CG 第六次作业

软件工程数字媒体 15331101 洪鹏圳

Basic:

1. 实现 Phong 光照模型

①实现 Phong Shading

（1）设置光源的初始位置以及绘制光源

设置光源的初始位置（全局变量）：

```
glm::vec3 lightPos(0.0f, 0.1f, 0.3f);//光源位置
```

创建一个表示光源的立方体，为这个光源创建一个专门的 VAO（与物体的 VAO 分开，以防对物体的顶点数据和属性做出修改时影响到灯）：

```
//灯源
unsigned int light_VAO;//顶点数组对象
glGenVertexArrays(1, &light_VAO);//生成一个VAO对象
glBindVertexArray(light_VAO);//绑定VAO
//把顶点数组复制到缓冲中供OpengGL使用
glBindBuffer(GL_ARRAY_BUFFER, cube_VBO);
//链接顶点属性
//位置属性，值为0
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//解析顶点数据
glEnableVertexAttribArray(0);
```

编写光源的顶点着色器和片段着色器：

```
//灯源顶点着色器源代码
#version 330 core
layout (location = 0) in vec3 aPos;//位置变量的属性位置值为0

uniform mat4 transform;//变换矩阵
uniform mat4 view;//观察矩阵
uniform mat4 projection;//投影矩阵

void main()
{
    gl_Position = projection * view * transform * vec4(aPos, 1.0f);//获得变换后的位置
}
```

```
//灯源片段着色器源代码
#version 330 core
out vec4 FragColor;

uniform vec3 lightColor;//光源颜色

void main()
{
    FragColor = vec4(lightColor, 1.0f);//将向量的4个分量都设置为1.0，表示灯源发出白光
}
```

绘制光源：

```
//绘制灯
lightingShader.use();
glm::mat4 model(1);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
lightingShader.setMat4("projection", p);
lightingShader.setMat4("view", v);
lightingShader.setMat4("transform", model);
lightingShader.setVec3("lightColor", lightColor);
glBindVertexArray(light_VAO);//绑定VAO
glDrawArrays(GL_TRIANGLES, 0, 36);//绘制图元
```

（2）冯氏光照模型的主要结构由 3 个分量组成：环境光照（Ambient）、漫反射（Diffuse）和镜面光照（Specular）

它们的定义分别为：

- 环境光照(Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色。
- 漫反射光照(Diffuse Lighting)：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮。
- 镜面光照(Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

所以接下来我们需要模拟这三种光照分量。

一．环境光照

环境光照的光的强度 = 光的颜色 * 环境光照强度

首先我们定义光源的颜色（初始化为白光），同时定义其三个分量 R、G、B 方便我们在 IMGUI 中修改：

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);//光源颜色，默认是白光
float r = 1.0f, g = 1.0f, b = 1.0f;//光源颜色的三个分量
```

在光源的片段着色器中加入 uniform 类型的光源颜色：

```
uniform vec3 lightColor;//光源颜色
```

同时我们在源代码中定义一个表示环境光照强度的初始值，初始化为 0.1：

```
float ambientStrength = 0.1;//环境光照强度
```

接着在 PhongShader 的片段着色器中声明一个 uniform 类型的光源颜色和环境光照强度：

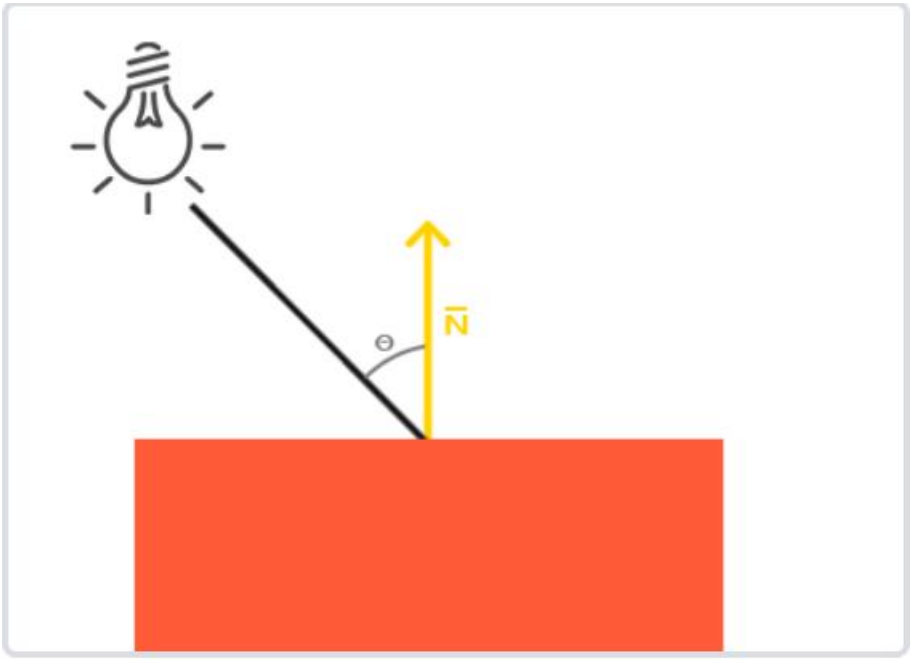
```
uniform vec3 lightColor;//光源颜色
uniform float ambientStrength;//环境光照强度
```

计算环境光照的光的强度：

