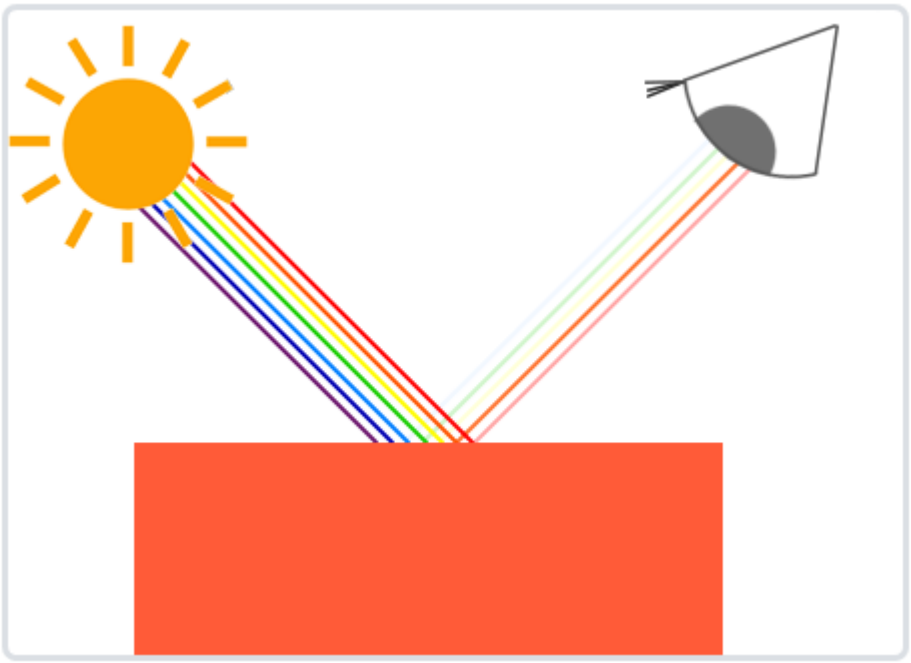
计算机图形学作业（五）：画一个立方体并实现 Phong Shading 和 Gouraud Shading 两种阴影

颜色和光照场景

我们在现实生活中看到某一物体的颜色并不是这个物体真正拥有的颜色，而是它所反射的颜色。换句话说，那些不能被物体所吸收的颜色（被拒绝的颜色）就是我们能够感知到的物体的颜色。例如，太阳光能被看见的白光其实是由许多不同的颜色组合而成的（如下图所示）。如果我们将白光照在一个蓝色的玩具上，这个蓝色的玩具会吸收白光中除了蓝色以外的所有子颜色，不被吸收的蓝色光被反射到我们的眼中，让这个玩具看起来是蓝色的。下图显示的是一个珊瑚红的玩具，它以不同强度反射了多个颜色。



所以，在 OpenGL 中，我们要创建一个自然光源和一个物体，那么要计算物体的反射颜色，就将自然光源和物体这两个颜色的向量作分量相乘，如下：

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

如果自然光源不是白光，而是其它颜色光，那么物体反射出来的颜色也会作相应的改变。

为了模拟 OpenGL 中的颜色、阴影，我们要创建一个光照场景，包括一个自然光源和一个正方体，为我们后面的工作作准备。为了不产生混淆，自然光源和物体使用不同的VAO。结果如下：

Phong Shading

在 OpenGL 中，常用的一个光照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

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，再乘以物体的颜色，然后将最终结果作为片段的颜色，片段着色器代码如下：

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

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

漫反射光照

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。为了实现这一点，我们利用自然光源发出的光线的向量与物体表面法向量形成的夹角，计算物体表面的漫反射光照。

法向量是一个垂直于顶点表面的向量，我们通过顶点着色器直接传入立方体六个面对应的法向量的值，顶点数据可以在官方资源中找到。要注意的是，当物体进行了放缩、旋转等操作时，我们在顶点着色器传入的法向量值就变得不准确了，所以我们要在顶点着色器对法向量进行处理，以保证准确性，如下：

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

最后，在片段着色器重，就可以进行漫反射光照的计算了：

```
//漫反射
 vec3 norm = normalize(Normal);
 vec3 lightDir = normalize(lightPos - FragPos);
 float diff = max(dot(norm, lightDir), 0.0);
 vec3 diffuse = diffuseStrength * diff * lightColor;
```

镜面光照

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。

所以比起漫反射光照，我们需要多一个观察者的位置变量，然后便可进行镜面光照计算，如下：

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

着色器代码

顶点着色器：

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

out vec3 FragPos;
out vec3 Normal;

