Homework 6 - Lights and Shading

Introduction

本次作业要求大家自己编写 shader，实现 Phong 光照模型，在场景中添加局部光照(Phong Shading/Gouraud Shading)。同时，要求调节不同的参数，观察不同的光照效果。

Homework

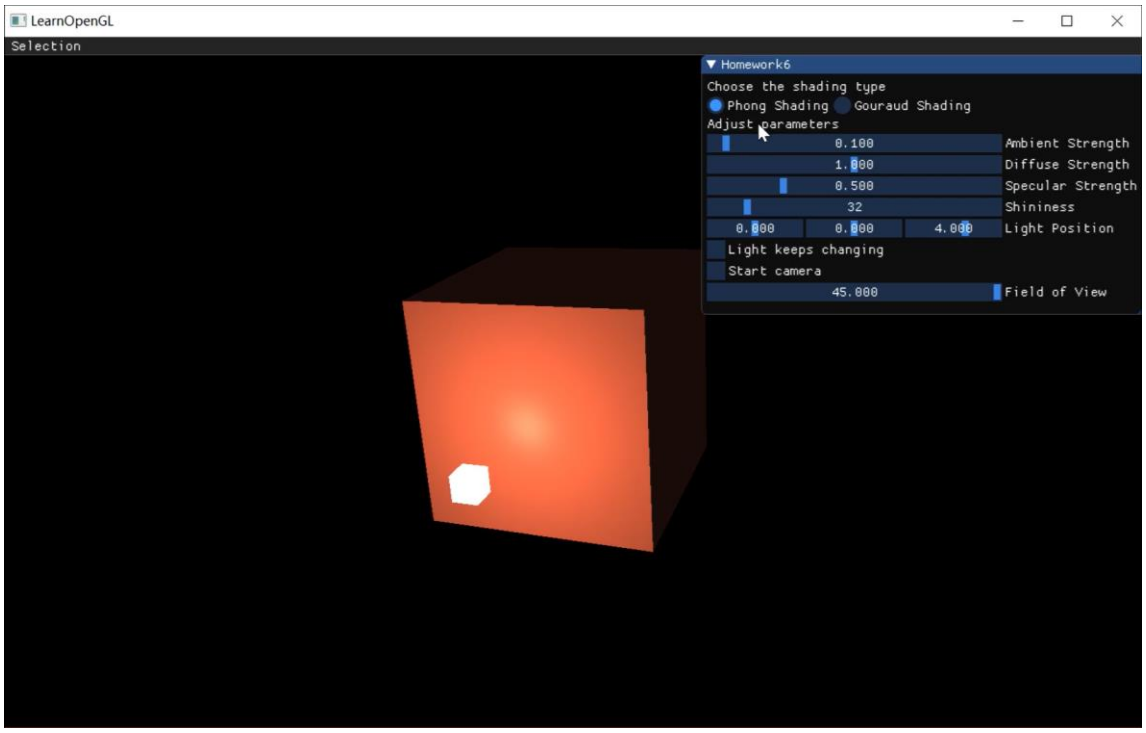
最终效果见 gif 文件

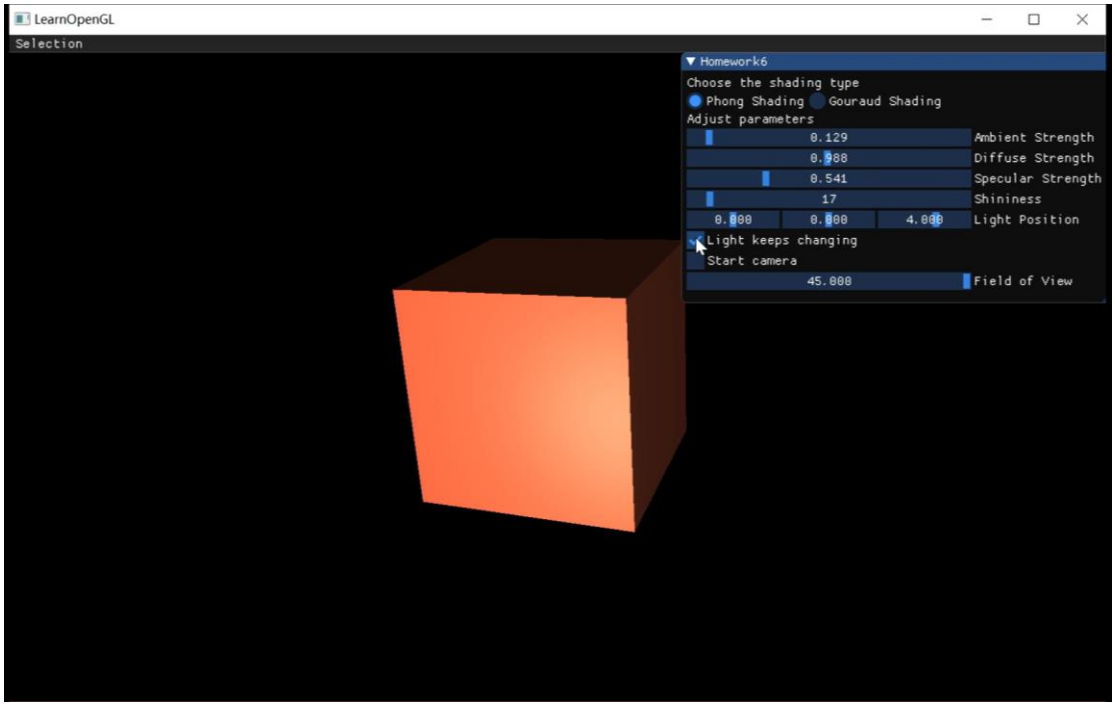
Basic:

