计算机图形学

Lighting

Assignment 4

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 作业任务与要求

(1) 在 Shader 类中的 fragment 函数中实现漫反射光照的计算逻辑；

(2) 在 Shader 类中的 fragment 函数中实现 Phong 的镜面光照计算逻辑；

(3) 在 Shader 类中的 fragment 函数中实现 Blinn-Phong 的镜面光照计算逻辑；

(4) 在 main 函数中实现 eye 绕着 y 旋转的动画；

(5)（选做）在 OpenGL 中实现传统的光照模型；

2 实验过程与结果

2.1 Task1：漫反射光照的计算逻辑实现

2.1.1 计算逻辑

基于图1，漫反射的计算公式如1所示：

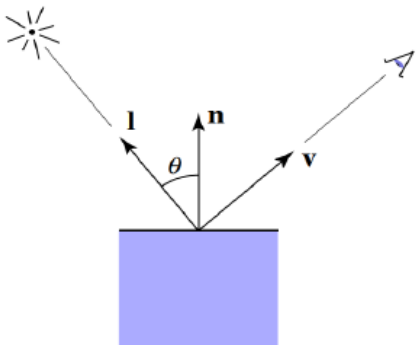

图 1: 漫反射模型

$$L_d = k_d \frac{I}{r^2} \max(0, \mathbf{n} \cdot \mathbf{l}) \tag{1}$$

其中 L_d 是人眼接收到的光照，k_d 是漫反射系数，I 是光的颜色（辐射率）。由于光会随着与光源的距离增加而衰减，故距离 r 用于调整光照。这样的话 $\frac{I}{r^2}$ 就是物体表面接收到的光。$\mathbf{n}$ 是物体表面的法向量，$\mathbf{l}$ 是光照方向向量，两者都是单位向量，点乘结果就是 $\sin\theta$。为了防止出现负数的情况（此时光照从背面射入，无意义），故需要 max() 函数。

对于本次实现，我们使用的是平行光、不考虑距离的衰减，故可以不考虑 r。

2.1.2 代码实现

实现代码如下所示：

```
virtual bool fragment(unsigned int index, Vec3f pos, Vec3f bar, TGAColor &color) {
    ......
    // use these vectors to finish your assignment.
```

```
    // the normal vector of this fragment.
    Vec3f normal = (B*models[index]->normal(uv)).normalize();
    // the lighting direction.
    Vec3f lightDir = light_dir;
    // the position of this fragment in world space.
    Vec3f fragPos = pos;
    // the viewing direction from eye to this fragment.
    Vec3f viewDir = (eye - fragPos).normalize();
    // the texture color of this fragment (sample from texture using uv coordinate)
    TGAColor fragColor = models[index]->diffuse(uv);

    static float ka = 0.0f;
    static float kd = 0.8f;
    static float ks = 0.4f;

    float diff = 1.0f;
    // TODO 1:
    // calculate the diffuse coefficient using Phong model, and then save it to diff.
    {
        diff=kd*MAX(0,normal*lightDir.normalize());
    }
    TGAColor diffColor = (fragColor*diff);
    ......
}
```

