Homework6 - Lights and Shading

Basic:

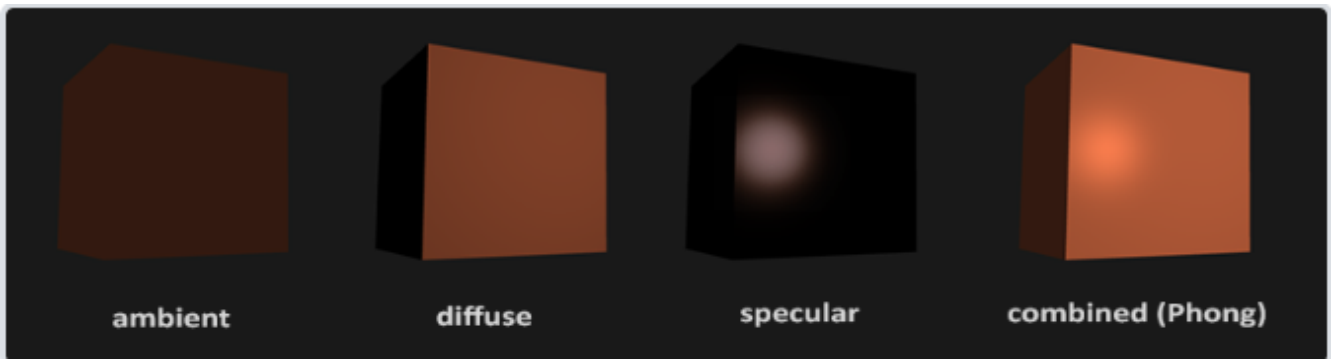
1. 实现Phong光照模型：
    - 场景中绘制一个cube
    - 自己写shader实现两种shading：Phong Shading和Gouraud Shading，并解释两种shading的实现原理
    - 合理设置视点，光照位置，光照颜色等参数，使光照效果明显显示
2. 使用GUI，使出参数可调节，效果实时更改：
    - GUI里可以切换两种shading
    - 使用如进度条这样的空间，使ambient因子，diffuse因子，specular因子，反射度等参数可调节，光照效果实时更改

Bonus:

当前光源为静止状态，尝试光源在场景中来回移动，光照效果实时更改。

> 在代码中，`main.cpp`是主函数，`shader.h`是之前实现的`shader`类，`camera.h`是之前实现的`camera`类，`h6shader1.vs`和`h6shader1.fs`是物体的`Phong Shading`的顶点着色器和片段着色器实现，`h6shader2.vs`和`h6shader2.fs`是光源的顶点着色器和片段着色器实现。`h6shader3.vs`和`h6shader3.fs`是物体的`Gouraud Shading`的顶点着色器和片段着色器实现。

现实世界的光照是极其复杂的，而且会受到诸多因素的影响，这是我们有限的计算能力所无法模拟的。因此OpenGL的光照使用的是简化的模型，对现实的情况进行近似，这样处理起来会更容易一些，而且看起来也差不多一样。这些光照模型都是基于我们对光的物理特性的理解。其中一个模型被称为冯氏光照模型(Phong Lighting Model)。冯氏光照模型的主要结构由3个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。下面这张图展示了这些光照分量看起来的样子：



- **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。

环境光照

光通常都不是来自于同一个光源，而是来自于我们周围分散的很多光源，即使它们可能并不是那么显而易见。光的一个属性是，它可以向很多方向发散并反弹，从而能够到达不是非常直接临近的点。所以，光能够在其它的表面上反射，对一个物体产生间接的影响。考虑到这种情况的算法叫做全局照明(Global Illumination)算法，但是这种算法既开销高昂又极其复杂。

由于我们现在对那种又复杂又开销高昂的算法不是很感兴趣，所以我们将会先使用一个简化的全局照明模型，即环境光照。正如你在上一节所学到的，我们使用一个很小的常量（光照）颜色，添加到物体片段的最终颜色中，这样子的话即便场景中没有直接的光源也能看起来存在有一些发散的光。