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

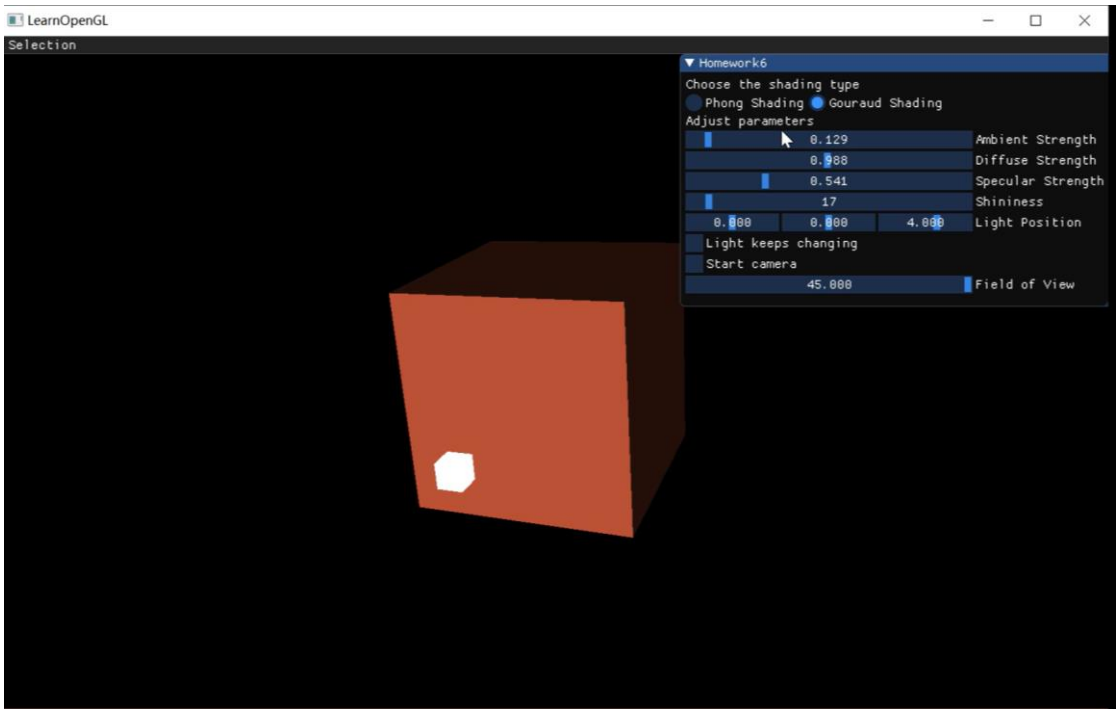
实现结果如图

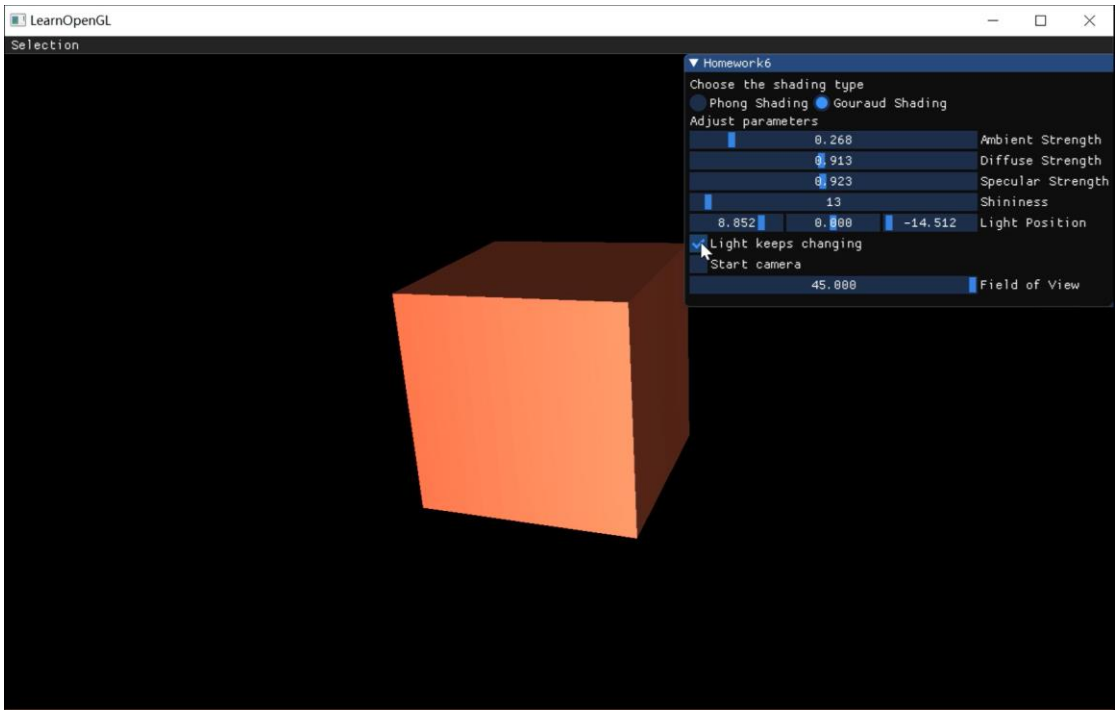
Phong Shading





Gouraud Shading





实现过程

冯氏光照模型(Phong Lighting Model)