```
//环境光照
vec3 ambient = ambientStrength * lightColor;//环境光照
```

二．漫反射

漫反射光照示意图为：



我们需要计算入射光线和入射平面法向量的夹角，该夹角为光源对当前片段实际的漫反射的影响，这一步我们通过入射方向向量和法向量的点乘得到。

注意：为了得到两个向量夹角的余弦值，我们需要将这两个向量单位化，否则点乘返回的就不是余弦值了，而是余弦值乘上这两个向量的长度。

所以，要计算漫反射的光照，我们需要计算以下两个分量：

①法向量：一个垂直于顶点表面的向量。

法向量是一个垂直于顶点表面的单位向量，由于顶点本身并没有表面，我们利用它周围的顶点来计算出这个顶点的表面。为了避免繁琐的运算，我们可以把法向量数据手工添加到顶点数据中：

```
float vertices[] = {
    //平面1                  //平面1的法向量
    -0.2f, -0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
     0.2f, -0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
     0.2f,  0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
     0.2f,  0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
    -0.2f,  0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
    -0.2f, -0.2f, -0.2f,     0.0f, 0.0f, -1.0f,
    //平面2                  //平面2的法向量
    -0.2f, -0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
     0.2f, -0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
     0.2f,  0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
     0.2f,  0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
    -0.2f,  0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
    -0.2f, -0.2f,  0.2f,     0.0f, 0.0f, 1.0f,
    //平面3                  //平面3的法向量
    -0.2f,  0.2f,  0.2f,     -1.0f, 0.0f, 0.0f,
    -0.2f,  0.2f, -0.2f,     -1.0f, 0.0f, 0.0f,
    -0.2f, -0.2f, -0.2f,     -1.0f, 0.0f, 0.0f,
    -0.2f, -0.2f, -0.2f,     -1.0f, 0.0f, 0.0f,
    -0.2f, -0.2f,  0.2f,     -1.0f, 0.0f, 0.0f,
    -0.2f,  0.2f,  0.2f,     -1.0f, 0.0f, 0.0f,
    //平面4                  //平面4的法向量
     0.2f,  0.2f,  0.2f,     1.0f, 0.0f, 0.0f,
     0.2f,  0.2f, -0.2f,     1.0f, 0.0f, 0.0f,
     0.2f, -0.2f, -0.2f,     1.0f, 0.0f, 0.0f,
     0.2f, -0.2f, -0.2f,     1.0f, 0.0f, 0.0f,
     0.2f, -0.2f,  0.2f,     1.0f, 0.0f, 0.0f,
     0.2f,  0.2f,  0.2f,     1.0f, 0.0f, 0.0f,
```

同时应该更新 PhongShader 的顶点着色器：

```
//冯氏顶点着色器源代码
#version 330 core
layout (location = 0) in vec3 aPos;//位置变量的属性位置值为0
layout (location = 1) in vec3 aNormal;//法向量的属性位置值为1
```

接着我们需要更新一下物体的链接顶点属性：

```
//链接顶点属性
//位置属性，值为0
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//解析顶点数据
glEnableVertexAttribArray(0);
//法向量属性，值为1
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));//解析顶点数据
glEnableVertexAttribArray(1);
```

注意我们不需要更新光源的链接顶点属性，因为其不需要用到法向量属性，但是步长还是需要设置为 6：

