计算机图形学

（Computer Graphics）

实验题目：OpenGL 中的光照功能

目录：

实验报告：

1. 实验内容描述，即使用的光照描述和特点

本次实验使用三个点光源，分别是红色、绿色、白色，红光位于（0.0,0.0,3.0），绿光（-4.0,0.0,-1.5），白光位于（0.0,6.0,3.0），每个光源的环境光系数为 0.2，漫射光系数和镜面反射光系数都为 0.8。用于计算点光源的光强衰减的参数：常数衰减系数为 0.2，线性衰减系数为 0.1，指数衰减系数为 0.1。然后，为了更好的看到镜面反射的效果，shininess 系数（K 值）默认为 1。

2. 实验功能算法描述，即光照是如何建立、如何定义，描述其原理

每个像素点的颜色值的计算

$$\text{Color} = \text{Texture} * \text{material} * (\text{AmbientLight} + \text{DiffuseLight} + \text{SpecularLight})$$

公式 1 每个像素的着色计算

点光源的距离使用另一个公式去模拟其衰减（常数衰减系数，线性衰减系数，指数衰减系数），这样可以避免当两个物体的距离过近使得距离无限小的情况出现

$$LightDistance^2 = Attenuation_{constant} + Attenuation_{liner} * LightDistance + Attenuation_{exp} + LightDistance^2$$

公式 2 光源与着色点距离计算

(1) 环境光(Ambient Light)

环境光定义：由于没有使用全局的光照算法，因此，使用粗略的环境光来模拟环境中各个物件反射所造成的每个物体都受其影响的光。这样可以避免，当一个物体的某个面并没有受到光源的影响（漫射光、镜面光均为 0）时，是完全黑色的情况出现

环境光的计算较为简单，只于光源的颜色和光源的环境光系数有关。

$$\text{AmbientLight} = \sum_{light} \text{AmbientIntensity} * \text{Color}$$

公式 3 环境光光强计算公式

(2) 漫射光(Diffuse Light)

漫射光定义：一个光源发出的光照射到一个物体上的时候会发生漫反射，然后使得物体在各个方向上看都会有一定的亮度。

漫射光的计算和光源的漫射光系数，光的入射方向(计算时取反方向)与物体表面法线的夹角 θ 有关

$$\text{DiffuseLight} = \sum_{light} \text{DiffuseIntensity} * \text{Color} * (\cos\theta)$$

$$= \sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{LightDirection}} \cdot \overrightarrow{\text{Normal}}\right)$$

公式 4 平行光源的漫反射计算公式

$$\text{DiffuseLight} = \frac{\sum_{light} \text{DiffuseIntensity} * \text{Color} * (\cos\theta)}{LightDistance^2}$$

$$= \frac{\sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{LightDirection}} \cdot \overrightarrow{\text{Normal}}\right)}{Attenuation_{constant} + Attenuation_{liner} * LightDistance + Attenuation_{exp} + LightDistance^2}$$

公式 5 点光源的漫射光计算公式

(3) 镜面反射光(Specular Light)

镜面反射光定义：一个光源发出的光在物体上发生镜面反射，并且视线与反射光成一定角度是可以看到这个光。

镜面反射光的计算和光源的镜面反射系数，光的反射方向以及眼睛方向的夹角 θ 有关

$$\text{SpecularLight} = \sum_{light} \text{SpecularIntensity} * \text{Color} * (\cos\theta)^K$$

$$= \sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{EyeDirection}} \cdot \overrightarrow{\text{ReflectDirection}}\right)^K$$

$$= \sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{EyeDirection}} \cdot \overrightarrow{\text{reflect}(\overrightarrow{\text{LightDirection}}, \overrightarrow{\text{Normal}})}\right)^K$$

公式 6 平行光源的镜面反射光计算公式

$$\text{SpecularLight} = \frac{\sum_{light} \text{SpecularIntensity} * \text{Color} * (\cos\theta)^K}{LightDistance^2}$$

$$= \frac{\sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{EyeDirection}} \cdot \overrightarrow{\text{ReflectDirection}}\right)^K}{LightDistance^2}$$

$$= \frac{\sum_{light} \text{DiffuseIntensity} * \text{Color} * \left(\overrightarrow{\text{EyeDirection}} \cdot \overrightarrow{\text{reflect}(\overrightarrow{\text{LightDirection}}, \overrightarrow{\text{Normal}})}\right)^K}{Attenuation_{constant} + Attenuation_{liner} * LightDistance + Attenuation_{exp} + LightDistance^2}$$