主要结构由 3 个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照

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。使用一个环境光照常量来模拟这种情况。

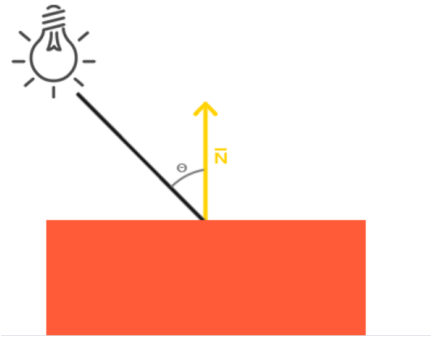
把环境光照添加到场景里非常简单。我们用光的颜色乘以一个很小的常量环境因子，再乘以物体的颜色，然后将最终结果作为片段的颜色：

```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

漫反射光照(Diffuse Lighting)

漫反射光照模拟光源对物体的方向性影响(Directional Impact)。物体的某一部分越是正

对着光源，它就会越亮。



光线与接触到的物体片段之间有一个角度。如果光线垂直于物体表面，这束光对物体的影响会最大化，即更亮。使用**法向量(Normal Vector)** 来测量光线和片段的角度，它是垂直于片段表面的一个向量（图中黄色箭头）。这两个向量之间的角度可以通过点乘计算出来。尤其注意这里计算两个向量夹角的余弦值，使用的是单位向量，因此需要确保所有的向量都是标准化的。

所以计算漫反射光照需要

- 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(Normal Vector)

法向量是一个垂直于顶点表面的（单位）向量。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。

更新光照的顶点着色器：

```
    layout (location = 1) in vec3 aNormal;
```

将法向量由顶点着色器传递到片段着色器

```
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Normal = aNormal;
}
```

在片段着色器中定义相应的输入变量：

```
in vec3 Normal;
```

计算漫反射光照

在片段着色器中声明光源的位置向量，这个值会在渲染时传入着色器程序

```
uniform vec3 lightPos;
```

在顶点着色器中计算片段的位置向量，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。

在顶点着色器中添加下面代码，其中，为了避免法向量错误缩放的影响，使用 inverse 和 transpose 函数生成一个为法向量专门定制的模型矩阵——法线矩阵

```
out vec3 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;
}
```

在片段着色器中添加相应的输入变量

```
in vec3 FragPos;
```

在片段着色器中添加光照计算

首先把法线和最终的方向向量都进行标准化

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

下一步，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和漫反射因子，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), 0.0);
vec3 diffuse = diffuseStrength * diff * lightColor;
```

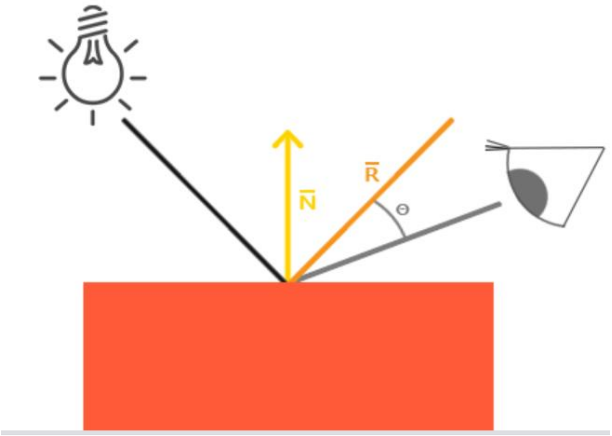
环境光分量和漫反射分量相加，把结果乘以物体的颜色，得到片段最后的输出颜色

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

镜面光照(Specular Lighting)

镜面光照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。

镜面光照效果如图



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，会看到一个高光。

计算镜面光照附加的变量，首先需要观察者世界空间位置，在片段着色器中声明

```
uniform vec3 viewPos;
```

下一步计算视线方向向量，和对应的沿着法线轴的反射向量

```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
```

接着计算镜面分量

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

其中 specularStrength 是镜面强度变量

最后把它加到环境光分量和漫反射分量里，再用结果乘以物体的颜色

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

最终顶点着色器如下

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

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec3 FragPos;
out vec3 Normal;
```

```
void main()
{
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   FragPos = vec3(model * vec4(aPos, 1.0));
   Normal = mat3(transpose(inverse(model))) * aNormal;
}
```

片段着色器如下

```
#version 330 core
out vec4 FragColor;
in vec3 Normal;
in vec3 FragPos;

uniform float ambientStrength;// 环境光强度
uniform float diffuseStrength; // 漫反射强度
uniform float specularStrength;// 镜面反射强度
uniform int shininess; //反光度
uniform vec3 lightPos;
uniform vec3 lightColor; // 光照颜色
uniform vec3 objectColor;// 物体颜色
uniform vec3 viewPos;

void main()
{
    vec3 ambient = ambientStrength * lightColor;