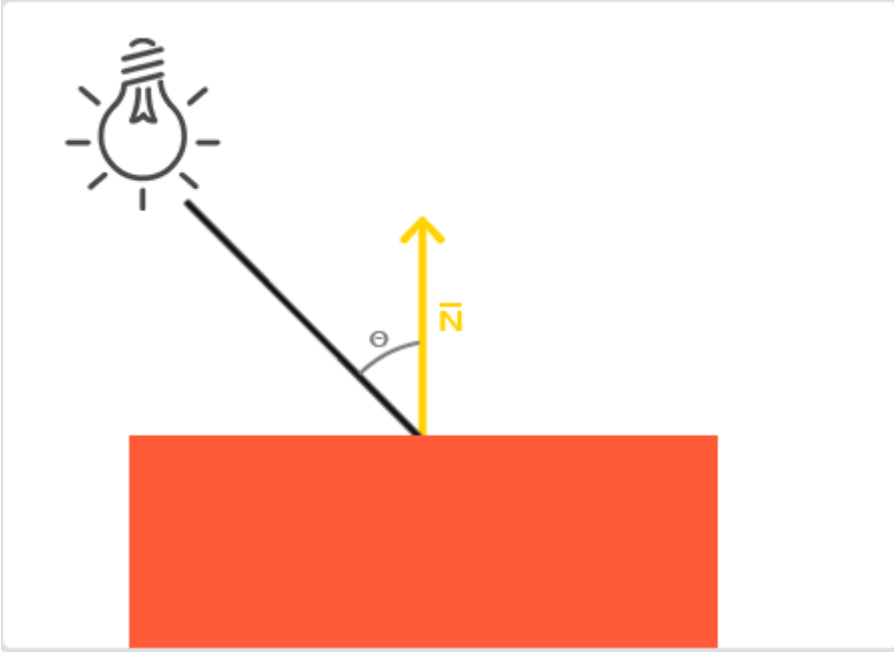
把环境光照添加到场景里非常简单。我们用光的颜色乘以一个很小的常量环境因子作为环境光照：

```
// ambient
// float ambientStrength = 0.1;
vec3 ambient = ambientStrength * lightColor;
```

这里的常量因子初始化为0.1，该因子被设置成uniform的类型，可以通过之后的GUI界面手动调整，之后的因子也都可以手动调整。

漫反射光照

环境光照本身不能提供最有趣的结果，但是漫反射光照就能开始对物体产生显著的视觉影响了。漫反射光照使物体上与光线方向越接近的片段能从光源处获得更多的亮度。为了能够更好的理解漫反射光照，请看下图：



图左上方有一个光源，它所发出的光线落在物体的一个片段上。我们需要测量这个光线是以什么角度接触到这个片段的。如果光线垂直于物体表面，这束光对物体的影响会最大化。为了测量光线和片段的角度，我们使用一个法向量(Normal Vector)，它是垂直于片段表面的一个向量（这里以黄色箭头表示）。两个单位向量的夹角越小，它们点乘的结果越倾向于1。当两个向量的夹角为90度的时候，点乘会变为0。这同样适用于θ，θ越大，光对片段颜色的影响就应该越小。

为了（只）得到两个向量夹角的余弦值，我们使用的是单位向量（长度为1的向量），所以我们需要确保所有的向量都是标准化的，否则点乘返回的就不仅仅是余弦值了。

点乘返回一个标量，我们可以用它计算光线对片段颜色的影响。不同片段朝向光源的方向的不同，这些片段被照亮的情况也不同。

计算漫反射光照需要什么?

- 法向量：一个垂直于顶点表面的向量。
- 定向的光线：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。

法向量

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（它只是空间中一个独立的点），我们利用它周围的顶点来计算出这个顶点的表面。我们能够使用一个小技巧，使用叉乘对立方体所有的顶点计算法向量，但是由于3D立方体不是一个复杂的形状，所以我们可以简单地把法线数据手工添加到顶点数据中。加入法向量之后的顶点数组如下所示：

```
97    // set up vertex data (and buffer(s)) and configure vertex attributes
98    float vertices[] = {
99        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
100       0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
101       0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
102       0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
103       -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
104       -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
105
106       -0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
107       0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
108       0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
109       0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
110       -0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
111       -0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,
112
113       -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,
114       -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,
115       -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,
116       -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,
117       -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,
118       -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,
119
120       0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,
121       0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,
122       0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,
123       0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,
124       0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,
```

```
        0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,

        -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,
        0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,
        0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,
        0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,

        -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,
        0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,
        0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,
        0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,
        -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,
        -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f
    };
```

