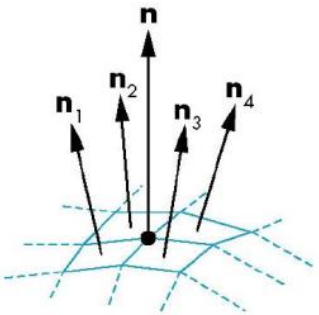
16340304 郑国林

Basic：

1. 实现 Phong shading

Phong shading 模型是一个经验模型，和 phong 光照明模型有所不同，phong 光照明模型是针对一个点的，而 phong shading 是针对每个 pixel。对于每个 pixel 使用 phong 光照明模型，不过此时对每个 pixel 用 phong 光照明模型的时候法向量是通过插值得出的。

首先求的顶点的法向量，是顶点所在多个面的法向量的均值。



$$n = \frac{n_1 + n_2 + n_3 + n_4}{\left| n_1 + n_2 + n_3 + n_4 \right|}$$

然后在对每个 pixel 使用 phong 光照明模型的时候，使用线性插值的办法，为每个 pixel 插值得到一个法向量，再使用 phong 光照明模型求得该 pixel 的光照。

Phong 光照明模型主要由下面三部分组成

a. 环境光
b. 漫反射
c. 镜面反射

a. 实现环境光

本来一个物体反射的光除了直接光源外，还会有很多其它物体反射过来的间接光源，但是如果要对这么多光进行计算的话就会复杂度很高，因此在 local 光照模型中，就采用了环境光来在一定程度上替代这些由其它物体反射过来的间接光源的效果。具体的做法就是给每一个物体的表面一个相同常量的光照，然后再乘以不同物体对应的材质因子，也就是我们程序中的 ambient 因子。

具体程序的实现的话就是在片段着色器中，首先我们要输入该物体的颜色信息以及光照信息，然后我们把光源的颜色与物体的颜色值相乘，所得到的就是这个物体所反射的颜色（也就是我们所感知到的颜色），也就是我们可以定义物体的颜色为物体从一个光源反射各个颜色分量的大小。

然后环境光的计算就是直接用 ambient 因子乘以光源的颜色信息得到的。

```
1. vec3 ambient = ambientStrength * lightColor;
```

另外如果想要动态的调整这个 ambient 因子，我们就不能在片段着色器中

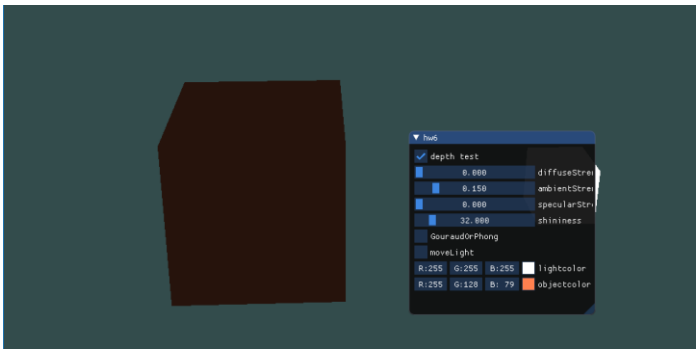
将其写死为某个固定值，而是通过

```
1.    uniform float ambientStrength;
```