2.1.3 渲染结果

漫反射光照的实现前后如图2和3所示。

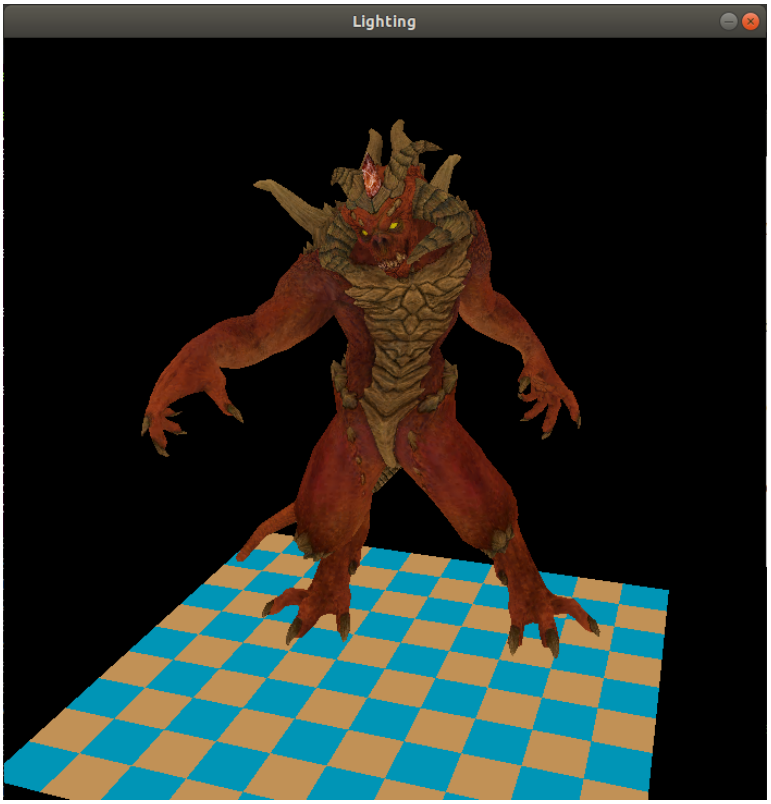

图 2: 漫反射实现前

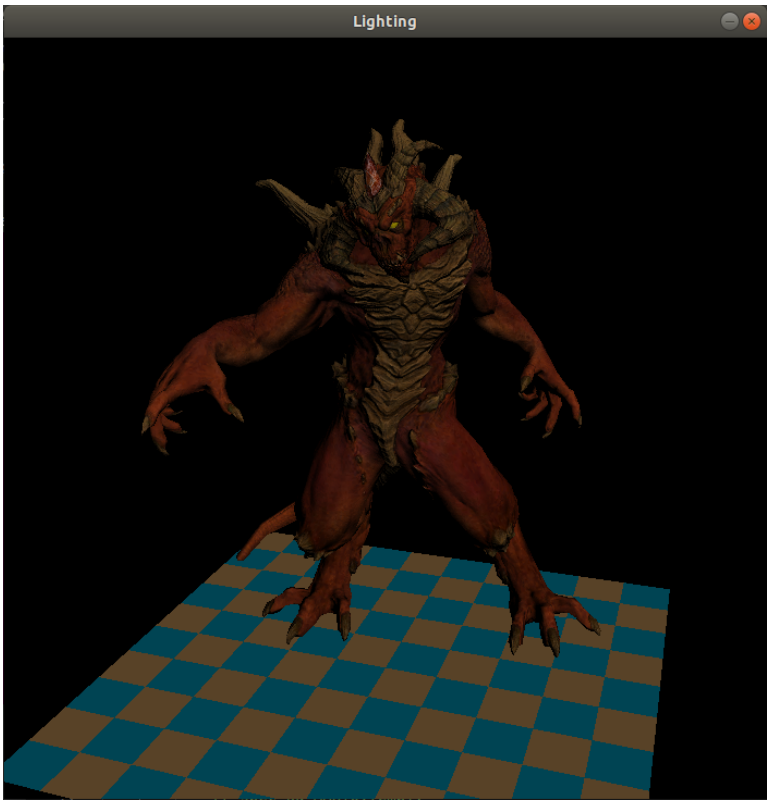

图 3: 漫反射实现后

可见实现漫反射后，模型的亮度下降严重，模型表面的颜色不再是原始颜色。

2.2 Task2：Phong 镜面光照计算逻辑实现

2.2.1 计算逻辑

基于图4，phong 镜面光照如式2所示：

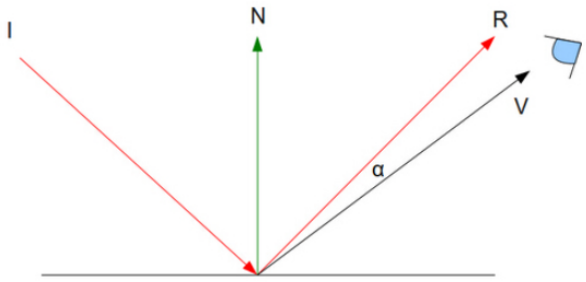

图 4: Phong 模型

$$L = k_s I_{light} \max(0, \mathbf{V} \cdot \mathbf{R})^p \tag{2}$$

k_s 是镜面反射系数，I_light 是入射的光照强度。p 为镜面反射的高光系数，其值越大时表面越接近镜面、高光面积越小。$\mathbf{R}$ 是反射向量，其和 $\mathbf{I}$ 与法向量 N 的夹角相同，均为单位向量。通过几何关系，我们可以用以下公式计算：

$$\mathbf{R} = \mathbf{I} - 2(\mathbf{I} \cdot \mathbf{N})\mathbf{N} \tag{3}$$

2.2.2 代码实现

实现代码如下所示。需要注意的是，代码中光线向量的方向与图4中的相反，故计算 R 时方向取反。

```
virtual bool fragment(unsigned int index, Vec3f pos, Vec3f bar, TGAColor &color) {
    ......
    // use these vectors to finish your assignment.
    // the normal vector of this fragment.
    Vec3f normal = (B*models[index]->normal(uv)).normalize();
    // the lighting direction.
    Vec3f lightDir = light_dir;
    // the position of this fragment in world space.
    Vec3f fragPos = pos;
    // the viewing direction from eye to this fragment.
    Vec3f viewDir = (eye - fragPos).normalize();
    // the texture color of this fragment (sample from texture using uv coordinate)
    TGAColor fragColor = models[index]->diffuse(uv);
    static float ka = 0.0f;
    static float kd = 0.8f;
    static float ks = 0.4f;
    float diff = 1.0f;
    ......
    // TODO 2:
    // calculate the specular coefficient using Phong model, and then save it to spec.
    {
```

```
22          int p=16;
23          // phong
24          Vec3f R=normal*(2*(normal*lightDir.normalize()))-lightDir.normalize();
25          float temp=pow(R*viewDir.normalize(),p);
26          spec=ks*MAX(0,temp);
27      }
28      TGAColor specColor = TGAColor(255,255,255)*spec;
29      // the final color of this fragment taking lighting into account.
30      color = diffColor + specColor;
31      return false;
32  }
```