```
//链接顶点属性
//位置属性，值为0
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//解析顶点数据
glEnableVertexAttribArray(0);
```

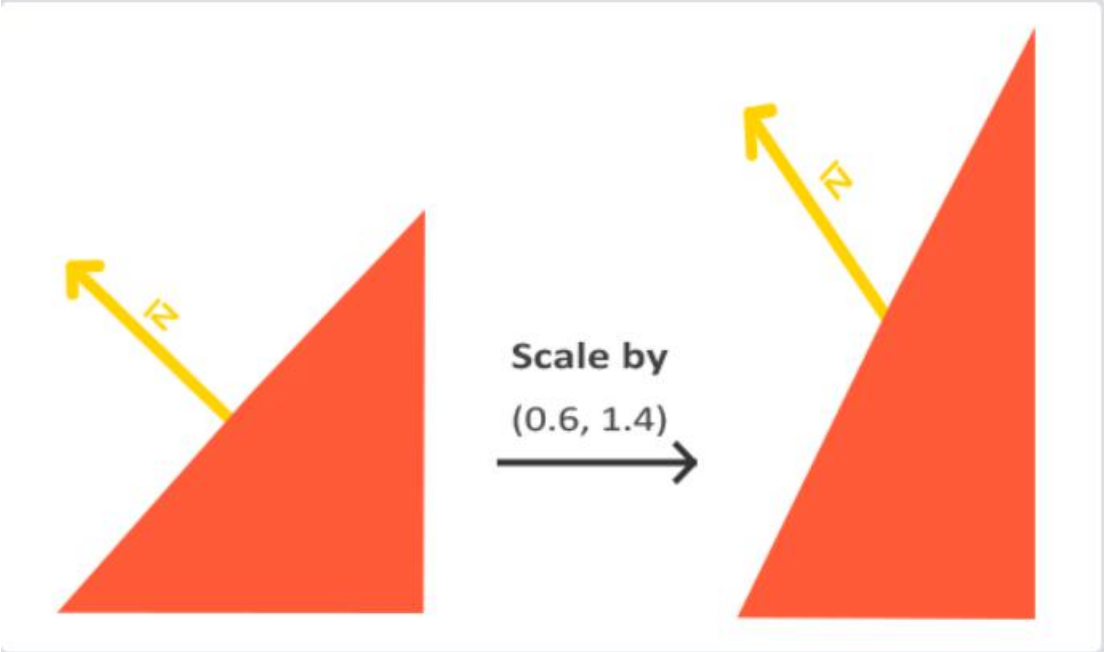
> 虽然对灯的着色器使用不能完全利用的顶点数据看起来不是那么高效，但这些顶点数据已经从箱子对象载入后开始就储存在GPU的内存里了，所以我们并不需要储存新数据到GPU内存中。这实际上比给灯专门分配一个新的VBO更高效了。

当有了法向量后，我们需要把法向量从顶点着色器传递到片段着色器

在 PhongShader 的顶点着色器中：

```
out vec3 Normal;//顶点法向量
```

因为如果物体做了不等比缩放时，顶点的改变会导致法向量不再垂直与表面：



因此我们需要生成一个法线矩阵（对变换矩阵先求逆再转置）来使得法向量能一直垂直于平面：

```
Normal = mat3(transpose(inverse(transform))) * aNormal;
```

在 PhongShader 的片段着色器中：

```
in vec3 Normal;//法向量
```

②入射光线：作为光源的位置与片段的位置之间向量差的方向向量。为了计算这个光线，我们需要光的位置向量和片段的位置向量。

要获得光源的位置向量我们需要在 Phong Shader 片段着色器中声明一个光源位置向量的 uniform 变量：

```
uniform vec3 lightPos;//光源位置
```

接着在绘制函数里将前面声明的全局变量 lightPos 向量传入 Phong Shader 片段着色器中：

```
objectShader.setVec3("lightPos", lightPos);
```

同时我们还需要片段的位置，我们先将顶点位置转换到世界空间坐标，在 PhongShader 的顶点着色器中：

```
out vec3 Normal;//顶点法向量

FragPos = vec3(transform * vec4(aPos, 1.0f));//片段位置为顶点位置属性乘变换矩阵
```

最后在 PhongShader 的片段着色器添加片段位置的输入变量：

```
in vec3 FragPos;//片段位置
```

现在我们已经可以计算光源和片段位置之间的方向向量了。因为光的方向向量是光源位置向量与片段位置向量之间的向量差，因此我们用光源位置向量减去片段位置向量得到光的方向向量，同时把他们都转换为单位向量，保证最后算出来的是正确的夹角余弦值：

```
vec3 norm = normalize(Normal);//标准化后的法向量
vec3 lightDir = normalize(lightPos - FragPos);//标准化后的光的方向向量
```

接着我们声明一个 uniform 类型的漫反射强度，方便我们在 imGUI 中对其修改：

