中山大學
SUN YAT-SEN UNIVERSITY

《计算机图形学》

（作业三）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	17 计算机科学与技术
姓名学号	：	17341067 江金昱

方案

利用 Opengl 中的点精灵配合点的绘制方式来模拟球体粒子。
通过 shader 达到对每个粒子的控制，其中顶点着色器用来模拟粒子大小以及粒子运动，片元着色器用来控制粒子的形状以及粒子的色彩表现。
每帧绘制采用绑定顶点数组对象（VAO）的方式，来实现对顶点数据的便捷访问与绘制。

功能解析

点精灵

想要自定义粒子，需要使用点精灵，并开启相关设置，之后才能在 shader 中变更点精灵的样貌，具体配置如下：

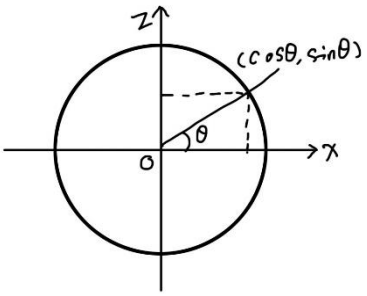
```
glEnable(GL_POINT_SPRITE_ARB);  //开启点精灵
glTexEnvi(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE);  //设置纹理替换模式，为没有纹理单元指定点精灵的纹理坐标替换模式
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_NV); //开启点在顶点着色器中设置大小
```

Shader 使用

使用 glCreateShader 创建 shader
glShaderSource 载入 shader 代码
glCompileShader 编译 Shader
glGetShaderiv 可知道编译是否成功
glGetShaderInfoLog 可得到编译输出的日志
glAttachShader 得到 shader 的控制权
glLinkProgram 链接 shader
glGetProgramiv 可知道链接是否成功
glGetProgramInfoLog 可得到链接输出的日志
当编译成功之后调用 glUseProgram 即可使用 shader
通过 glGetUniformLocation 可得到参数位置，并使用 glUniformXX 系列函数来配置参数

顶点着色器

球体粒子运动



如上图在已知角度 θ 的情况下，假设轨迹为半径为 1 的单位圆，可以得出圆上点（x,y）的坐标为（cos θ ,sin θ ）。

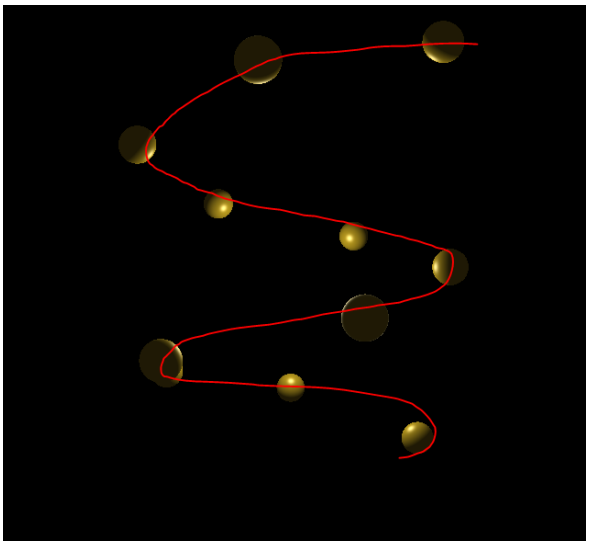
参照这个坐标公式，以粒子原始坐标为中心 O，环绕半径设定为 RoundDis，则粒子的位置偏移量为 RoundDis*（cos θ ,sin θ ）。此时只要不断改变 θ 的值，粒子就能绕圆进行运动，因此以时间作为 θ 的变化因子，θ 就能随时间变化而变化。因为同时绘制的点有多个，为了让粒子间的运动产生区别，于是将粒子的高度 y 也引入到 θ 作为变化因子。

具体代码如下：

```
uniform float radius;    //半径
uniform float time;      //时间
uniform float RoundDis;      //环绕中心的半径

varying vec4 v;          //点在视图空间的坐标

void main()
{
    float angle = time / 30 + gl_Vertex.y;  //按点的不同高度得到角度随时间的变化
    float Xoffset = cos(angle) * RoundDis;     //X方向的偏移量
    float Zoffset = sin(angle) * RoundDis;     //Z方向的偏移量

    vec4 eyePos = gl_ModelViewMatrix * gl_Vertex + vec4(Xoffset, 0, Zoffset, 1); //点坐标在视图空间的坐标

    //利用点坐标在视图的深度和预设的半径大小组合成新的坐标并转化到投影空间
    vec4 PointPos = gl_ProjectionMatrix * vec4(radius, radius, eyePos.z, eyePos.w);

    gl_PointSize = 800 * PointPos.x / PointPos.w; //设置点大小

    v = eyePos;    //传递给片元着色器

    gl_Position = gl_ProjectionMatrix * eyePos;      //设置点坐标
}
```