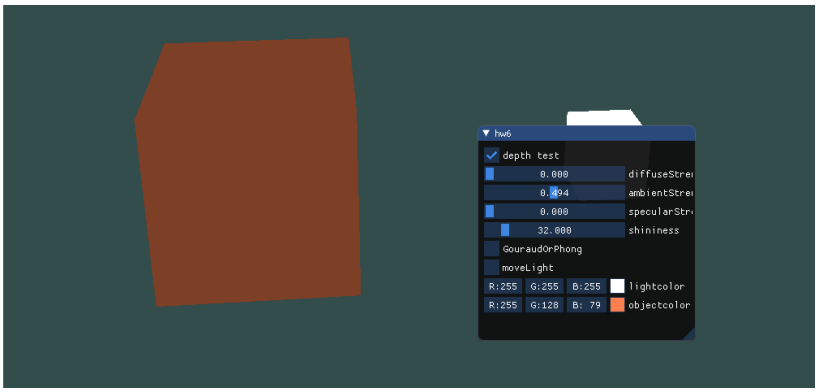
的方式来从外部输入到片段着色器中。

单独的环境光效果截图：

Ambient 因子为 0.15



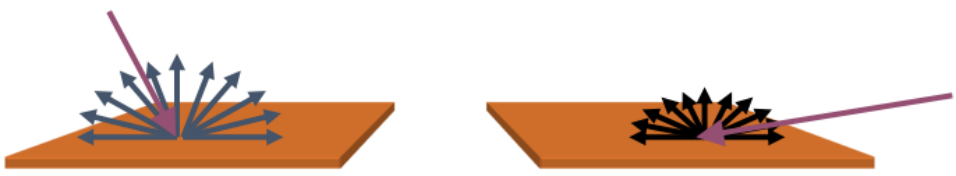
Ambient 因子为 0.5



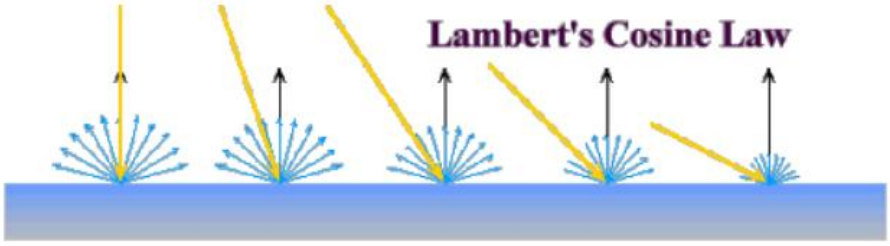
可见，ambient 因子越大，就相当于直接把光照施加在物体上。

b. 实现漫反射

在粗糙物体的表面，漫反射就是无论入射光从什么角度射入，反射光都是从所有角度射出的。所以反射光的角度是和入射光角度无关。



并且反射光的强度在不同方向都是一样的，不过随着入射光角度的不同，反射光的强度也会发生变化。根据朗伯余弦定律，漫反射的强度是和入射光与平面的法向量夹角有关的。夹角越大，发射的强度越小。

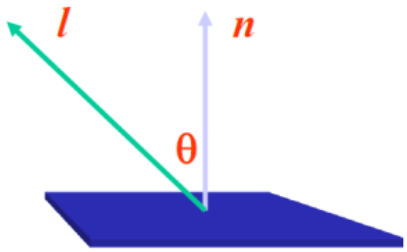


具体的计算，反射光强度就等于 diffuse 因子与入射光强度和夹角的 cos 值相乘。

$$I_{diffuse} = k_d I_{light} \cos \theta$$

在计算的时候，cos 值我们可以通过单位的入射光向量和法向量的点乘来得到。并且我们在计算的时候要注意夹角大于 90 度的情况，这种情况漫反射的光强是等于 0 的。（不能从背面穿过）

$$I_{diffuse} = k_d I_{light} (\mathbf{n} \cdot \mathbf{l})$$



代码的具体实现的话，我们首先要得到平面的法向量和入射光的向量。平面的法向量的话就直接在顶点数组中将每个顶点所在面的法向量信息加了进去，跳过了为顶点在不同面求平均法向量的过程。

```
1.  //顶点矩阵，前面是位置信息，后面是顶点对应的所在面的法向量
2.      float vertices[] = {
3.          -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
```

然后还要在顶点着色器代码中加入输入的信息，

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

以及告诉 opengl 怎么看顶点数组的数据

```
1.  //法向量信息
```

```
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);
```

可是，片段着色器里的计算都是在世界空间坐标中进行的。所以，我们要把法向量也转换为世界空间坐标。而这个转换是不能简单的通过乘以 model 矩阵进行的。

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。

因此就要计算出一个新的转换法向量到世界坐标的矩阵。这个矩阵的计算是通过将原来的 module 矩阵先求逆，然后转置，再去左上角 3x3 矩阵得到的。并且由于计算量比较大，所以也可以在顶点着色器外先将这个矩阵计算好传入顶点着色器中，避免每个点都要计算一次。

```
lightingShader.setMat3("NormalMatrix", glm::mat3(transpose(inverse(model))));
```

```
//进行法向量变换
    Normal = NormalMatrix * aNormal;
```

通过以上的操作就得到了法向量信息可以传递给片段着色器了（在片段着色器中，会自动根据顶点的法向量来对不同 pixel 进行插值得到 pixel 对应的法向量）。然后接下来就是怎么求入射光的向量表示。其实就是用光源的位置减片段的位置就行，而这个片段的位置首先要在顶点着色器中通过 module 矩阵相乘得到，传递给片段着色器进行操作。（过程中应该也会为每个 pixel 进行插值）

```
//转换到世界空间
    FragPos = vec3(model * vec4(aPos, 1.0));
```