2.2.3 渲染结果

渲染图像如下图所示，其中图5为 $p=4$ 的结果，图6为 $p=16$ 的结果，图7为 $p=64$ 的结果。

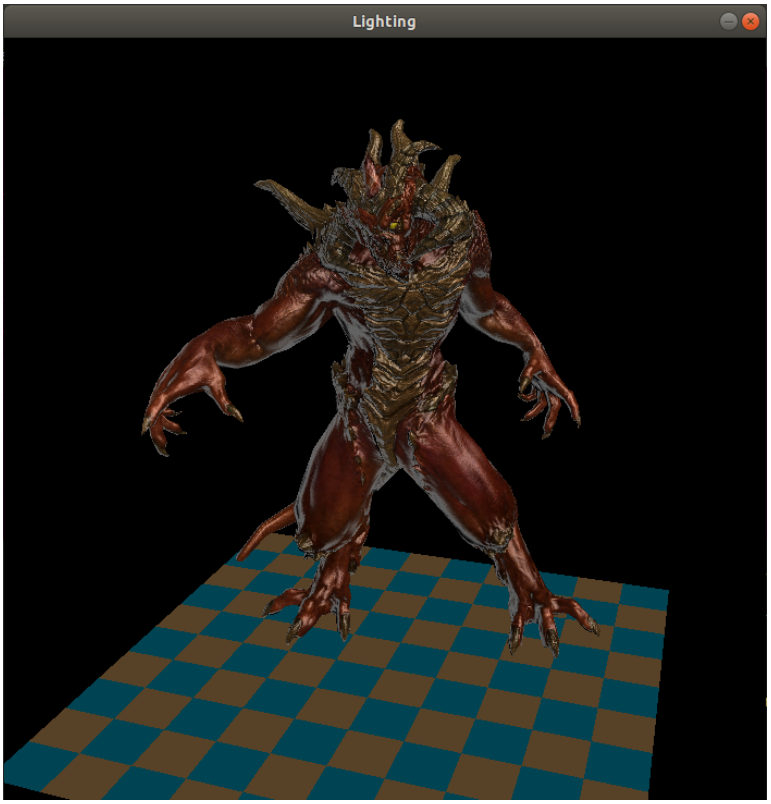

图 5: phong 模型 $p=4$ 渲染图

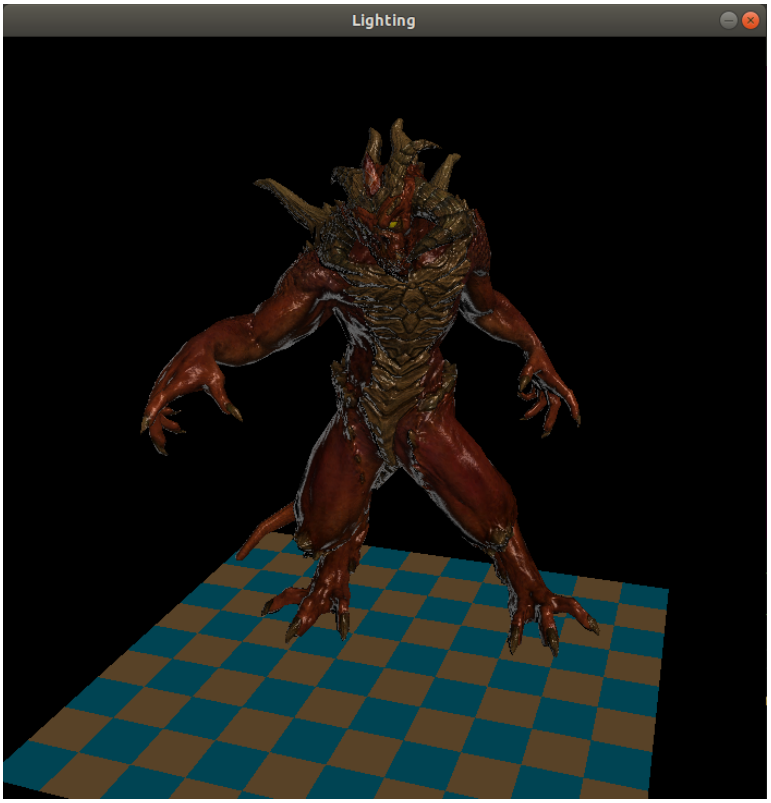

图 6: phong 模型 $p = 16$ 渲染图

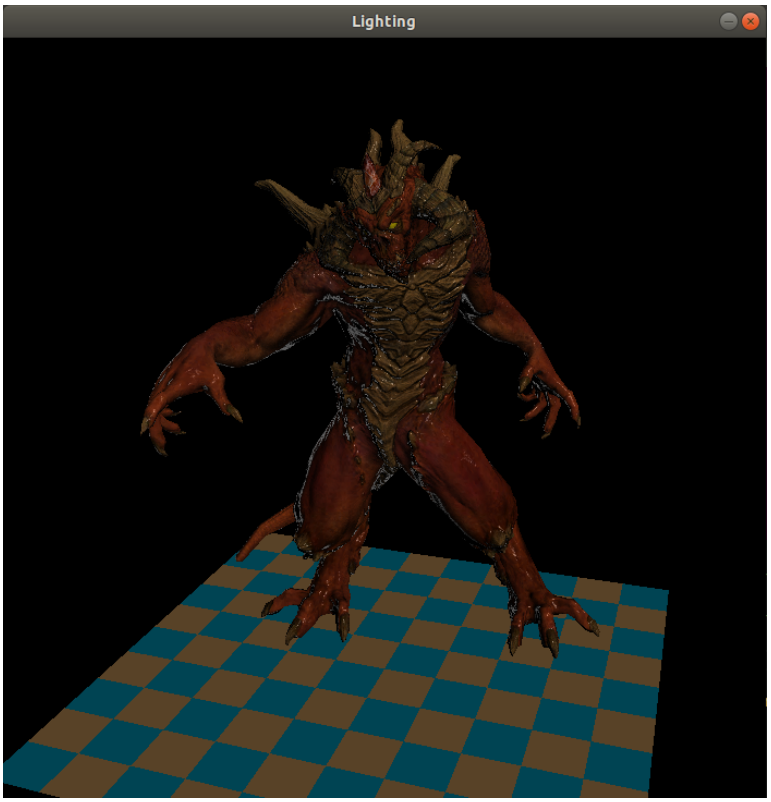

图 7: phong 模型 $p = 64$ 渲染图

可见，相比于只有漫反射的模型，使用 phong 镜面反射的模型多了些高光。随着 p 的逐渐增大，高光范围越来越小、模型总体越来越暗。当 $p = 64$ 时只有部分地方存在高光。