每一行是一个顶点的数据前三个表示顶点位置，后三个表示顶点所在平面的法向量。

由于我们向顶点数组添加了额外的数据，所以我们应该更新光照的顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
```

现在我们已经向每个顶点添加了一个法向量并更新了顶点着色器，我们还要更新顶点属性指针。注意，灯使用同样的顶点数组作为它的顶点数据，然而灯的着色器并没有使用新添加的法向量。我们不需要更新灯的着色器或者是属性的配置，但是我们必须至少修改一下顶点属性指针来适应新的顶点数组的大小：

```
    // second, configure the light's VAO (VBO stays the same; the vertices are the same for
    unsigned int lightVAO;
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // note that we update the lamp's position attribute's stride to reflect the updated buf
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
```

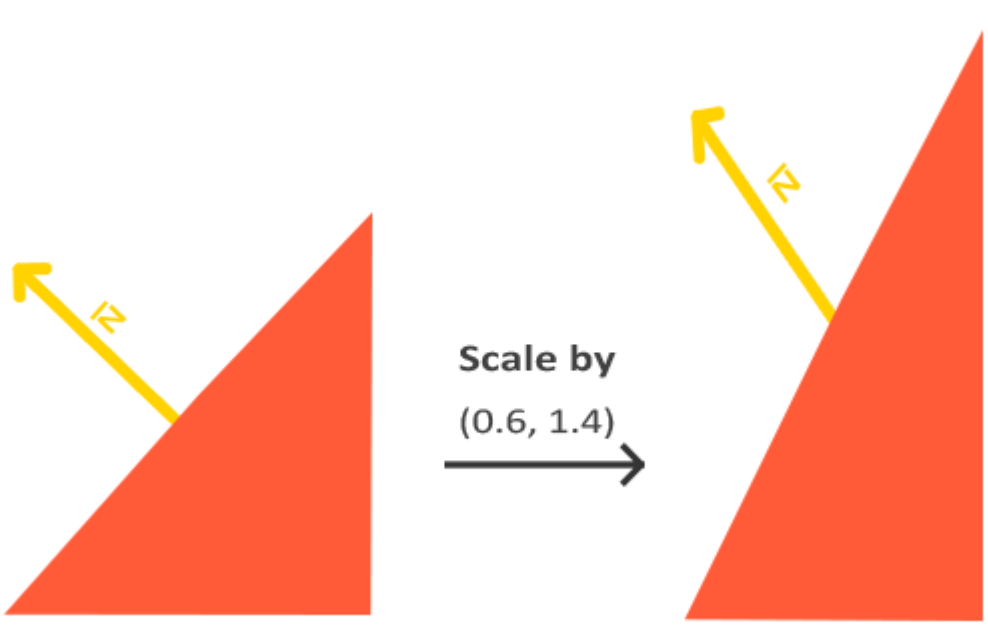
我们只想使用每个顶点的前三个float，并且忽略后三个float，所以我们只需要把**步长**参数改成`float`大小的6倍就行了。

所有光照的计算都是在片段着色器里进行，所以我们需要将法向量由顶点着色器传递到片段着色器。

目前片段着色器里的计算都是在世界空间坐标中进行的。所以，我们是不是应该把法向量也转换为世界空间坐标？基本正确，但是这不是简单地把它乘以一个模型矩阵就能搞定的。

首先，法向量只是一个方向向量，不能表达空间中的特定位置。同时，法向量没有齐次坐标（顶点位置中的w分量）。这意味着，位移不应该影响到法向量。因此，如果我们打算把法向量乘以一个模型矩阵，我们就要从矩阵中移除位移部分，只选用模型矩阵左上角3×3的矩阵（注意，我们也可以把法向量的w分量设置为0，再乘以4×4矩阵；这同样可以移除位移）。对于法向量，我们只希望对它实施缩放和旋转变换。

其次，如果模型矩阵执行了不等比缩放，顶点的改变会导致法向量不再垂直于表面了。因此，我们不能用这样的模型矩阵来变换法向量。下面的图展示了应用了不等比缩放的模型矩阵对法向量的影响：



每当我们应用一个不等比缩放时（注意：等比缩放不会破坏法线，因为法线的方向没被改变，仅仅改变了法线的长度，而这很容易通过标准化来修复），法向量就不会再垂直于对应的表面了，这样光照就会被破坏。

修复这个行为的诀窍是使用一个为法向量专门定制的模型矩阵。这个矩阵称之为法线矩阵(Normal Matrix)。

法线矩阵被定义为「模型矩阵左上角的逆矩阵的转置矩阵」。注意，大部分的资源都会将法线矩阵定义为应用到模型-观察矩阵(Model-view Matrix)上的操作，但是由于我们只在世界空间中进行操作（不是在观察空间），我们只使用模型矩阵。

在顶点着色器中，我们可以使用inverse和transpose函数自己生成这个法线矩阵，这两个函数对所有类型矩阵都有效。注意我们还要把被处理过的矩阵强制转换为3×3矩阵，来保证它失去了位移属性以及能够乘以`vec3`的法向量。