```
uniform float diffuseStrength;//漫反射强度
```

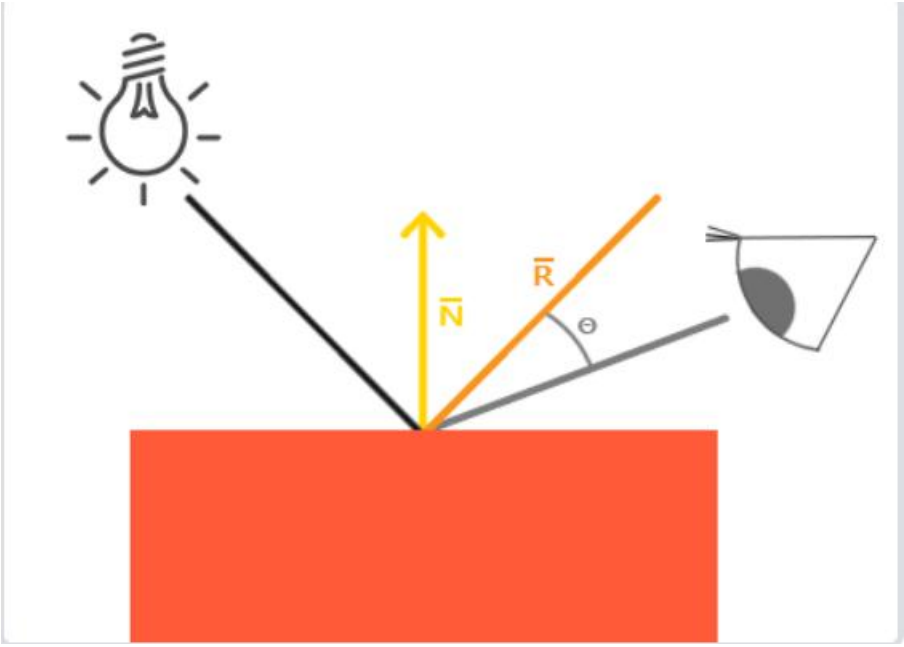
最后我们将光线向量和法向量进行点乘得到夹角余弦值，将余弦值与光的颜色相乘得到漫反射分量。两个向量之间的角度越大，漫反射分量就会越小：

```
float diff = max(dot(norm,lightDir),0.0f);//计算光源对当前片段实际的漫反射影响，即夹角
vec3 diffuse = diff * lightColor * diffuseStrength;//漫反射分量
```

注意：如果入射光线向量和法向量之间的夹角大于 90 度点乘的结果就会变成负数，因此我们使用 max 函数来保证漫反射影响为正数。

三．镜面反射

镜面反射的示意图如下：



我们需要计算反射向量和视线方向得角度差，如果夹角越小，那么镜面光的影响就会越大。它的作用效果就是，当我们去看光被物体所反射的那个方向的时候，我们就会看到一个高光。因此我们接下来要计算观察向量和反射向量：

①观察向量

观察者的世界空间坐标即摄像机对象的位置，我们在 PhongShader 的片段着色器中声明一个 uniform 类型的观察者世界空间坐标：

```
uniform vec3 viewPos;//摄像机位置
```

然后在源代码中把相应的摄像机位置坐标传给片段着色器：

```
objectShader.setVec3("viewPos", camera.Position);
```

计算观察向量：

```
vec3 viewDir = normalize(viewPos - FragPos);//视线方向向量
```

②反射向量

首先需要计算视线方向向量，接着计算沿着法线轴的反射向量：

```
vec3 reflectDir = reflect(-lightDir,norm);//反射向量
```

注意：reflect 函数中第一个向量是从光源指向片段位置的向量，而我们前面计算的 lightDir 是从片段指向光源：

```
vec3 lightDir = normalize(lightPos - FragPos);//标准化后的光的方向向量
```

因此我们需要对其取反，reflect 函数的第二个参数要求是一个法向量，我们传入的是已经标准化的法向量 norm，通过 reflect 函数我们可以计算反射向量 reflectDir。

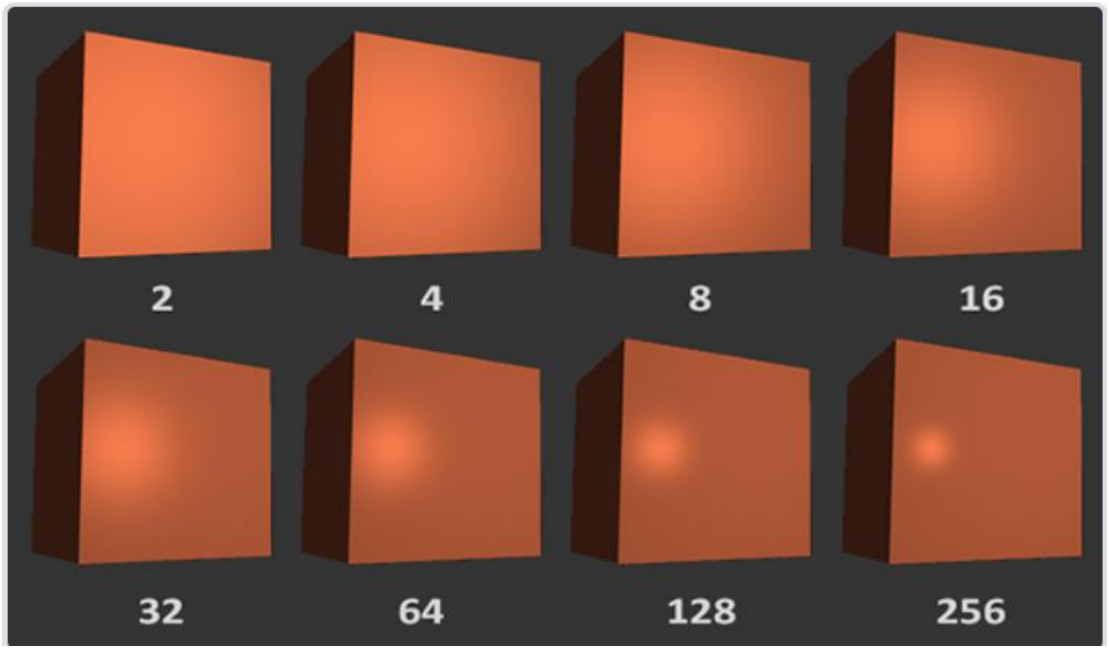
最后，我们需要计算镜面反射的光强度，首先我们声明一个 uniform 类型的镜面强度和一个 uniform 类型的反光度，方便我们在 imGUI 中修改：