最终效果：

粒子以螺旋形的方式在空间中排布旋转。

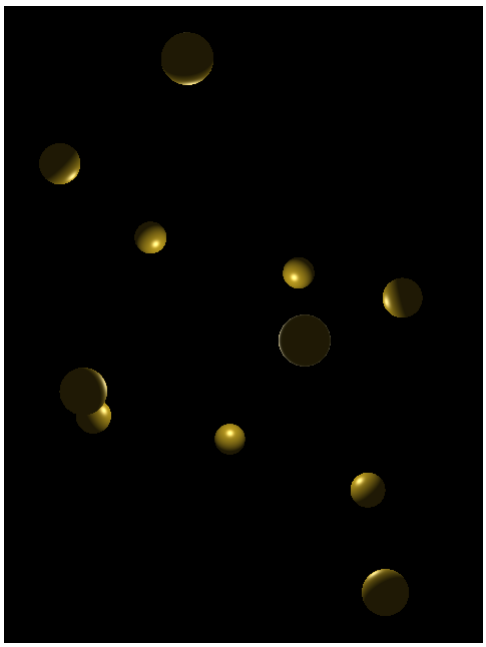


球体粒子大小控制

利用粒子在视图空间内位置的深度（Z 方向）作为大小的参照因子，并假设粒子位于屏幕正中心，将粒子半径 radius 映射到 X 轴 Y 轴，并与实际坐标结合形成一个新的坐标（radius, radius， z，eyePos.w），通过将该坐标变换到投影空间，访问最新的 X 或 Y 的映射数据即可得到最终的粒子半径大小，具体代码如下：

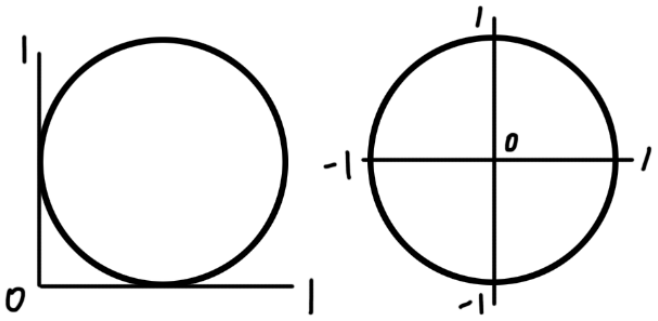
```
vec4 eyePos = gl_ModelViewMatrix * gl_Vertex + vec4(Xoffset, 0, Zoffset, 1); //点坐标在视图空间的坐标

//利用点坐标在视图的深度和预设的半径大小组合成新的坐标并转化到投影空间
vec4 PointPos = gl_ProjectionMatrix * vec4(radius, radius, eyePos.z, eyePos.w);

gl_PointSize = 800 * PointPos.x / PointPos.w; //设置点大小
```

具体效果，近大远小：



片元着色器

法线与形状



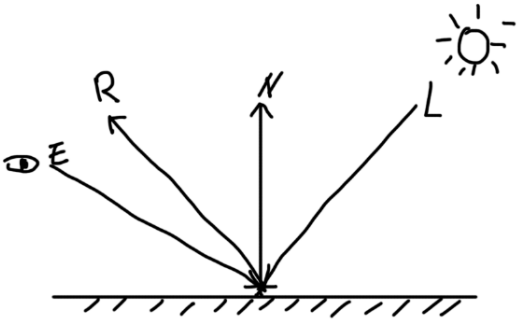
如图因为一开始配置过粒子的纹理，所以此时所有片元的纹理坐标值大小在0-1之间。这时候相当于球心的中心纹理坐标为（0.5， 0.5）。为了更方便运算，将球心的纹理坐标移动到（0,0）位置，并将纹理坐标映射到（-1，1）这个范围，之后只需要通过纹理坐标就可以判断该片元是否落在圆形范围内，不落在该范围的片元剔除即可。

因为圆表面点的法线必定通过圆心，而上述操作已经将圆心移动到了（0,0）点，因此可以利用单位向量长度为1，在已知X,Y坐标的情况下求得Z的坐标，法向量就能求出来了。

具体代码如下：

```
vec3 N;    //定义法线
N.xy = gl_PointCoord*2 - vec2(1.0f);      //把纹理坐标从(0,1)范围映射到(1,1)
float r2 = dot(N.xy, N.xy); //计算在x,y纹理平面上的点半径

//超出单位半径剔除片元
if(r2>1.0)
    discard;

//利用单位向量求z维度的值
N.z = sqrt(1-r2);
//从纹理坐标转化到真实空间的朝向
N.y = -N.y;
```