```
10  out vec3 Normal;
11
12  void main()
13  {
14      FragPos = vec3(model * vec4(aPos, 1.0));
15      Normal = mat3(transpose(inverse(model))) * aNormal;
16
```

接下来，在片段着色器中定义相应的输入变量：

```
14  in vec3 Normal;
```

计算漫反射光照

我们现在对每个顶点都有了法向量，但是我们仍然需要光源的位置向量和片段的位置向量。由于光源的位置是一个静态变量，我们可以简单地在片段着色器中把它声明为uniform：

```
4   uniform vec3 lightPos;
5   uniform vec3 objectColor;
6   uniform vec3 lightColor;
7   uniform vec3 viewPos;
8
```

然后在渲染循环中（渲染循环的外面也可以，因为它不会改变）更新uniform。我们使用在前面声明的lightPos向量作为光源位置：

```
34      // lighting
35      glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
36
```

```
225         (*lightingShader).setVec3("lightPos", lightPos);
```

最后，我们还需要片段的位置。我们会在世界空间中进行所有的光照计算，因此我们需要一个在世界空间中的顶点位置。我们可以通过把顶点位置属性乘以模型矩阵（不是观察和投影矩阵）来把它变换到世界空间坐标。这个在顶点着色器中很容易完成，所以我们声明一个输出变量，并计算它的世界空间坐标：

```
1   #version 330 core
2   layout (location = 0) in vec3 aPos;
3   layout (location = 1) in vec3 aNormal;
4
5   uniform mat4 model;
6   uniform mat4 view;
7   uniform mat4 projection;
8
9   out vec3 FragPos;
10  out vec3 Normal;
11
12  void main()
13  {
14      FragPos = vec3(model * vec4(aPos, 1.0));
15      Normal = mat3(transpose(inverse(model))) * aNormal;
16
17      gl_Position = projection * view * vec4(FragPos, 1.0);
18  }
```

最后，在片段着色器中添加相应的输入变量。

```
15  in vec3 FragPos;
16
```

现在，所有需要的变量都设置好了，我们可以在片段着色器中添加光照计算了。

我们需要做的第一件事是计算光源和片段位置之间的方向向量。前面提到，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。我们能够简单地通过让两个向量相减的方式计算向量差。我们同样希望确保所有相关向量最后都转换为单位向量，所以我们把法线和最终的方向向量都进行标准化：