公式 7 点光源的镜面反射光计算公式

3. 实验 shader 程序描述，即 vertex shader 和 fragment shader 的程序代码及说明

(1) vertex shader

```
#version 330

layout(location = 0) in vec3 Position;  //传入的顶点坐标
layout(location = 1) in vec2 TexturePosition;   //传入的纹理坐标
layout(location = 2) in vec3 Normal; //传入的顶点法向量

//传入的变换到眼睛坐标系的变换矩阵
uniform mat4 gWVP;

//光照相关
uniform mat4 gWorld; //传入的变换到世界坐标系的变换矩阵
uniform vec3 gEye; //传入的眼睛的世界坐标系坐标

out vec2 TextureAfterTrans; //输出的纹理坐标
//out vec4 Color;

//光线相关
out vec3 NormalInWorld;
out vec3 PointPosition;
out vec3 EyePosition;

void main()
{
    //顶点位置
    gl_Position = gWVP * vec4(Position, 1.0);

    //光照相关
    PointPosition = (gWorld*vec4(Position, 0.0)).xyz;
    NormalInWorld = (gWorld*vec4(Normal, 0.0)).xyz;
    EyePosition = (vec4(gEye, 0.0)).xyz;

    TextureAfterTrans = TexturePosition;
    //Color = vec4(clamp(Position, 0.0, 1.0), 1.0);
}
```

图 1 vertex shader 程序代码

这次实验中的 vertex shader 接收的顶点属性多了一个，顶点的法向量（Normal），用来进行光照相关的计算，因此将它连同顶点坐标和纹理坐标一同传入 Fragment Shader 中。然后，由于镜面光光照计算的时候需要用到眼睛在世界坐标系的位置（相机的位置），因此通过 uniform 变量将其传入（gEye）。

(2) fragment shader

fragment shader 程序分为以下几个部分

(i) 光源结构体

```
//平行光源
struct DirectionalLight
{
    vec3 Color;//光源颜色
    float AmbientIntensity; //环境光系数
    float DiffuseIntensity;//漫射光光强
    float SpecularIntensity;//镜面反射光光强
    vec3 Direction;//光照方向
};
```

图 2 平行光光源定义

平行光光源由光源颜色、环境光系数、漫射光系数、镜面反射光系数以及光照方向组成。

```
//点光源
struct PointLight
{
    vec3 Color;
    float DiffuseIntensity; //漫反射系数
    float AmbientIntensity; //环境光系数
    float SpecularIntensity; //反射光系数
    float Attenuation_constant; //常数衰减系数
    float Attenuation_liner; //线性衰减系数
    float Attenuation_exp;  //指数衰减系数
    vec3 position; //点光源位置
};
```

图 3 点光源定义

点光源由除了平行光源所拥有的信息外，还需要记录点光源位置，以及各个衰减系数用于计算衰减，当然，由于是点光源，因此没有特定的光照方向。

(ii) 光照计算函数

较为复杂的光照计算使用了光照函数来封装过程，计算公式来自于上面的推导。

```
//漫射光最后的效果计算(光源是平行光源)
vec4 CaculateDiffuseColor_directionLight(DirectionalLight light, vec3 PointPosition, vec3 NormalInWorld)
{
    //点乘正在处理的点的法线和光照方向得到漫射光的具体光强
    float LightDirectionFactor = dot(normalize(NormalInWorld), normalize(light.Direction));
    //判定得到的具体光强系数，如果小于0，则说明光照是从物体背面传来的，因此将光照系数赋为0，如果大于一(理论上不会，则将其限制为1)
    LightDirectionFactor = LightDirectionFactor > 0.0 ? LightDirectionFactor : 0.0;
    LightDirectionFactor = LightDirectionFactor < 1.0 ? LightDirectionFactor : 1.0;
    //最后的光的效果是  (漫射光的颜色*漫射光的影响系数*漫射光的具体光强系数)
    return vec4(light.Color * light.DiffuseIntensity * LightDirectionFactor, 1.0f);
};
```