```
uniform float specularStrength;//镜面强度
uniform int shiniess;//反光度
```

接着计算镜面反射的光强度：

```
float spec = pow(max(dot(viewDir,reflectDir),0.0),shiniess);
vec3 specular = specularStrength *  spec * lightColor;//镜面分量
```

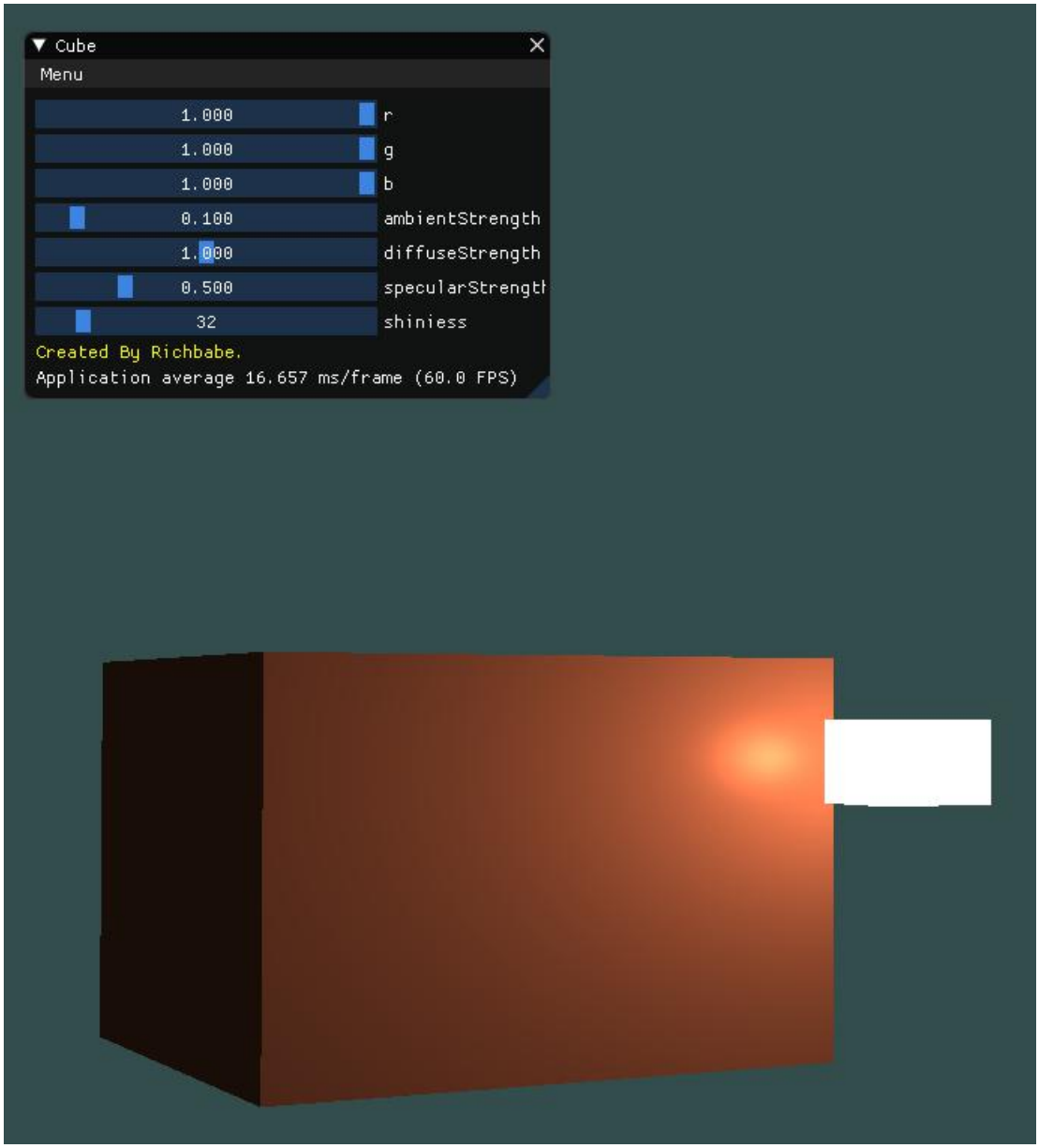
我们首先计算视线方向与反射方向得点乘（保证它不是复制），接着取它的 shiniess 次幂。shiniess 是高光的反光度。一个物体的反光度越高，反射光的能力越强，散射得越少，高光点就会越小：



当我们计算完环境光分量、漫反射分量、镜面反射分量后，在 PhongShader 的片段着色器我们将其加起来，再用和乘以物体的颜色，我们就得到了冯氏光照模型的全部光照分量：

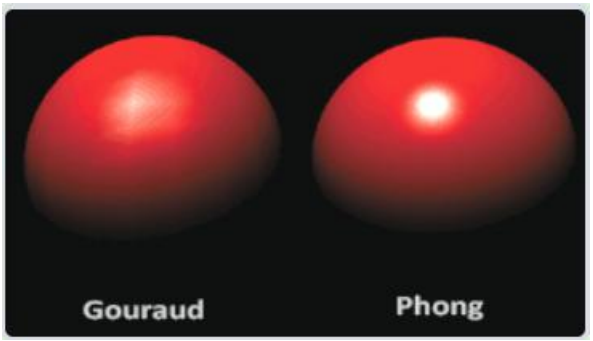
```
vec3 result = (ambient + diffuse + specular) * objectColor;