```
25      vec3 norm = normalize(Normal);
26      vec3 lightDir = normalize(lightPos - FragPos);
```

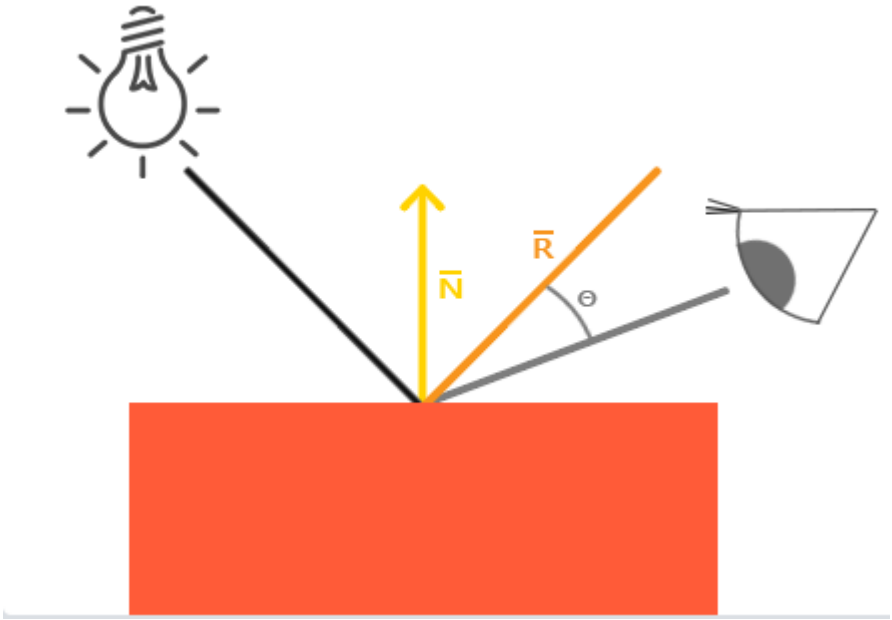
下一步，我们对norm和lightDir向量进行点乘，计算光源对当前片段实际的漫发射影响。结果值再乘以光的颜色，得到漫反射分量。两个向量之间的角度越大，漫反射分量就会越小：

```
27      float diff = max(dot(norm, lightDir), 0.0);
28      vec3 diffuse = diff * diffuseStrength * lightColor;
```

如果两个向量之间的角度大于90度，点乘的结果就会变成负数，这样会导致漫反射分量变为负数。为此，我们使用max函数返回两个参数之间较大的参数，从而保证漫反射分量不会变成负数。

镜面光照

和漫反射光照一样，镜面光照也是依据光的方向向量和物体的法向量来决定的，但是它也依赖于观察方向，例如玩家是从什么方向看着这个片段的。镜面光照是基于光的反射特性。如果我们想象物体表面像一面镜子一样，那么，无论我们从哪里去看那个表面所反射的光，镜面光照都会达到最大化。你可以从下面的图片看到效果：



我们通过反射法向量周围光的方向来计算反射向量。然后我们计算反射向量和视线方向的角度差，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，我们会看到一个高光。

观察向量是镜面光照附加的一个变量，我们可以使用观察者世界空间位置和片段的位置来计算它。之后，我们计算镜面光强度，用它乘以光源的颜色，再将它加上环境光和漫反射分量。

为了得到观察者的世界空间坐标，我们简单地使用摄像机对象的位置坐标代替（它当然就是观察者）。所以我们把另一个uniform添加到片段着色器，把相应的摄像机位置坐标传给片段着色器：

```
7   uniform vec3 viewPos;
```

```
22      // camera
23      Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
```

```
226     (*lightingShader).setVec3("viewPos", camera.Position);
```

现在我们已经获得所有需要的变量，可以计算高光强度了。首先，我们定义一个镜面强度(Specular Intensity)变量，给镜面高光一个中等亮度颜色，让它不要产生过度的影响，这个因子可以通过GUI调整。

```
30  // specular
31  // float specularStrength = 0.5;
32  vec3 viewDir = normalize(viewPos - FragPos);
33  vec3 reflectDir = reflect(-lightDir, norm);
34  float spec = pow(max(dot(viewDir, reflectDir), 0.0), specularLevel);
35  vec3 specular = specularStrength * spec * lightColor;
```

需要注意的是我们对`lightDir`向量进行了取反。`reflect`函数要求第一个向量是从光源指向片段位置的向量，但是`lightDir`当前正好相反，是从片段**指向**光源（由先前我们计算`lightDir`向量时，减法的顺序决定）。为了保证我们得到正确的`reflect`向量，我们通过对`lightDir`向量取反来获得相反的方向。第二个参数要求是一个法向量，所以我们提供的是已标准化的`norm`向量。然后计算镜面分量，我们先计算视线方向与反射方向的点乘（并确保它不是负值），然后取它的specularLevel次幂。这个幂是高光的反光度(Shininess)。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。反光度同样可以通过GUI调整。

最后将所有分量相加乘以物体的颜色：