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

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

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

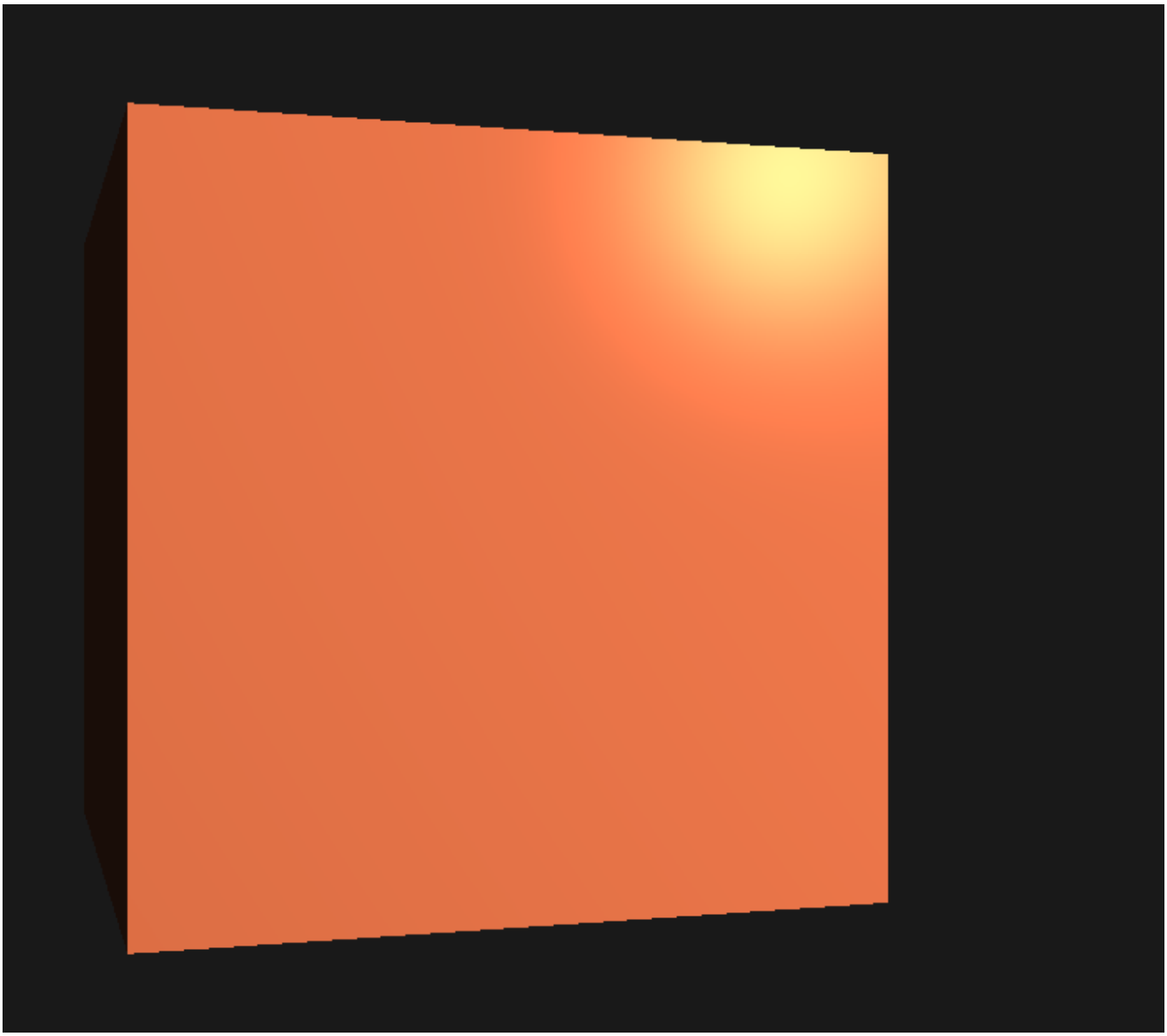
片段着色器：

```
#version 330 core

out vec4 FragColor;

in vec3 Normal;
in vec3 FragPos;

uniform float ambientStrength;
uniform float diffuseStrength;
uniform float specularStrength;
uniform int ShininessStrength;

uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;
```

```
uniform vec3 viewPos;

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

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

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

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

结果

最终，基于Phong Shading 模型的效果图如下：



Gouraud Shading

Phong Shading 是在片段着色器实现冯氏光照模型，而Gouraud Shading是在顶点着色器实现的冯氏光照模型。所以要实现 Gouraud Shading，只需要修改两个着色器的代码，把片段着色器中关于冯氏光照模型的计算移到顶点着色器即可。

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。

着色器代码

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;
uniform float diffuseStrength;
uniform float specularStrength;
uniform int ShininessStrength;