FragColor = vec4(result, 1.0f);
```

运行结果为：



②实现 Gouraud Shading

Gouraud Shading 与 Phong Shading 的不同之处在于：Gouraud Shading 是在顶点着色器实现冯氏光照模型，而 Phong Shadinng 是在片段着色器实现冯氏光照模型。相对于片段来说，顶点要少得多，因此 Gouraud Shading 比 Phong Shading 的计算量要少很多，但是光照看起来不会非常真实：



Gouraud Shading 的实现如下（也是用了冯氏光照模型，因此实现说明这里略过）：

（1）Gouraud Shading 的顶点着色器：

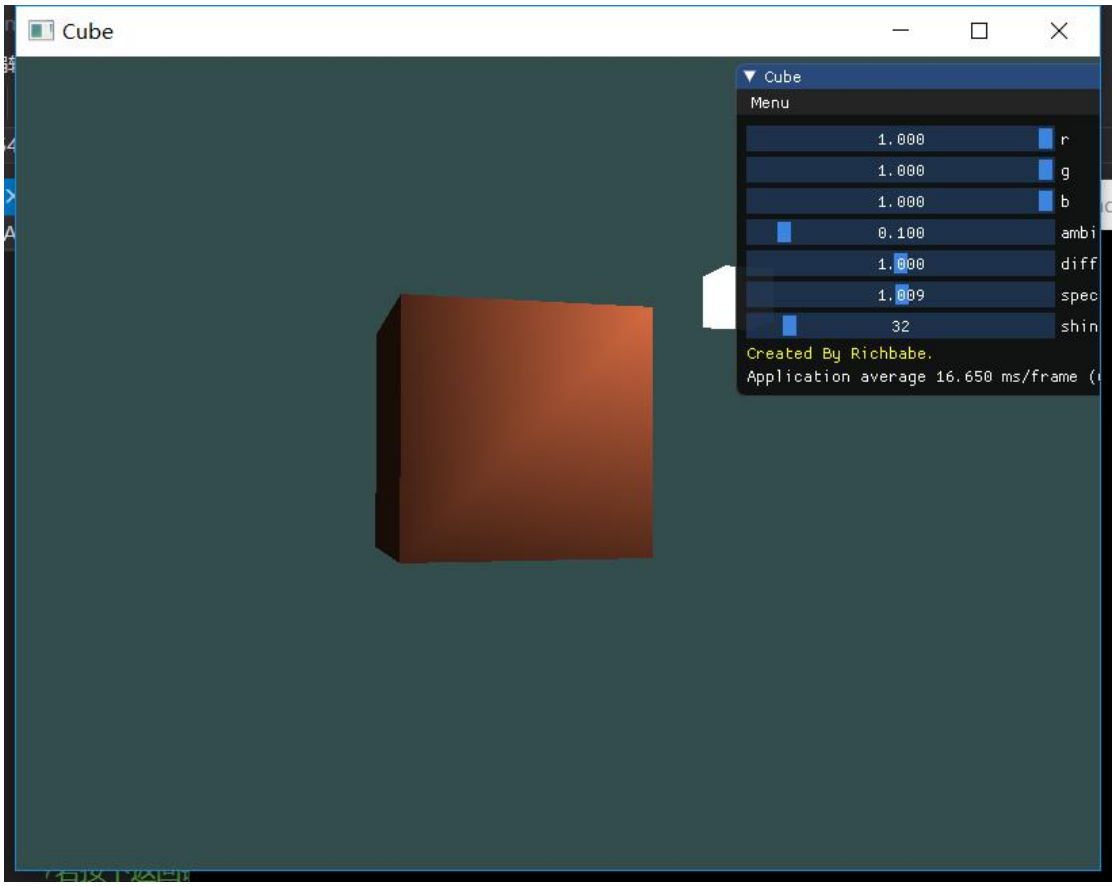
```
//Gouraud顶点着色器源代码
#version 330 core
layout (location = 0) in vec3 aPos;//位置变量的属性位置值为0
layout (location = 1) in vec3 aNormal;//法向量的属性位置值为1