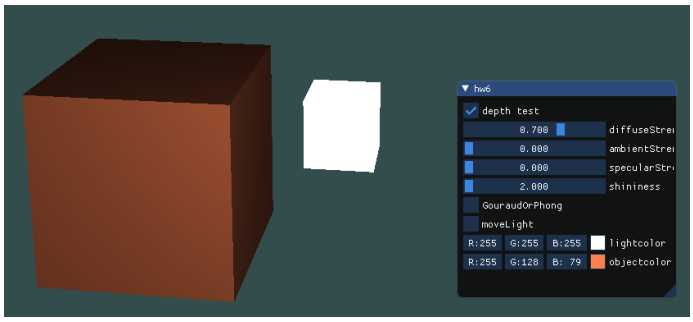
然后在片段着色器中就按上述的公式计算就行

$$I_{diffuse} = k_d I_{light} (\mathbf{n} \cdot \mathbf{l})$$

```
// diffuse
    vec3 norm = normalize(Normal);
```

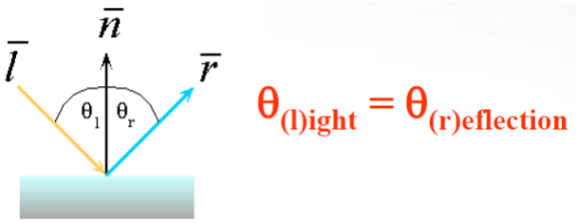
```
3.      vec3 lightDir = normalize(lightPos - FragPos);
4.      float diff = max(dot(norm, lightDir), 0.0);
5.      vec3 diffuse = diff * lightColor;
6.      diffuse = diffuseStrength * diffuse;
```

单独的漫反射效果图:

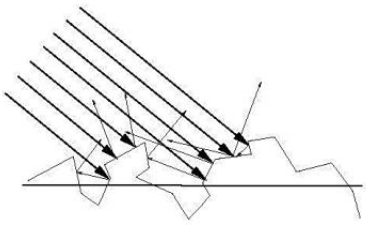


c. 实现镜面反射

镜面反射，根据 Snell's Law，就是入射角等于反射角

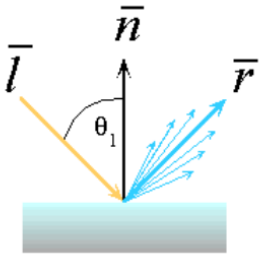


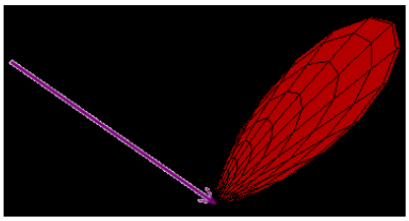
但是在细节层面上，有些我们看上去是光滑的平面还是很难理想的进行镜面反射。



所以在进行镜面反射的计算时，我们大部分的光都是按入射角等于反射角来进行计算的，但是还要考虑部分因为细节层面不够光滑导致的细微偏移。

- Angular falloff (角度下降)

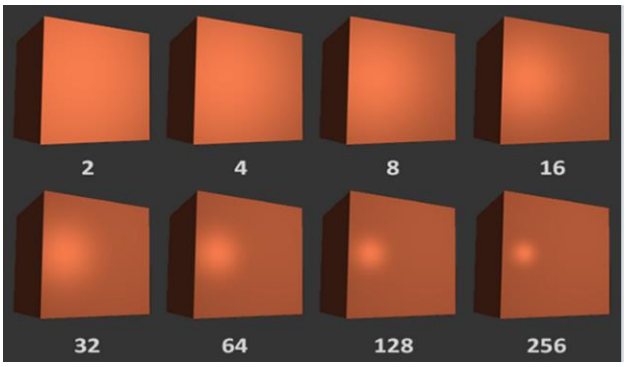




所以就得到了一个经验公式，

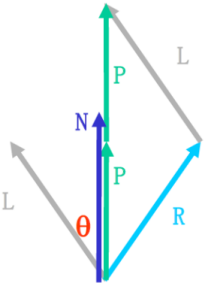
$$I_{specular} = k_s I_{light}(\bar{v} \cdot \bar{r})^{n_{shiny}}$$

其中 v 是指向观察者的向量，在代码实现的时候我们可以通过照相机位置和片段位置向量相减并做单位化得到。Ks 是 specular 因子，n 是材质发光常数，值越大，表面越接近镜面，高光面积越小。