```
37      vec3 result = (ambient + diffuse + specular) * objectColor;
38      FragColor = vec4(result, 1.0);
39  }
```

这样冯氏光照就完成了。

所有的因子，以及反光度均可通过GUI利用着色器的uniform调整：

```
190     // 2. Show a simple window that we create ourselves. We use a Begin / End pair to crea
191     {
192         ImGui::Begin("EDIT WINDOW");                          // Create a window called "
193         ImGui::Checkbox("Demo Window", &show_demo_window);      // Edit bools storing our
194         ImGui::Checkbox("Phong Shading", &isPhong);
195         ImGui::Checkbox("Move light source", &isBonus);
196         ImGui::SliderFloat("Ambient strength", &ambientStrength, 0.0f, 1.0f);
197         ImGui::SliderFloat("Diffuse strength", &diffuseStrength, 0.0f, 1.0f);
198         ImGui::SliderFloat("Specular strength", &specularStrength, 0.0f, 1.0f);
199         ImGui::SliderInt("Specular level", &specularLevel, 0, 256);
200         ImGui::End();
201     }
```

```
221     // be sure to activate shader when setting uniforms/drawing objects
222     (*lightingShader).use();
223     (*lightingShader).setVec3("objectColor", 1.0f, 0.5f, 0.31f);
224     (*lightingShader).setVec3("lightColor", 1.0f, 1.0f, 1.0f);
225     (*lightingShader).setVec3("lightPos", lightPos);
226     (*lightingShader).setVec3("viewPos", camera.Position);
227
228     (*lightingShader).setFloat("ambientStrength", ambientStrength);
229     (*lightingShader).setFloat("diffuseStrength", diffuseStrength);
230     (*lightingShader).setFloat("specularStrength", specularStrength);
231     (*lightingShader).setInt("specularLevel", specularLevel);
```

同样地可以通过GUI选择是想要Phong Shading 还是 Gouraud Shading ，两者的不同之处在于Phong Shading是在片段着色器上实现冯氏光照模型，而Gouraud Shading是在顶点着色器上实现冯氏光照模型在顶点着色器中做光照的优势是，相比片段来说，顶点要少得多，因此会更高效，所以（开销大的）光照计算频率会更低。然而，顶点着色器中的最终颜色值是仅仅只是那个顶点的颜色值，片段的颜色值是由插值光照颜色所得来的。结果就是这种光照看起来不会非常真实，除非使用了大量顶点。