out vec3 LightingColor;//通过Gouraud着色后的光照颜色

uniform mat4 transform;//变换矩阵
uniform mat4 view;//观察矩阵
uniform mat4 projection;//投影矩阵

uniform vec3 lightPos;//光源位置
uniform vec3 lightColor;//光源颜色
uniform vec3 viewPos;//摄像机位置

uniform float ambientStrength;//环境光照强度
uniform float diffuseStrength;//漫反射强度
uniform float specularStrength;//镜面强度
uniform int shiniess;//反光度

void main()
{
    gl_Position = projection * view * transform * vec4(aPos, 1.0f);//获得变换后的位置

    vec3 Position = vec3(transform * vec4(aPos, 1.0f));//片段位置为顶点位置属性乘变换矩阵
    vec3 Normal = mat3(transpose(inverse(transform))) * aNormal;

    //环境光照
    vec3 ambient = ambientStrength * lightColor;//环境光照

    //漫反射光照
    vec3 norm = normalize(Normal);//标准化后的法向量
    vec3 lightDir = normalize(lightPos - Position);//标准化后的光的方向向量
    float diff = max(dot(norm,lightDir),0.0f);//计算光源对当前片段实际的漫发射影响，即夹角
    vec3 diffuse = diff * lightColor * diffuseStrength;//漫反射分量

    //镜面光照
    vec3 viewDir = normalize(viewPos - Position);//视线方向向量
    vec3 reflectDir = reflect(-lightDir,norm);//反射向量
    float spec = pow(max(dot(viewDir,reflectDir),0.0),shiniess);
    vec3 specular = specularStrength *  spec * lightColor;//镜面分量

    LightingColor = ambient + diffuse + specular;

}
```

（2）Gouraud Shading 的片段着色器

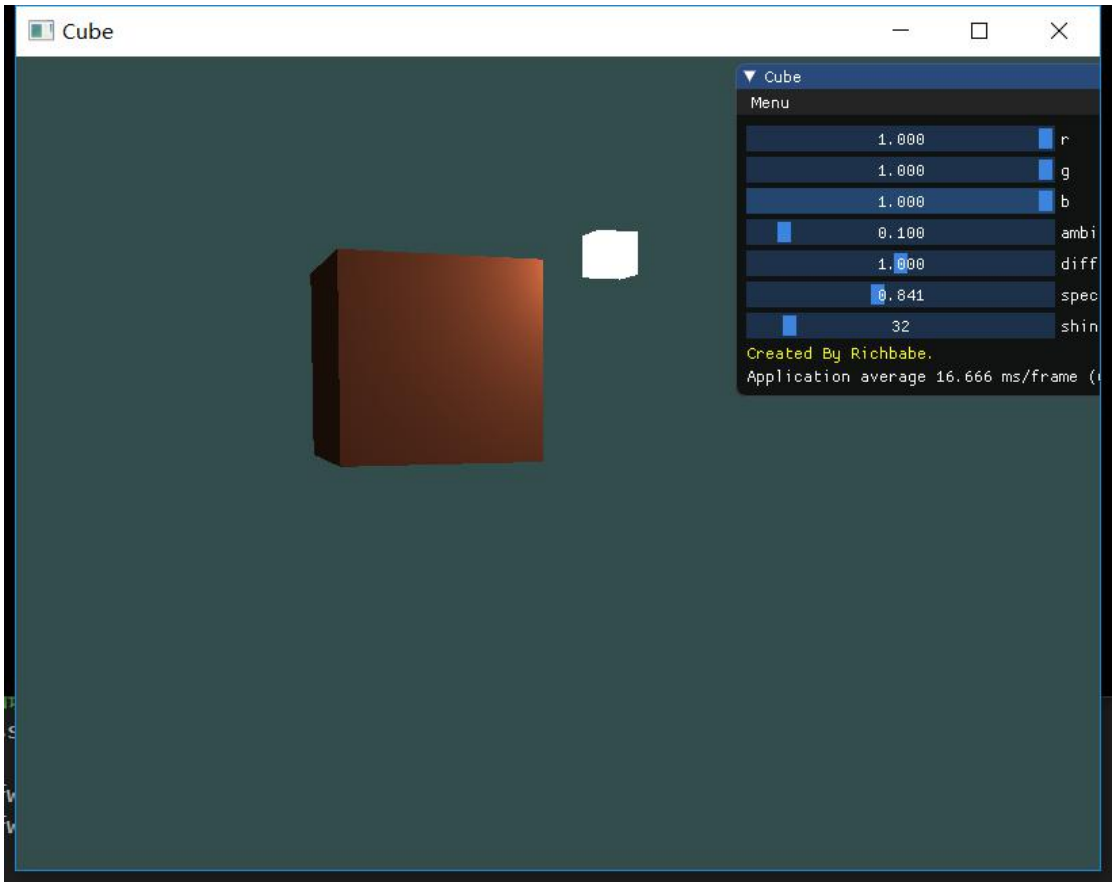
```
//Gouraud片段着色器源代码
#version 330 core
out vec4 FragColor;