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

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

    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = specularStrength * spec * lightColor;

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

绘制立方体

创建顶点数组，生成并绑定立方体的顶点数组对象 cubeVAO，再生成顶点数据缓冲对

象 VBO 并绑定。设置顶点属性，包括位置和法向量

```
    //立方体
    unsigned int VBO, cubeVAO;
    glGenVertexArrays(1, &cubeVAO);
    glGenBuffers(1, &VBO);
    glBindVertexArray(cubeVAO);
    //VBO
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    //把之前定义的顶点数据复制到缓冲的内存
    glBufferData(GL_ARRAY_BUFFER,        sizeof(vertices),        vertices,
GL_STATIC_DRAW);
    //位置
    glVertexAttribPointer(0,  3,  GL_FLOAT,  GL_FALSE,  6  *  sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);
    //法向量
    glVertexAttribPointer(1,  3,  GL_FLOAT,  GL_FALSE,  6  *  sizeof(float),
(void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);
```

在着色器程序中，设置一些 uniform 变量的值，绑定 cubeVAO 进行绘制

```
    //绘制立方体
    ourShader.use();

    ourShader.setMat4("projection", projection);
    ourShader.setMat4("model", model);
    ourShader.setMat4("view", view);

    ourShader.setFloat("ambientStrength", ambientStrength);
    ourShader.setFloat("diffuseStrength", diffuseStrength);
    ourShader.setFloat("specularStrength", specularStrength);
    ourShader.setInt("shininess", shininess);

    ourShader.setVec3("lightPos", lightPos);
    ourShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
    ourShader.setVec3("objectColor", 1.0f, 0.4f, 0.3f);
    ourShader.setVec3("viewPos", camera.GetPostion());

    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
```

绘制光源

光源使用之前立方体的顶点数组，创建顶点数组对象 lightVAO，此外只需要绑定 VBO 不用

再次设置 VBO 的数据，立方体的 VBO 数据中已经包含了正确的顶点数据。

```
    //立方体
    unsigned int VBO, cubeVAO;
    glGenVertexArrays(1, &cubeVAO);
    glGenBuffers(1, &VBO);
    glBindVertexArray(cubeVAO);
    //VBO
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    //把之前定义的顶点数据复制到缓冲的内存
    glBufferData(GL_ARRAY_BUFFER,        sizeof(vertices),        vertices,
GL_STATIC_DRAW);
```

光源的着色器只需使用之前作业的版本

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

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

光源的片段着色器定义了一个不变的常量白色，保证了光源的颜色一直是亮的

```
    //绘制光源
    lightShader.use();

    model = glm::mat4(1.0f);
    model = glm::translate(model, lightPos);
    model = glm::scale(model, glm::vec3(0.1f));

    lightShader.setMat4("model", model);
    lightShader.setMat4("view", view);
    lightShader.setMat4("projection", projection);

    glBindVertexArray(lightVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
```

Gouraud 着色

两种模型的区别就是光照计算是在顶点着色器还是片段着色器中完成。Gouraud 着色只需

要把计算光照的部分在顶点着色器中完成即可。

顶点着色器

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

out vec3 LightingColor;

uniform float ambientStrength;// 环境光强度
uniform float diffuseStrength; // 漫反射强度
uniform float specularStrength;// 镜面反射强度
uniform int shininess; //反光度

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    vec3 Position = vec3(model * vec4(aPos, 1.0));
    vec3 Normal = mat3(transpose(inverse(model))) * aNormal;

    vec3 ambient = ambientStrength * lightColor;

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - Position);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diffuseStrength * diff * lightColor;

    vec3 viewDir = normalize(viewPos - Position);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = specularStrength * spec * lightColor;