```
208     if (isPhong == true) {
209         lightingShader = new Shader("h6shader1.vs", "h6shader1.fs");
210     }
211     else {
212         lightingShader = new Shader("h6shader3.vs", "h6shader3.fs");
213     }
```

要使得光源在场景中来回移动只要将光源的位置设置为时间的函数，这里用的是sin函数

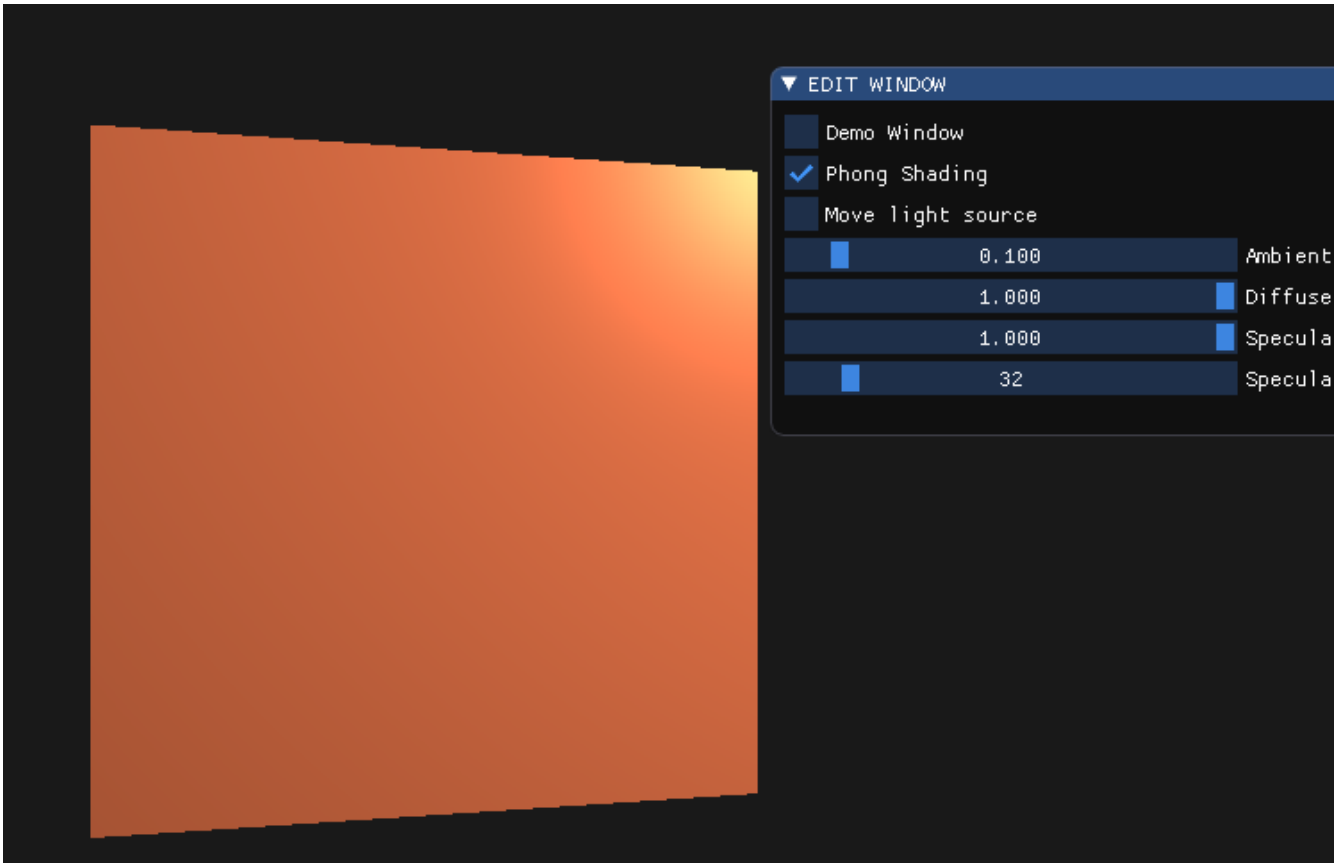
```
215         if (isBonus) {
216             // change the light's position values over time (can be done anywhere in the rend
217             lightPos.x = 1.0f + sin(glfwGetTime()) * 2.0f;
218             lightPos.y = sin(glfwGetTime() / 2.0f) * 1.0f;
219         }
```

由于通过鼠标移动视角需要捕获鼠标，这样就无法通过GUI调整参数了，这里我将视角的调整改成用上下左右键进行调整：

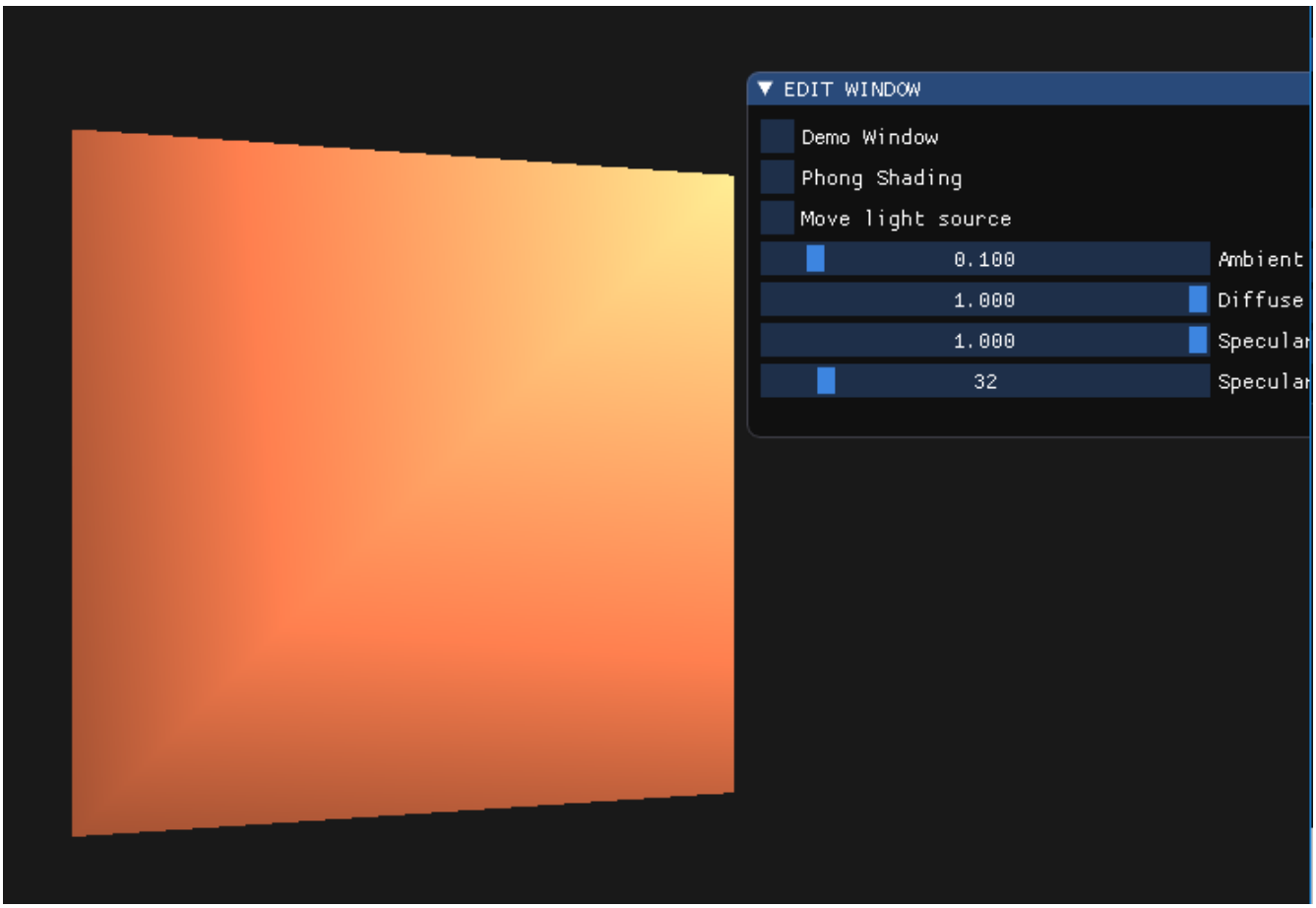
```
281   // process all input: query GLFW whether relevant keys are
282  void processInput(GLFWwindow *window)
283  {
284      if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
285          glfwSetWindowShouldClose(window, true);
286
287      if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
288          camera.ProcessKeyboard(FORWARD, deltaTime);
289      if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
290          camera.ProcessKeyboard(BACKWARD, deltaTime);
291      if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