2.3 Task3：Blinn-Phong 镜面光照计算逻辑实现

2.3.1 计算逻辑

相比于 Phong 模型需要计算反射向量，Blinn-Phong 模型只需要计算半程向量，计算耗时大大减少。

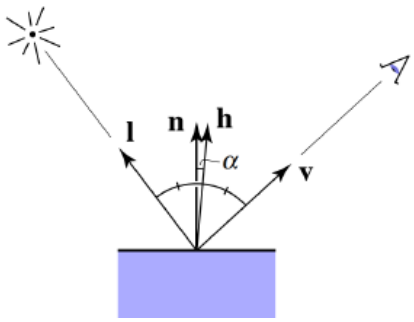

图 8: Blinn-Phong 模型

其计算公式如4所示：

$$L = k_s I_{light} \max(0, \mathbf{n} \cdot \mathbf{h})^p \tag{4}$$

其中 $\mathbf{h}$ 是半程向量，是单位向量，计算公式如下：

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{I}}{\|\mathbf{v} + \mathbf{I}\|} \tag{5}$$

可见，$\mathbf{n} \cdot \mathbf{h} = \cos\alpha$。

2.3.2 代码实现

实现代码如下所示：

```
virtual bool fragment(unsigned int index, Vec3f pos, Vec3f bar, TGAColor &color) {
    ......
    // use these vectors to finish your assignment.
    // the normal vector of this fragment.
    Vec3f normal = (B*models[index]->normal(uv)).normalize();
    // the lighting direction.
    Vec3f lightDir = light_dir;
    // the position of this fragment in world space.
    Vec3f fragPos = pos;
    // the viewing direction from eye to this fragment.
    Vec3f viewDir = (eye - fragPos).normalize();
    // the texture color of this fragment (sample from texture using uv coordinate)
    TGAColor fragColor = models[index]->diffuse(uv);
    static float ka = 0.0f;
    static float kd = 0.8f;
    static float ks = 0.4f;
    float diff = 1.0f;
    ......
    // TODO 2:
    // calculate the specular coefficient using Phong model, and then save it to spec.
```

```
    {
        int p=16;
        // blinn-phong
        Vec3f h=(viewDir+lightDir).normalize();
        float temp2=pow(normal*h,p);
        spec=ks*MAX(0,temp2);
    }
    TGAColor specColor = TGAColor(255,255,255)*spec;
    // the final color of this fragment taking lighting into account.
    color = diffColor + specColor;
    return false;
}
```

2.3.3 渲染结果与改进原因

使用 Blinn-Phong 渲染结果如图9所示。

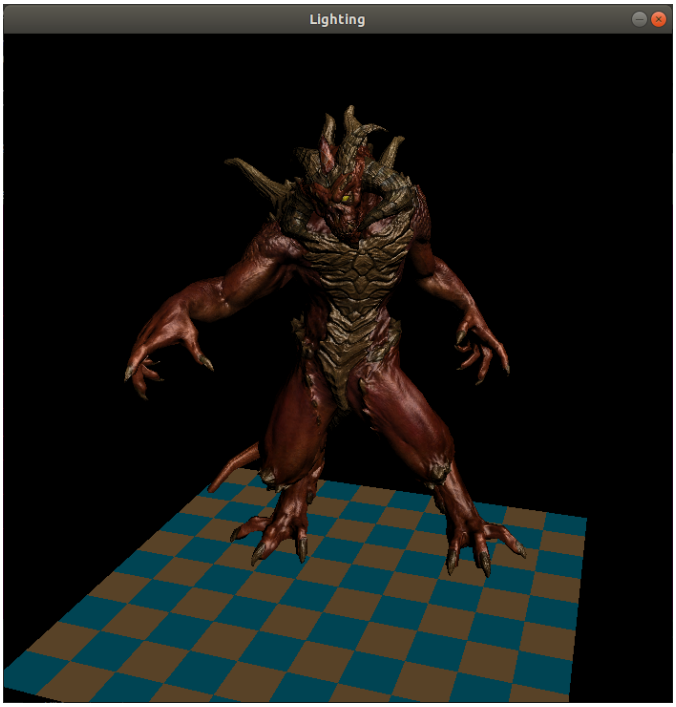

图 9: Blinn-Phong 渲染效果（$p = 16$）

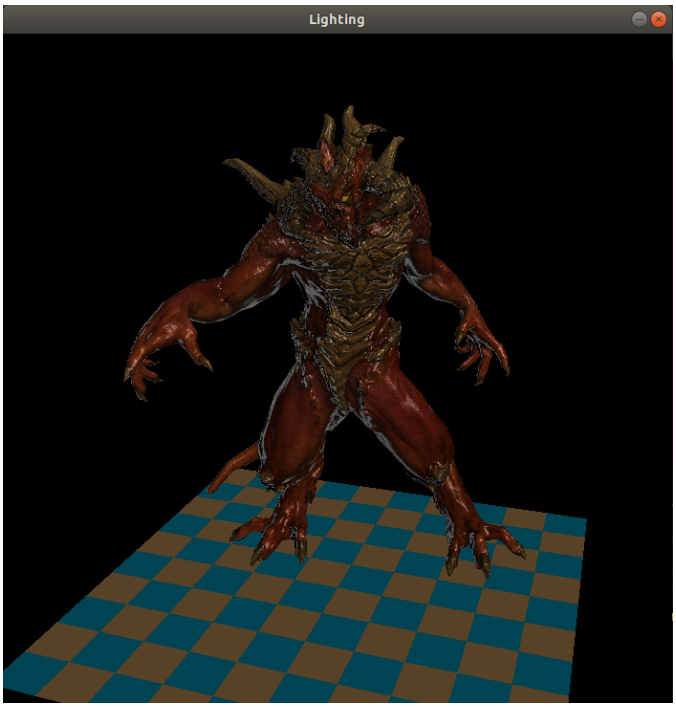

图 10: Phong 渲染效果（$p = 16$）

可见，相同 p 的情况下，Blinn-Phong 高光范围更大，而 Phong 模型更加真实。

Blinn-Phong 模型做出该改进的目的是在不损失太多真实性的情况下，加快运算速度，因为计算 h 比计算反射向量需要的运算更为简单。在实际运行中也的确有此感受，使用 Blinn-Phong 模型后图像的渲染速度比 Phong 模型稍微快一点。