图 4 平行光源的漫射光计算函数

```
//漫射光最后的效果计算(光源是点光源)
vec4 CaculateDiffuseColor_pointLight(PointLight light, vec3 PointPosition, vec3 NormalInWorld)
{
    //入射光线向量（从光源指向当前物体上正在处理的点）(暂时不进行单位化，因为需要用来计算距离)
    vec3 LightDirection = PointPosition - light.position;
    //得到光源和物体上的点的距离
    float LightDistance = length(LightDirection);
    //点乘正在处理的点的法线和光照方向得到漫射光的具体光强（此时需要进行单位化，因为是用来计算夹角）
    float LightDirectionFactor =dot(normalize(NormalInWorld), normalize(-LightDirection));
    //判定得到的具体光强系数，如果小于0，则说明光照是从物体背面传来的，因此将光照系数赋为0，如果大于一(理论上不会，则将其限制为1)
    LightDirectionFactor = LightDirectionFactor>0.0?LightDirectionFactor:0.0;
    LightDirectionFactor = LightDirectionFactor<1.0?LightDirectionFactor:1.0;
    //最后的光的效果是  (漫射光的颜色*漫射光的影响系数*漫射光的具体光强系数) /  (点光源的常数衰减系数+光照距离*点光源的线性衰减系数+光照距离的平方*点光源的指数衰减系数)
    return vec4(light.Color * light.DiffuseIntensity * LightDirectionFactor, 1.0f)/
                    (light.Attenuation_constant + light.Attenuation_liner * LightDistance + light.Attenuation_exp * LightDistance * LightDistance);
    //return vec4(light.Color * light.DiffuseIntensity * LightDirectionFactor, 1.0f);
};
```

图 5 点光源的漫射光计算函数

```
//镜面光最后的影响计算(光源是点光源)
vec4 CaculateSpecularColor_pointLight(PointLight light, vec3 PointPosition, vec3 NormalInWorld,vec3 EyePosition ,float SpecularPower)
{
    //指向眼睛的方向向量
    vec3 VertexToEye= normalize(EyePosition - PointPosition);
    //入射光线向量（光源指向正在处理的点）
    vec3 LightDirection = PointPosition - light.position;
    //反射光线向量（使用内置的reflect函数计算，第一个参数为入射光线向量，第二个参数为法线向量）
    vec3 LightReflect = normalize(reflect(LightDirection, NormalInWorld));

    //光照距离
    float LightDistance = length(LightDirection);

    //镜面反射的具体光强系数
    float SpecularFactor = dot(VertexToEye, LightReflect);

    //如果最后的光强系数大于0，说明眼睛和反射光线在同一方向，则计算对一个的镜面反射光光强，如果小于等于零，说明眼睛不再反射光影响范围内，因此将反射光光强赋为0
    if (SpecularFactor > 0 && SpecularFactor>cos(10))
    {
        if (dot(NormalInWorld, VertexToEye) <= 0)
        {
            return vec4(0.0, 0.0, 0.0, 0.0);
        }
        else
        {
            SpecularFactor = pow(SpecularFactor, SpecularPower);
            return vec4(light.Color * light.SpecularIntensity * SpecularFactor, 1.0f) /
                (light.Attenuation_constant + light.Attenuation_liner * LightDistance + light.Attenuation_exp * LightDistance * LightDistance);
            //return vec4(light.Color * light.SpecularIntensity * SpecularFactor, 1.0f);
        }
    }
    else
    {
        return vec4(0.0,0.0,0.0,0.0);
    }
};
```

图 6 点光源的镜面反射光的计算函数

```
//镜面光最后的影响计算(光源是平行光源)
vec4 CaculateSpecularColor_directionLight(DirectionalLight light, vec3 PointPosition, vec3 NormalInWorld, vec3 EyePosition, float SpecularPower)
{
    //指向眼睛的方向向量
    vec3 VertexToEye = normalize(EyePosition - PointPosition);

    //反射光线向量（使用内置的reflect函数计算，第一个参数为入射光线向量，第二个参数为法线向量）
    vec3 LightReflect = normalize(reflect(light.Direction, NormalInWorld));


    //镜面反射的具体光强系数
    float SpecularFactor = dot(VertexToEye, LightReflect);

    //如果最后的光强系数大于0，说明眼睛和反射光线在同一方向，则计算对一个的镜面反射光光强，如果小于等于零，说明眼睛不再反射光影响范围内，因此将反射光光强赋为0
    if (SpecularFactor > 0)
    {
        SpecularFactor = pow(SpecularFactor, SpecularPower);
        return vec4(light.Color * light.SpecularIntensity * SpecularFactor, 1.0f);
    }
    else
    {
        return vec4(0.0, 0.0, 0.0, 0.0);
    }
};
```

