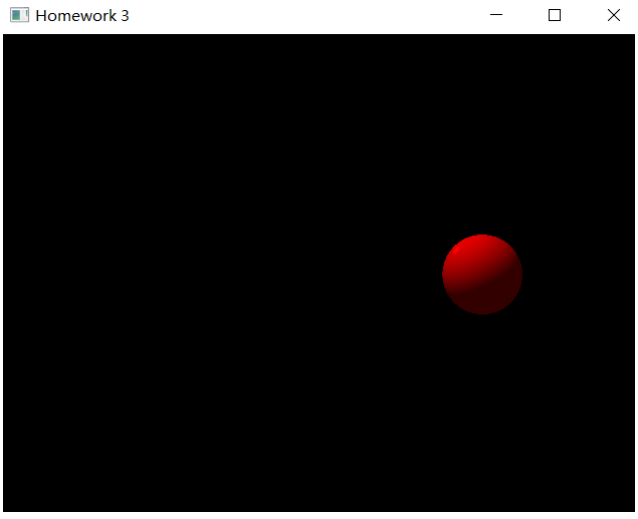
只有漫反射的情况下，一个圆已经有球的感觉了。

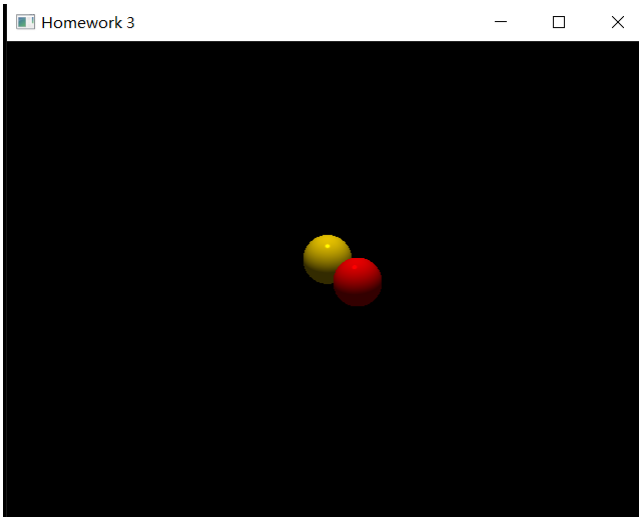
3. 环境光反射加镜面反射的效果（没有环境光的话就只会有一个亮点）



镜面反射就是产生一个高亮点。

4. 三种反射融合后的效果




Homework 3