2.4 Task4：实现 eye 绕 y 旋转的动画

2.4.1 计算逻辑

假设我们需要将 eye 绕 y 轴每帧旋转一定角度，而 $eye = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$，则我们有：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6}$$

因为在代码中，eye 为三维向量，故我们可以将计算公式简化为：

$$\begin{cases} x' = \cos\theta x + \sin\theta z \\ y' = y \\ z' = -\sin\theta x + \cos\theta z \end{cases}$$

2.4.2 代码实现

实现代码如下所示：

```
int main(int argc, char** argv) {
    ......
    while(key != 27)
    {
        ......
        {
            // do something to variable "eye" here to achieve rotation.
            eye=rotate_y(5); // 实现旋转
        }
        ......
    }
    ......
}

Vec3f rotate_y(float angle)
{
    float theta=M_PI*angle/180; // 将角度转换成弧度
    float cos_angle=cos(theta);
    float sin_angle=sin(theta);
    Vec3f new_eye(cos_angle*eye.x+sin_angle*eye.z, eye.y, -sin_angle*eye.x+cos_angle*eye.z
        );
    return new_eye;
}
```

其中我们将 eye 绕 y 轴旋转封装成函数 rotate_y，然后在 main() 函数中调用，实现 eye 的旋转。

2.4.3 渲染结果

渲染效果如图11、12和13所示，这是从不同角度获取的渲染图。

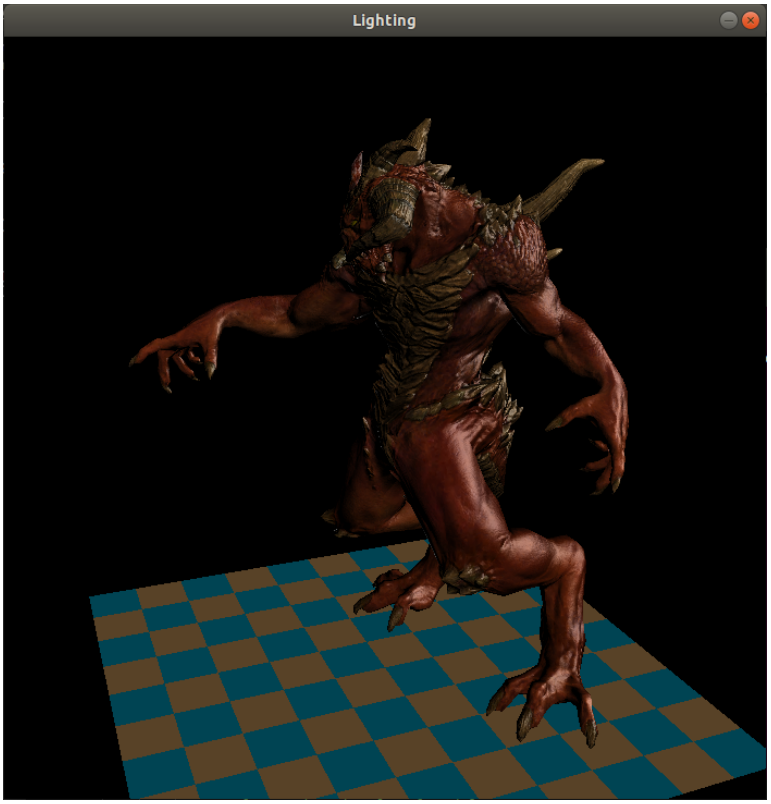

图 11: 渲染图 1

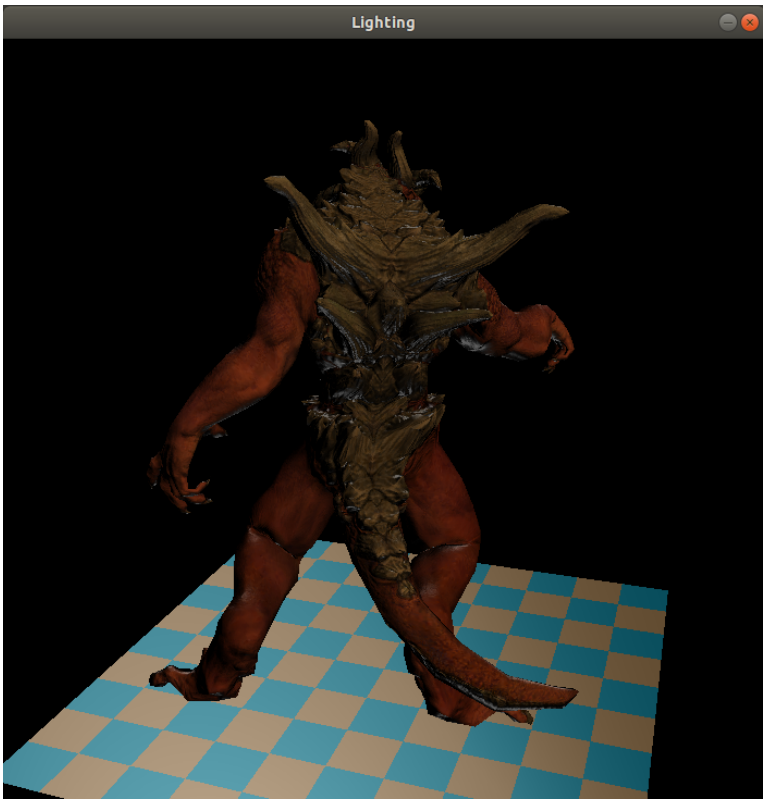

图 12: 渲染图 2

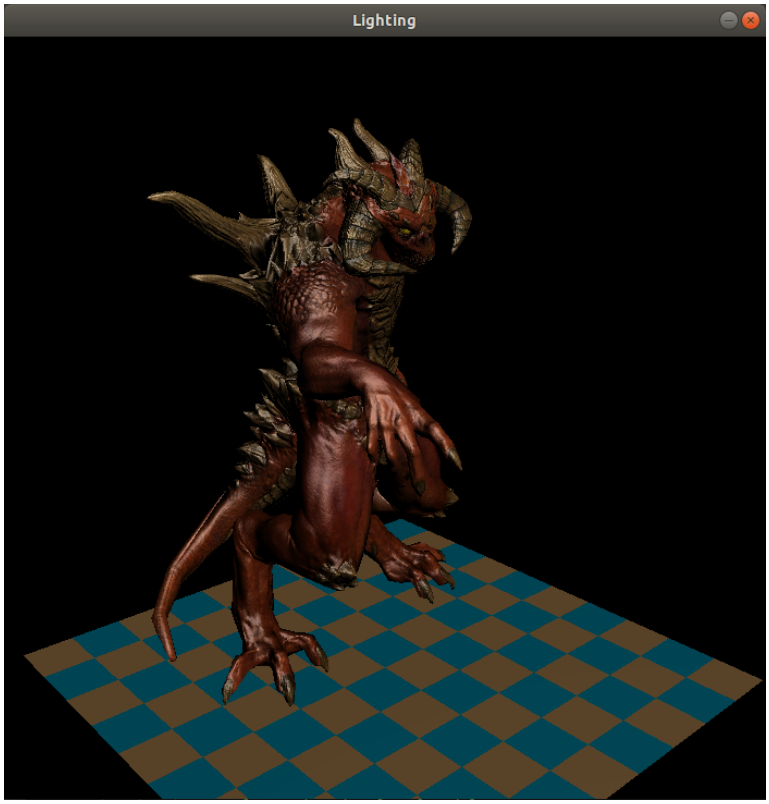

图 13: 渲染图 3

2.5 Task5：在 OpenGL 中实现传统的光照模型

选做任务要求我们学习教程，在 OpenGL 中实现传统的光照模型。

相比于第 0 次作业，在光照教程前面还讲了很多其他未学习的内容，如变换、坐标系统、摄像机等。在这里我们不再讨论，只讨论光照的实现。

2.5.1 漫反射和镜面光的实现

在 OpenGL 中，光照的实现主要在片段着色器中完成。我们有三种光照：环境光、漫反射和镜面反射。

对于环境光照，其实现十分简单，就是将光的颜色乘以一个常量环境因子，再乘以物体颜色，最终结果即为片段颜色。对于漫反射光照和镜面反射光照，我们采用 Phong 模型实现，具体计算原理见2.1.1和2.2.1。最终在实现代码如下所示：

```
#version 330 core
out vec4 FragColor;          // 片段颜色

in vec3 Normal;              // 法向量
in vec3 FragPos;             // 片段位置

uniform vec3 lightPos;       // 光源位置
uniform vec3 viewPos;        // 视点位置
uniform vec3 lightColor;     // 光照颜色
uniform vec3 objectColor;    // 物体颜色

```

```
void main()
{
    // 渲染环境光
    float ambientStrength = 0.1; // 环境因子
    vec3 ambient = ambientStrength * lightColor;// 环境光照

    // 渲染漫反射光
    vec3 norm = normalize(Normal);// 法向量的单位向量