Phong Shading



Phong Shading 由环境光，漫反射以及高光组成。

通过已知灯和点的坐标可得灯光方向 L，点积灯光和法线方向可以得到物体表面的明暗关系即漫反射，为了防止这个结果为负数，这里使用了 max 函数。

之后通过反射 L 得到出射光的方向 R，点积 R 和点到眼睛的方向-E，可以得到在该观察视角下的表面高光亮度。

因为物体背面的法线 N 与灯光 L 点积之后的结果通过 max 函数后肯定会为 0 也就是黑色，为了让颜色不那么黑，引入了环境光，将其与漫反射相乘得到背后的颜色。

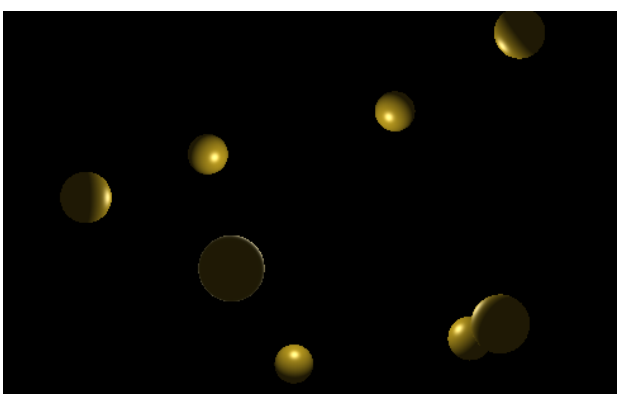
需要向片元着色器传递的信息：

```
uniform vec4 ambient;    //环境光颜色
uniform vec4 diffuse;    //小球漫反射颜色
uniform vec4 specL;      //高光颜色
uniform float specPow;   //高光渐变曲率
uniform vec4 ligthv;     //灯在视图空间的坐标
uniform vec4 ligthDiffuse;    //灯光颜色

varying vec4 v;          //点在视图空间的坐标
```

具体代码如下：

```
vec3 L = normalize(v.xyz - ligthv.xyz);      //灯光的方向
vec3 E = normalize(v.xyz);  // 得到眼睛看向点的朝向
vec3 R = normalize(reflect(L,N));  //得到灯光经过该点反射后的方向

vec4 diffuseColor = ligthDiffuse * diffuse * max(dot(N,-L), 0.0);  //得到明暗变化
vec4 spec = ligthDiffuse * specL *pow(max(dot(R,-E),0.0), specPow); //得到高光

gl_FragColor = ambient * diffuse + diffuseColor + spec;      //得到最终颜色
gl_FragDepth = r + v.z;         //深度排序
```

最终效果：



VBO/VAO

顶点缓冲对象(Vertex Buffer Objects, VBO)，它会在 GPU 内存（通常被称为显存）中储存大量顶点，这使得我们可以一次性的发送一大批数据到显卡上，而不是每个顶点发送一次。

顶点数组对象(Vertex Array Object, VAO)，用于存储顶点属性调用。当配置好顶点属性指针时，你只需要将那些调用执行一次，之后再绘制物体的时候只需要绑定相应的 VAO 就行了，这使在不同顶点数据和属性配置之间切换变得非常容易。

具体代码如下：

```
//生成并绑定相应功能指针
glGenVertexArrays(1, &vaoId);
glGenBuffers(1, &vboId);

//绑定数组指针
glBindVertexArray(vaoId);
glBindBuffer(GL_ARRAY_BUFFER, vboId);
//写入顶点数据
glBufferData(GL_ARRAY_BUFFER, sizeof (vex), vex, GL_STATIC_DRAW);

//设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

//解绑VAO
glBindVertexArray(0);
```

之后在渲染循环中就能很方便的使用：

```
glBindVertexArray(vaoId);
glDrawArrays(GL_POINTS, 0, 12);
glBindVertexArray(0);
```

总结

1. 困难：在实现 Phong Shading 的时候缺少了粒子表面的法线这个关键变量，导致后续的光照计算无法进行下去。
   解决方法：通过开启点精灵的纹理替换模式，使用纹理坐标来模拟出表面法线的方向。
2. 困难：通过纹理坐标直接转化出来的法线方向角度整体有偏转
   解决方法：后来发现是因为纹理坐标系的方向和 3 维坐标系的 Y 轴方向是相反的，所以只需要乘以-1 翻转即可。
3. 困难：多个粒子间深度无法确定，导致渲染顺序出错
   解决方法：片元着色器中的片元深度采用粒子在空间坐标中的深度并累加上像素距离球体中心的距离来解决。

通过这次的作业，了解到了粒子的实现思路，并且学习到了 shader 的用法，给了自己很大的灵感在之后的图像处理上，未来可以利用 shader 慢慢实现很多炫酷的效果。