    LightingColor = ambient + diffuse + specular;
}
```

片段着色器

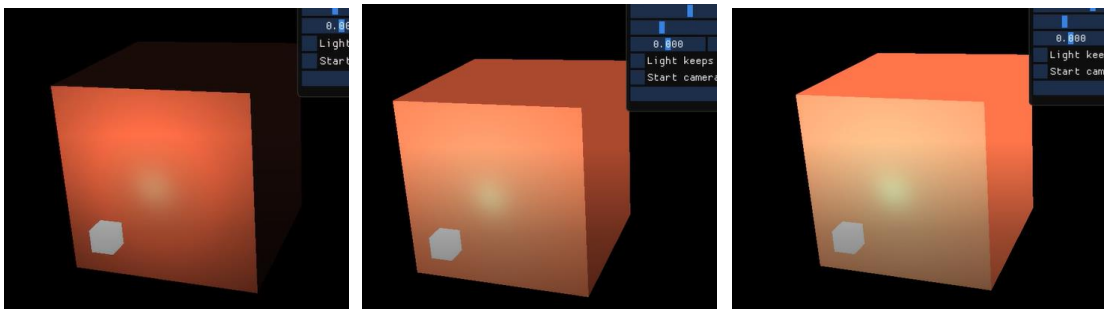
```
#version 330 core
out vec4 FragColor;

in vec3 LightingColor;

uniform vec3 objectColor;

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

2. 使用 GUI，使参数可调节，效果实时更改：

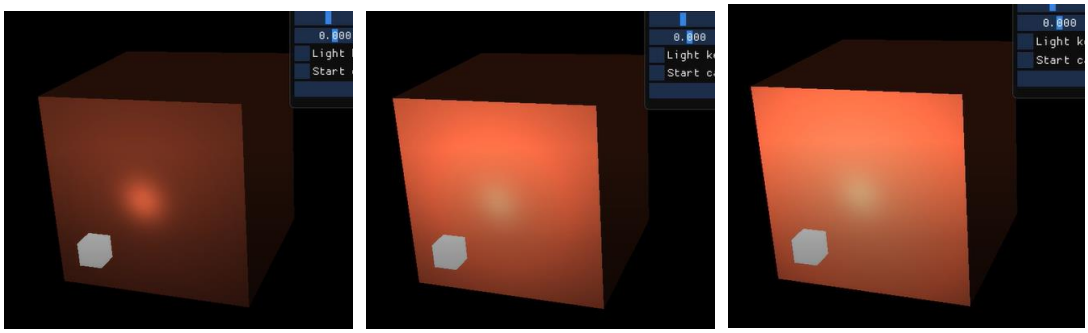
- GUI 里可以切换两种 shading
- 使用如进度条这样的控件，使 ambient 因子、diffuse 因子、specular 因子、反光度等参数可调节，光照效果实时更改