    vec3 lightDir = normalize(lightPos - FragPos);// 光照向量
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;              // 计算漫反射

    // 渲染镜面反射光
    float specularStrength = 0.5;
    vec3 viewDir = normalize(viewPos - FragPos);// 视线向量
    vec3 reflectDir = reflect(-lightDir, norm); // 反射向量
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;// 计算镜面反射

    // 汇总计算结果
    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}
```

2.5.2 材质和光照贴图的实现

在现实世界中，物体的不同材质会导致不同的环境光照、漫反射光照和镜面光照。对于某一种材质，我们可以设置三个三维向量作为相应系数分别对应不同类型的光照。在计算某一种光照时，通过将其与原始光照输出点乘，即可实现这种材质特有的光照效果。

相比材质，更常用的一种实现不同光照的方法是光照贴图。虽然教程里面的“光照贴图”章节似乎写得很高大上，但实际上就是把纹理应用到光照中……在2.5.1中，光照的实现是基于物体的颜色向量*objectColor*。而当我们应用纹理后，在计算环境光照和漫反射光照时，我们使用相应的漫反射光照贴图的纹理向量替换颜色向量；在计算镜面反射光照时，使用相应的镜面光照贴图的纹理向量替换颜色向量。实现代码如下所示：

```
#version 330 core
out vec4 FragColor;      // 输出片段颜色
struct Material {        // 材质（纹理）结构体
    sampler2D diffuse;   // 漫反射光照贴图
    sampler2D specular; // 镜面反射光照贴图
    float shininess;
};
struct Light {           // 光源结构体
    vec3 position;       // 光源位置
    vec3 ambient;        // 环境光照系数
    vec3 diffuse;        // 漫反射光照系数
    vec3 specular;       // 镜面反射光照系数
```

```
};
in vec3 FragPos;          // 片段位置
in vec3 Normal;           // 法向量
in vec2 TexCoords;        // 纹理坐标
uniform vec3 viewPos;     // 视点位置
uniform Material material;
uniform Light light;
void main()
{
    // 计算环境光
    vec3 ambient = light.ambient * texture(material.diffuse, TexCoords).rgb;

    // 计算漫反射
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = light.diffuse * diff * texture(material.diffuse, TexCoords).rgb;

    // 计算镜面反射
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    vec3 specular = light.specular * spec * texture(material.specular, TexCoords).rgb;
    // 综合结果
    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

2.5.3 不同类型投光物和多光源的实现

在教程中，介绍了三种投光物：平行光、点光源和聚光源。

对于平行光，其没有位置、只有方向向量，不会出现衰减。因此，在计算其漫反射和镜面反射时，我们直接使用其方向向量即可，不需要通过光源位置和物体位置计算光照方向向量。其代码如下所示：