所以我们主要要计算的其实是 r，理想的反射角度向量。可以通过以下矮到平行四边形法则进行计算。其中 P 是单位法向量，L 是单位入射光方向向量，R 是单位反射光方向向量。在 opengl 中我们可以直接调用 reflect 函数来完成这一步的工作。

$$2P = R + L$$
$$2P - L = R$$
$$2(N(N \cdot L)) - L = R$$



片段着色中的镜面反射的代码实现如下：

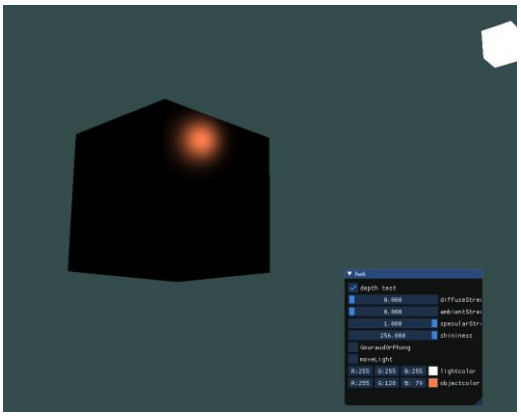
```
1.  // specular
```

```
    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;
```

如果想要在外部调节 shininess 和 specular 因子也可以通过外部传入参数到片段着色器方法实现。

```
uniform float specularStrength;
uniform float Shininess;
```

单独的镜面反射效果图:



最后 phong 模型就是要把上面三种光的效果结合在一起，我们只要在片段着色器中将上面三个结果相加并且和物体颜色信息相乘，就得到了反射的颜色。

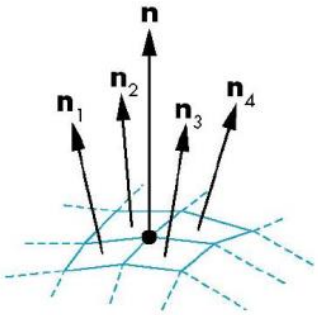
```
vec3 result = (ambient + diffuse + specular) * objectColor;

    FragColor = vec4(result, 1.0);
```

2. 实现 Gouraud shading

和 phong shading 其实差不多，不过和 phong shading 插值的内容不同，他不是针对每个每个像素级在片段着色器来进行计算光照，而是在顶点级，也就是顶点着色器中对每个顶点通过 phong 光照明模型来计算光照强度，然后再通过线性插值的方法对其中的像素光照进行计算，用于整个平面。

对于每个顶点的法向量计算，同样也是将顶点所在各个面的法向量求均值



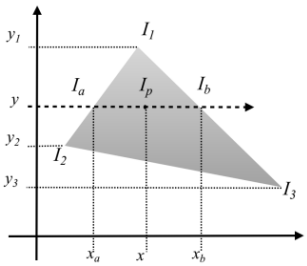
$$n = \frac{n_1 + n_2 + n_3 + n_4}{\left| n_1 + n_2 + n_3 + n_4 \right|}$$

通过 phong 光照明模型计算出顶点的光强之后对顶点内的 pixel 进行插值

$$I_a = I_1 + (I_2 - I_1)\frac{y - y_1}{y_2 - y_1}$$

$$I_b = I_1 + (I_3 - I_1)\frac{y - y_1}{y_3 - y_1}$$

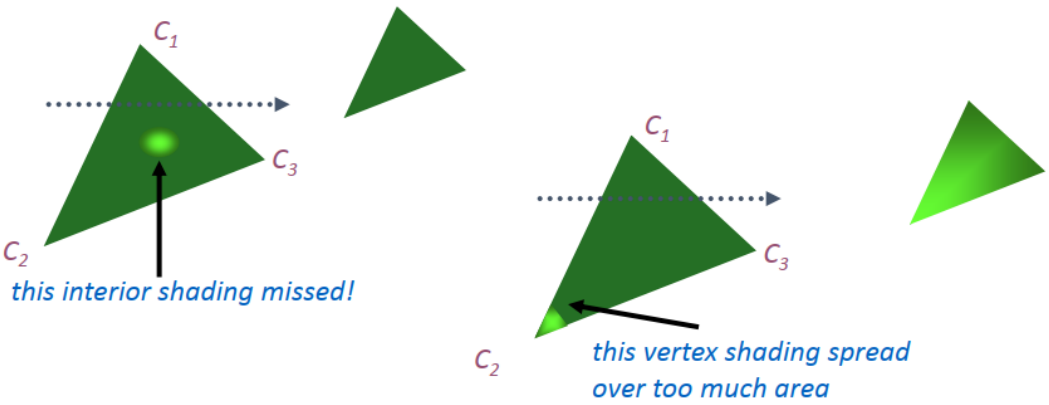
$$I_p = I_a + (I_b - I_a)\frac{x - x_a}{x_b - x_a}$$