292          camera.ProcessKeyboard(LEFT, deltaTime);
293      if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
294          camera.ProcessKeyboard(RIGHT, deltaTime);
295
296      if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
297          float velocity = -speed * deltaTime;
298          camera.ProcessMouseMovement(velocity, 0);
299      }
300      if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS) {
301          float velocity = speed * deltaTime;
302          camera.ProcessMouseMovement(velocity, 0);
303      }
304      if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS) {
305          float velocity = -speed * deltaTime;
306          camera.ProcessMouseMovement(0, velocity);
307      }
308      if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
309          float velocity = speed * deltaTime;
310          camera.ProcessMouseMovement(0, velocity);
311      }
312
313  }
```

实验结果

Phong Shading:



Gouraud Shading:



在这里可以明显的看到由于插值造成的影响。这个“条纹”是可见的，因为片段插值。从图像中，我们可以看到立方体正面的右上角被高光照亮。由于右下角三角形的右上顶点是亮的，而三角形的其他两个顶点不是亮的，所以亮值内插到其他两个顶点。左上角的三角形也是如此。由于中间片段的颜色不是直接来自光源，而是插值的结果，中间片段的光照是不正确的，左上角和右下角的三角形在亮度上发生碰撞，导致两个三角形之间出现可见的条纹。当使用更复杂的形状时，这种效果会更加明显。

移动的光源：