```
// 定向光
struct DirLight {
    vec3 direction; // 方向：从光源至片段

    vec3 ambient;   // 漫反射
    vec3 diffuse;   // 镜面反射
    vec3 specular;  // 反光度
};
// 平行光照计算
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);// 光照方向取反：现从片段至光源
    // 计算漫反射
    float diff = max(dot(normal, lightDir), 0.0);
```

```
    // 计算镜面反射
    vec3 reflectDir = reflect(-lightDir, normal); // 得到反射向量
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // combine results
    // 合并结果
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    return (ambient + diffuse + specular);
}
```

对于点光源，其相比平行光源有两个不同：

(1) 其有位置向量，我们需要通过物体位置和光源位置计算光照方向向量；

(2) 它发射的光线会随着传播距离增长而衰减。

第一点很容易解决，事实上，前面代码我们都是这样子完成的。而对于第二点，教程提供了一个公式：

$$F_{att} = \frac{1.0}{K_c + k_l * d + K_q * d^2} \tag{7}$$

F_{att} 是衰减比例，值域为 $F_{att} \in [0, 1]$。d 为距离，K_c, K_l 和 K_q 分别为常数项、一次项和二次项。这样，当 d 较小时，光照会以线性的方式衰退，当距离足够大时，就会以更快的速度下降。至于这些值如何选取，这需要通过实验比较进行选择（毕竟这是经典局部照明）。得到衰减比例后，我们将其与最终片段颜色相乘即可。实现代码如下所示：

```
// 点光源
struct PointLight {
    vec3 position;  // 位置

    float constant; // 衰减值常数项
    float linear;   // 衰减值一阶项
    float quadratic;// 衰减值二阶项
    vec3 ambient;   // 漫反射
    vec3 diffuse;   // 镜面反射
    vec3 specular;  // 反光度
};
// 点光源计算
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    // 得到光照向量
    vec3 lightDir = normalize(light.position - fragPos);
    // 漫反射光照计算
    float diff = max(dot(normal, lightDir), 0.0);
    // 镜面反射光照计算
    vec3 reflectDir = reflect(-lightDir, normal); // 得到反射
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // 计算衰减
    float distance = length(light.position - fragPos);// 距离
```

```
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic
        * (distance * distance));// 计算衰减系数
    // 计算各光照结果
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    // 进行衰减
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

对于聚光光源，其是一种特殊的点光源，只会朝一个特定方向照射，如图14所示。

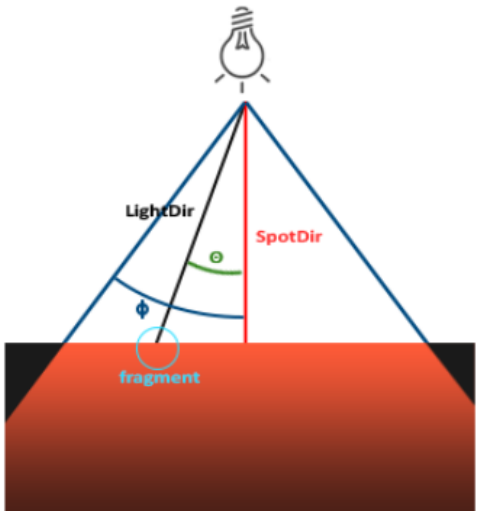

图 14: 聚光源

内切光角 ϕ 指定了聚光半径，外切光角 γ 用于平滑边缘。如果片段在内圆锥范围内即 $\theta < \phi$，那么该片段就会被照亮；如果 $\phi < \theta < \gamma$，那么片段的亮度将会递减；当片段在外圆锥范围外时，片段将不会被照亮，只有环境光照。