实现效果如图

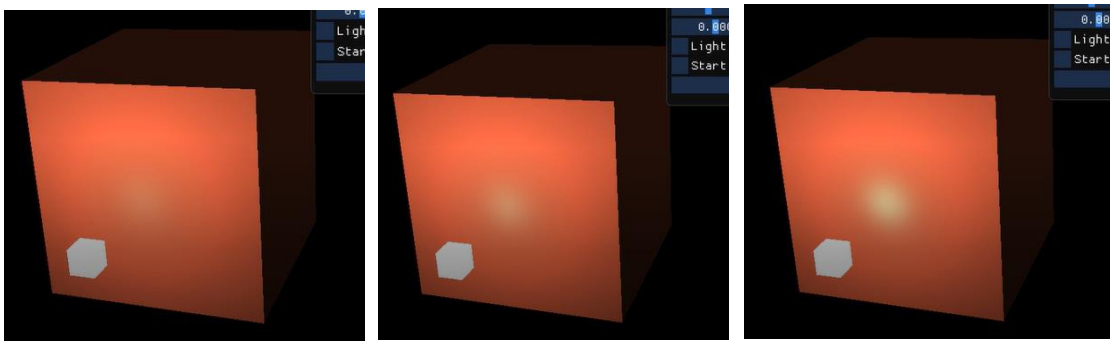
从左到右 ambient 因子越大立方体越亮



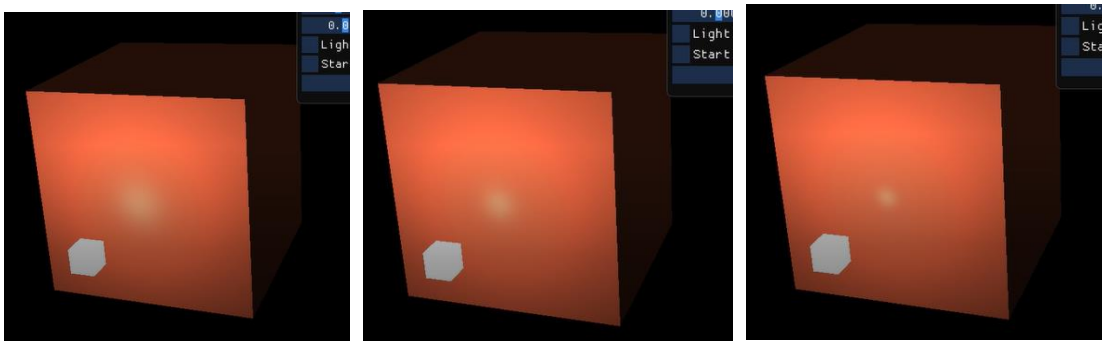
从左到右 diffuse 因子越大亮的范围越大



从左到右 specular 因子越大，高光点越亮



从左到右反光度越高，高光点越小



GUI 设置如下

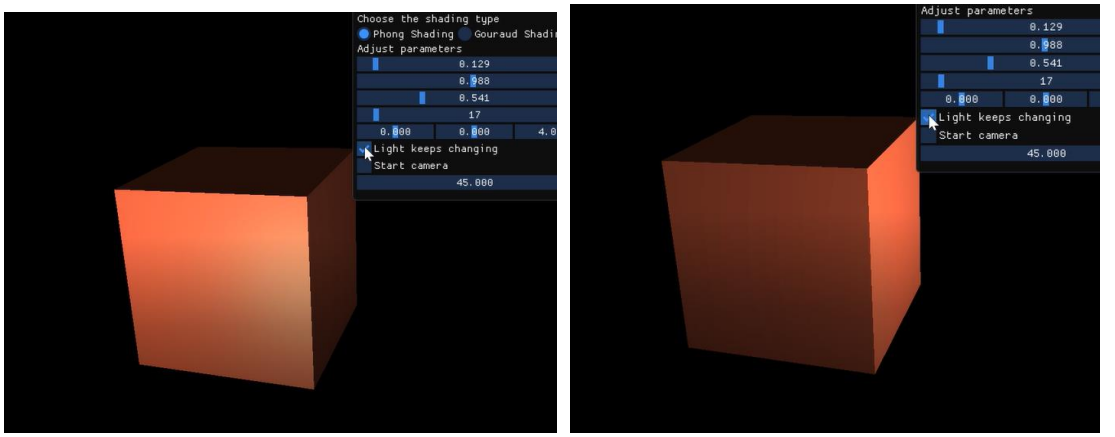
```
        ImGui::Begin("Homework6", &is_run);

        ImGui::Text("Choose the shading type");
        ImGui::RadioButton("Phong Shading", &e, 0); ImGui::SameLine();
        ImGui::RadioButton("Gouraud Shading", &e, 1);

        ImGui::Text("Adjust parameters");
       ImGui::SliderFloat("Ambient Strength", &ambientStrength, 0.0f, 2.0f);
       ImGui::SliderFloat("Diffuse Strength", &diffuseStrength, 0.0f, 2.0f);
        ImGui::SliderFloat("Specular  Strength",  &specularStrength,  0.0f,
2.0f);
        ImGui::SliderInt("Shininess", &shininess, 0, 256);

        ImGui::SliderFloat3("Light Position", lpos, -15.0f, 15.0f);
        ImGui::Checkbox("Light keeps changing", &change);
        ImGui::Checkbox("Start camera", &start_camera);
        ImGui::SliderFloat("Field of View", &fov, 0.0f, 45.0f);
        ImGui::End();
```