in vec3 LightingColor;

uniform vec3 objectColor;//物体反射各个颜色分量的大小

void main(){
    FragColor = vec4(LightingColor * objectColor,1.0f);
}
```

运行结果：

Gouraud Shading：



Phong Shading：



可以看到因为 Gouraud 的马赫带效应，在两个三角形的交界处有一条明显的亮边，当使用更复杂的形状时，这种效果会变得更加明显。其原因是因为对于立方体正面的两个三角形右上方顶点用镜面高光照明，而这两个三角形的另外两个顶点没有。而立方体正面的中间片段的颜色不是直接来自光源，而是来自顶点的插值，因此在两个三角形之间会产生一条亮边。

2. 使用 GUI，使参数可调节，效果实时更改：
在源代码中声明如下光照参数：

```
//光照参数声明
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);//光源颜色，默认是白光
float r = 1.0f, g = 1.0f, b = 1.0f;//光源颜色的三个分量
float ambientStrength = 0.1;//环境光照强度
float diffuseStrength = 1.0;//漫反射强度
float specularStrength = 0.5;//镜面强度
int shiniess = 32;//反光度
```

在 While 渲染循环中通过调用 ImGui 的 SliderFloat 和 SliderInt 来实现冯氏光照模型参数的修改：

```
//GUI设置isPhongShading的参数
if (isPhongShading) {
    //设置光源颜色参数
    ImGui::SliderFloat("r", &r, 0.0, 1.0);
    ImGui::SliderFloat("g", &g, 0.0, 1.0);
    ImGui::SliderFloat("b", &b, 0.0, 1.0);
    lightColor.x = r;
    lightColor.y = g;
    lightColor.z = b;

    //设置环境光照强度
    ImGui::SliderFloat("ambientStrength", &ambientStrength, 0.0, 1.0);

    //设置漫反射强度
    ImGui::SliderFloat("diffuseStrength", &diffuseStrength, 0.0, 2.0);

    //设置镜面强度和反光度
    ImGui::SliderFloat("specularStrength", &specularStrength, 0.0, 2.0);
    ImGui::SliderInt("shiniess", &shiniess, 2, 256);
}
```

```
//GUI设置GouraudShading的参数
if (isGouraudShading) {
    //设置光源颜色参数
    ImGui::SliderFloat("r", &r, 0.0, 1.0);
    ImGui::SliderFloat("g", &g, 0.0, 1.0);
    ImGui::SliderFloat("b", &b, 0.0, 1.0);
    lightColor.x = r;
    lightColor.y = g;
    lightColor.z = b;

    //设置环境光照强度
    ImGui::SliderFloat("ambientStrength", &ambientStrength, 0.0, 1.0);

    //设置漫反射强度
    ImGui::SliderFloat("diffuseStrength", &diffuseStrength, 0.0, 2.0);

    //设置镜面强度和反光度
    ImGui::SliderFloat("specularStrength", &specularStrength, 0.0, 2.0);
    ImGui::SliderInt("shiniess", &shiniess, 2, 256);
}
```

运行效果见演示视频。

Bonus：

要实现光源在场景中来回移动，我们只需要在 While 渲染循环中通过调用 sin 和 cos 函数来实时更新光源的位置即可，实现如下：

```
//可移动光源
if (isMoveableLight) {
    float radius = 1.0f;
    lightPos.x = sin(glfwGetTime()) * radius;
    lightPos.y = sin(glfwGetTime() / 2.0) * 1.0f;
    lightPos.z = cos(glfwGetTime()) * radius;
    glm::mat4 trans(1);//变换矩阵
    glm::mat4 view(1);//观察矩阵
    view = camera.GetViewMatrix();
    glm::mat4 projection(1);//投影矩阵
    projection = glm::perspective(camera.Zoom, (float)screenWidth / (float)screenHeight, 0.1f, 100.0f);
    drawCube(trans, view, projection, PhongShader, lightingShader, lightColor, ambientStrength, diffuseStrength, specularStre
}
```

运行效果见演示视频。