将上述转换成公式的话，如下所示：

$$I = \frac{\cos\theta - \cos\gamma}{\cos\phi - \cos\gamma} \tag{8}$$

其中 I 就是一个光照强度值，因为其值会出现大于 1 或小于 0 的情况，我们可以使用 clamp() 函数进行约束，即可得到正确的值。最终实现代码如下所示：

```
// 聚光源
struct SpotLight {
    vec3 position;  // 位置
    vec3 direction; // 方向
    float cutOff;   // 近切光角的余弦值
    float outerCutOff;// 远切光角余弦值

    float constant; // 衰减值常数项
```

```
    float linear;    // 衰减值一阶项
    float quadratic;// 衰减值二阶项

    vec3 ambient;    // 漫反射
    vec3 diffuse;    // 镜面反射
    vec3 specular;   // 反光度
};
// 聚光源计算
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    // 光照向量
    vec3 lightDir = normalize(light.position - fragPos);
    // 漫反射计算
    float diff = max(dot(normal, lightDir), 0.0);
    // 镜面反射计算
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // 计算衰减
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic
        * (distance * distance));
    // 计算是否在聚光内外
    float theta = dot(lightDir, normalize(-light.direction));
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);// clamp函
        数：将第一个参数约束到第二/三个参数之间
    // 综合结果
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    ambient *= attenuation * intensity;
    diffuse *= attenuation * intensity;
    specular *= attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

当我们有多个光源时，在片段着色器中，就需要针对不同的光源分别计算一次环境光照、漫反射光照和镜面反射光照。最后将所有计算结果相加，即可得到最终的光照效果。

2.5.4 实现效果

最终实现效果如图15和16所示：

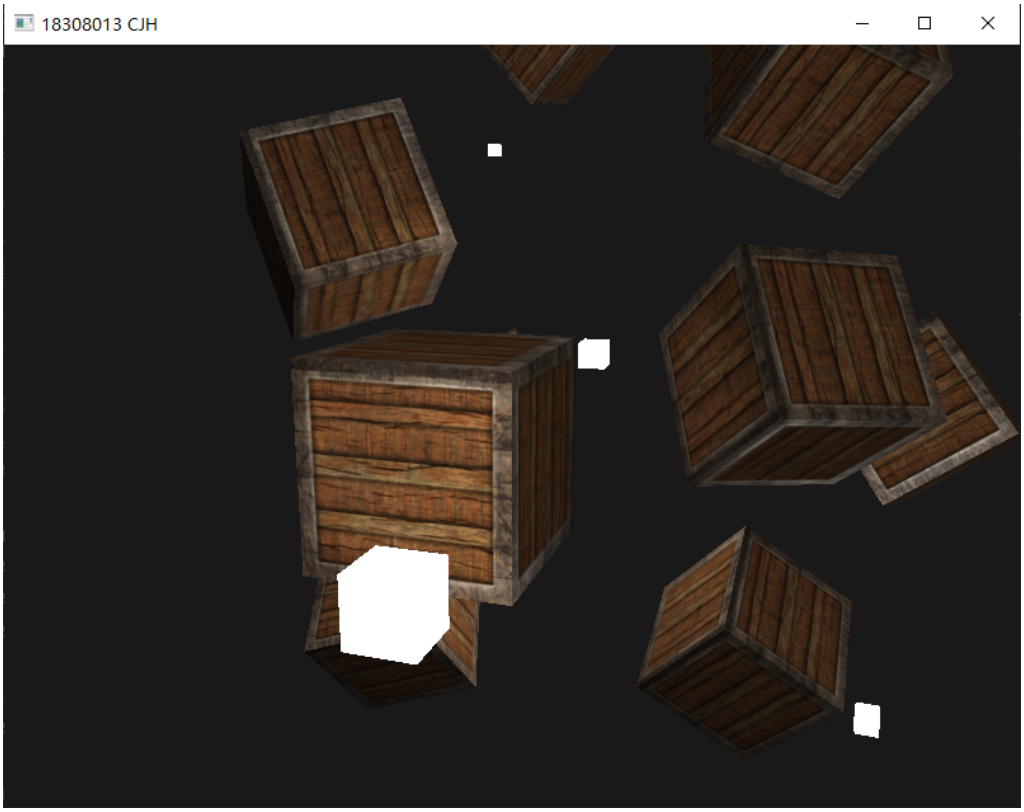

图 15: 实现效果 1

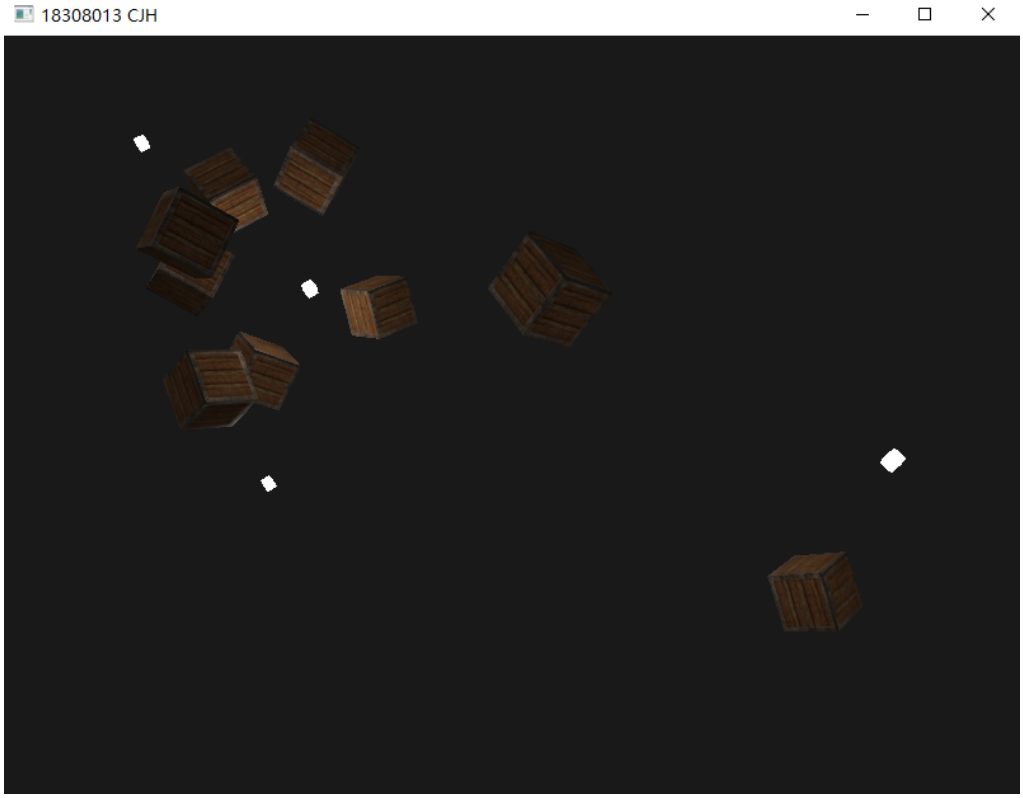

图 16: 实现效果 2

我们按照教程设计了平行光和四个点光源，并从视线方向建立一个聚光源充当“手电筒”。可见实现效果不错。

3 作业总结与感想

本次作业如果不做 OpenGL 的选做任务的话任务量并不是很大，但如果需要完成选做任务的话就要花很长时间去学习教程了。我花了几天时间阅读教程的代码，所幸的是最后弄明白了光照的实现原理，取得了不错的效果。

在此感谢老师和助教提供的帮助，希望我能顺利完成下一次实验。