Bonus:

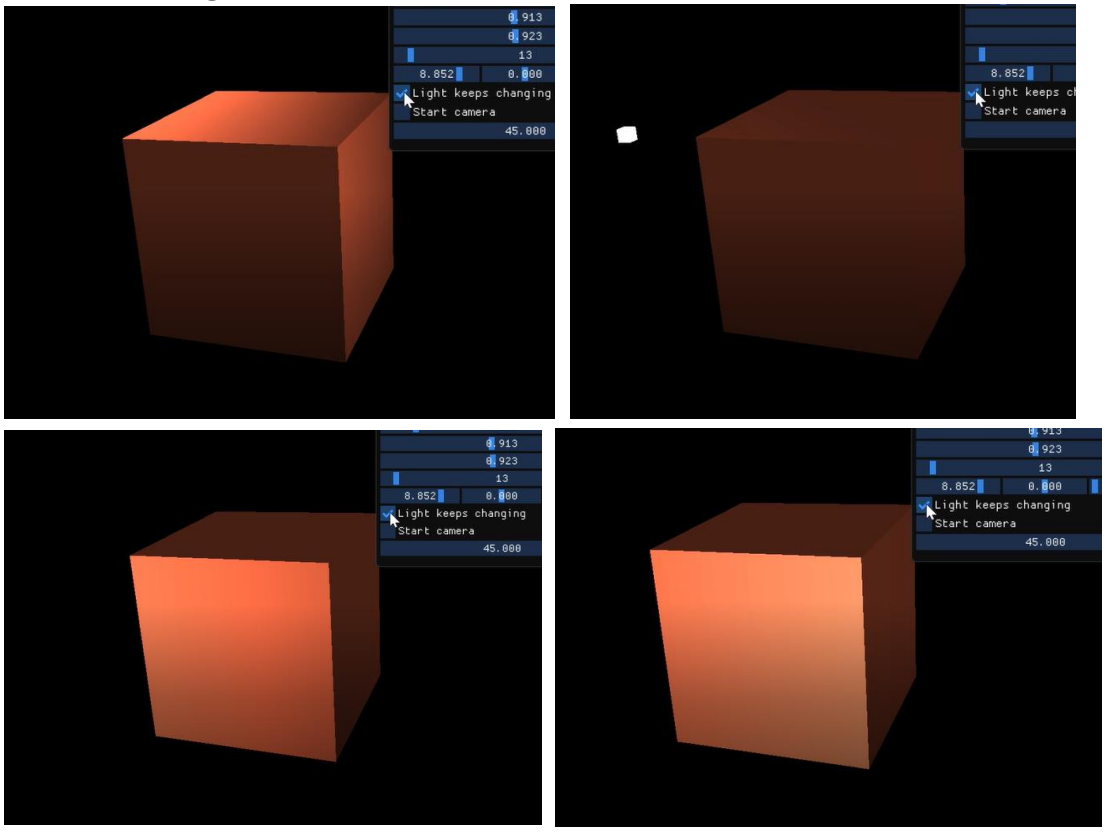
当前光源为静止状态，尝试使光源在场景中来回移动，光照效果实时更改。

实现效果如图

Phong Shading



Gouraud Shading



使光源移动只需更改光源坐标位置 lightPos 的值，再传入着色器程序中

代码如下

```
    if (change) {
            float raduis = 10.0f;
            float x = sin((float)glfwGetTime()) * raduis;
            float z = cos((float)glfwGetTime()) * raduis;

            lightPos = glm::vec3(x, lpos[1], z);
        }

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