图 7 平行光源的镜面反射光计算函数

(iii) 主函数部分

最后，在主函数中定义各个光源的参数

```
//红色点光源
PointLight RedPointLight;
RedPointLight.Color=vec3(0.8,0.0,0.0);
RedPointLight.AmbientIntensity=0.2;
RedPointLight.DiffuseIntensity=0.8;
RedPointLight.SpecularIntensity= 0.8;
RedPointLight.Attenuation_constant= 0.2;
RedPointLight.Attenuation_liner= 0.1;
RedPointLight.Attenuation_exp= 0.1;
RedPointLight.position=vec3(0.0,0.0,3.0);
```

图 8 光源定义

使用各个光照计算函数进行计算

```
//计算环境光光强
vec4 AmbientColor = vec4(RedPointLight.Color * RedPointLight.AmbientIntensity, 1.0f)+
                    vec4(BluePointLight.Color * BluePointLight.AmbientIntensity, 1.0f)+
                    vec4(GreenPointLight.Color * GreenPointLight.AmbientIntensity, 1.0f);

//计算漫射光强
vec4 DiffuseColor1 = CaculateDiffuseColor_pointLight(RedPointLight, PointPosition, NormalInWorld);
vec4 DiffuseColor2 = CaculateDiffuseColor_pointLight(GreenPointLight, PointPosition, NormalInWorld);
vec4 DiffuseColor3 = CaculateDiffuseColor_pointLight(BluePointLight, PointPosition, NormalInWorld);

//计算镜面反射光
vec4 SpecularColor1 = CaculateSpecularColor_pointLight(RedPointLight, PointPosition, NormalInWorld,EyePosition,1.0);
vec4 SpecularColor2 = CaculateSpecularColor_pointLight(GreenPointLight, PointPosition, NormalInWorld,EyePosition,1.0);
vec4 SpecularColor3 = CaculateSpecularColor_pointLight(BluePointLight, PointPosition, NormalInWorld,EyePosition,1.0);
```

图 9 光照计算

最后进行着色

```
FragColor = texture2D(gSampler, TextureAfterTrans.st) * MatarialColor *(
        AmbientColor +
        DiffuseColor1 + DiffuseColor2 + DiffuseColor3 +
        SpecularColor1 + SpecularColor2 + SpecularColor3);
```

图 10 最后着色

4. 实验结果，要贴实验结果图

图 11 实验结果视图 1

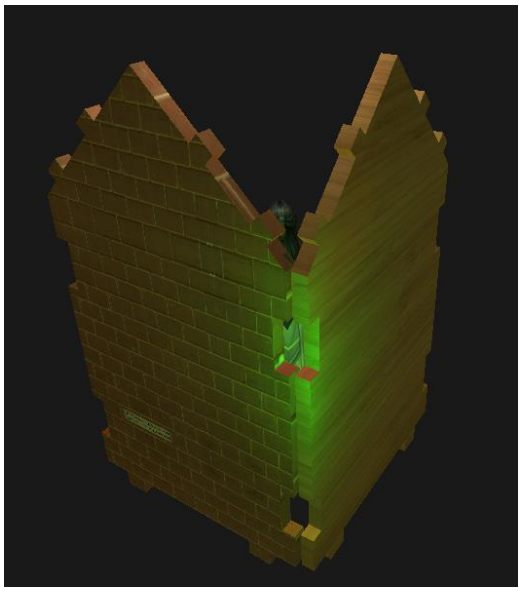

图 12 漫反射效果（从绿色光源位置看，因此只有漫射光以及环境光）

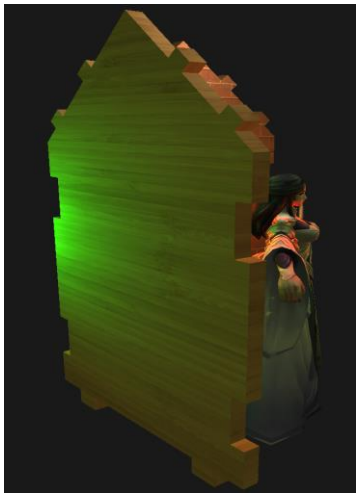

图 13 点光源漫反射、镜面反射光效果（在反射方向观看，因此有明显的镜面反射效果）

5. 心得体会

在这次实验中，我实现了环境光、漫射光、镜面反射光，三种光照，把光照计算的理论用于实践。然后，更重要的是，通过将不同的模型引入并给他们附上不同的纹理，对 OpenGL 作为自动机的本质有了更深的理解，也将之前遇到的很多问题解决了，受益匪浅！