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);

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

    // 环境光
    vec3 ambient = ambientStrength * lightColor;

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

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

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

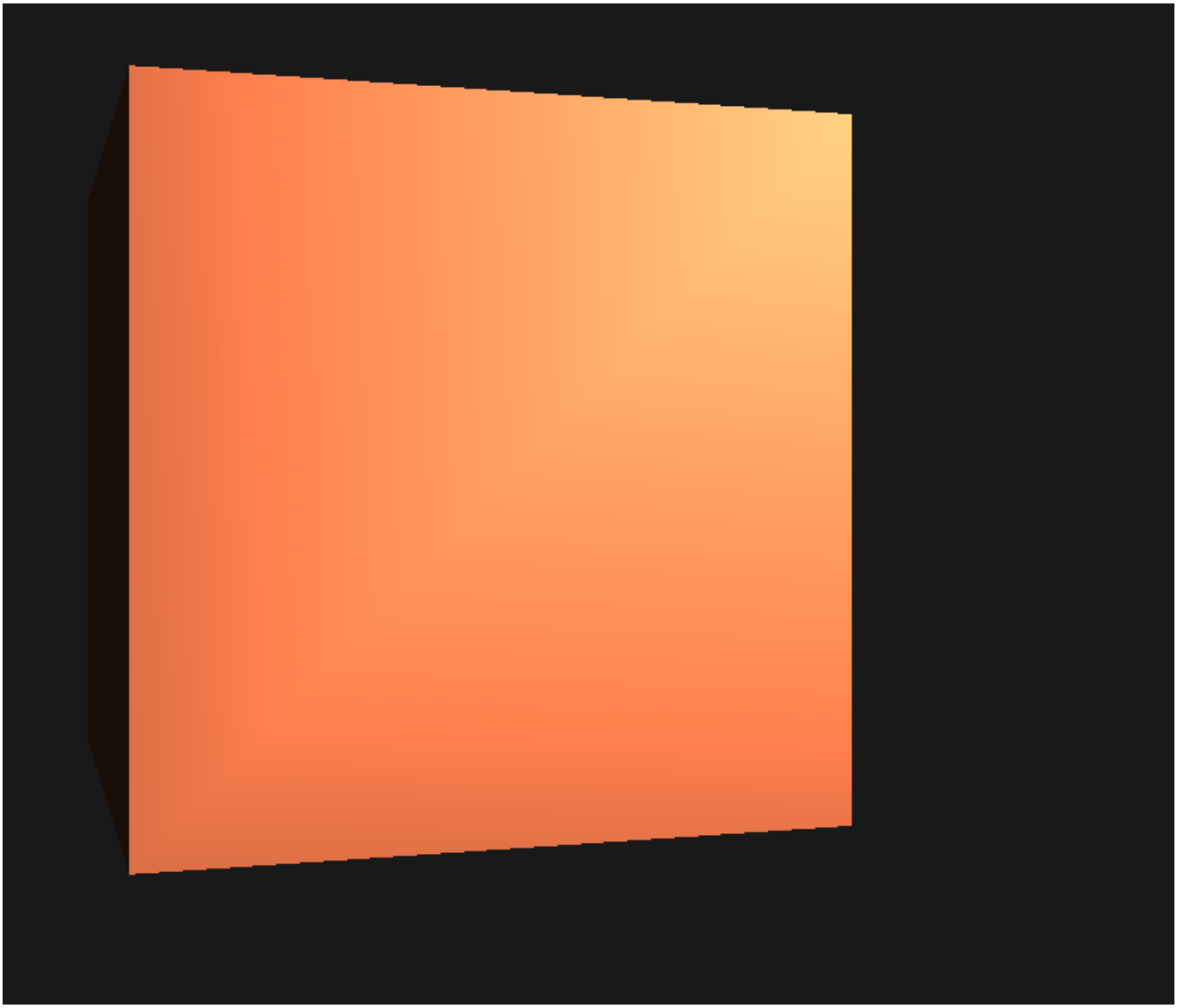
片段着色器：

```
#version 330 core
out vec4 FragColor;

in vec3 LightingColor;

uniform vec3 objectColor;

void main()
```

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

结果

最终基于 Gouraud Shading 模型的效果图如下：



光源来回移动（加分项）

要实现光源的来回移动并不难，我先定义了光源的坐标，如下：

```
glm::vec3 lightPos(20.0f, 8.0f, 20.0f);
```

然后在绘图的循环中不断控制 light.x 的增减，即可实现观察者左右来回移动，如下：

```
//灯光左右移动
if (enable_translate) {
        if (is_translate_to_left) {
                lightPos.x = lightPos.x - 0.5f;
                if (lightPos.x < -30) {
                        is_translate_to_left = false;
                }
        }
        else {
                lightPos.x = lightPos.x + 0.5f;
                if (lightPos.x > 30) {
                        is_translate_to_left = true;
                }
        }
}
```