这样做能够减少很多计算量，但是同时也会带来一些问题。

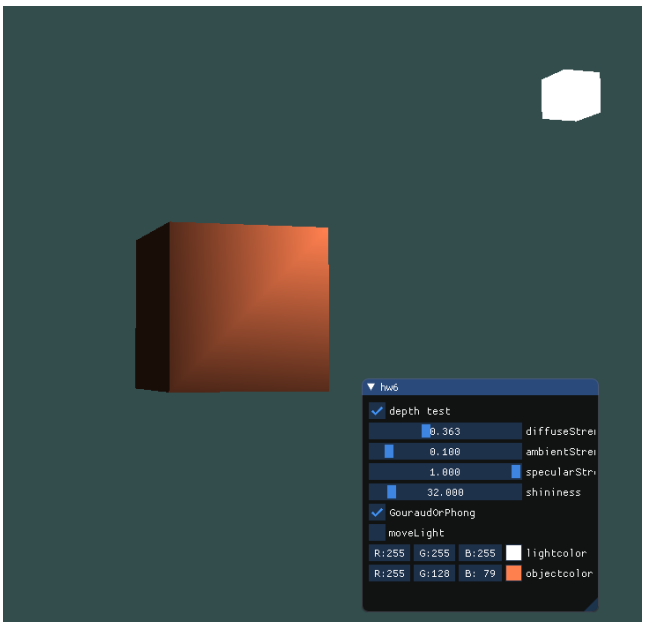
a. 镜面反射，高光的效果变差

当高光不在顶点出现的时候，进行插值是无法插值出高光的效果的，所以会丢失掉高光的这部分效果。并且如果高光发生在顶点的时候，由于插值是将顶点从多个方向进行插值，会使高光的效果作用到过多的部分。

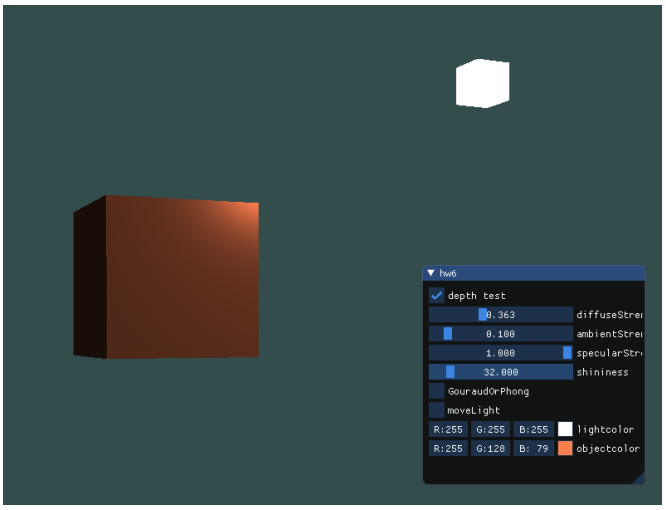


运行效果图：

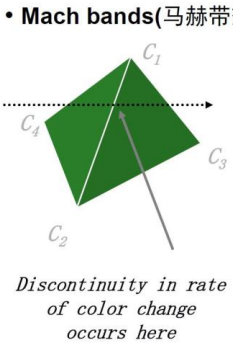
高光发生在顶点，然后出现了一条直线



而对比 phong 模型则只会出现在顶点周围



b. 另外还会出现马赫带效应，在过度的时候显得不够平滑



具体的代码的实现的话，其实只要将原先 phong shading 中在片段着色器中对 ambient，diffuse，specular 的计算移动到顶点着色器中就行了，然后将光照强度结果传给片段着色器，

（这个过程中会为不同 pixel 对不同顶点传入的光照强度进行插值，得到该 pixel 的光照强度）然后在片段着色器中乘以物体的颜色信息得到该 pixel 的反射颜色。

2. 使用 GUI，使参数可调节，效果实时更改：

a. GUI 里可以切换两种 shading

首先在 imGUI 里加一个 checkBox 判断是要使用哪一种 shading，

```
ImGui::Checkbox("GouraudOrPhong", &GouraudOrPhong);
```

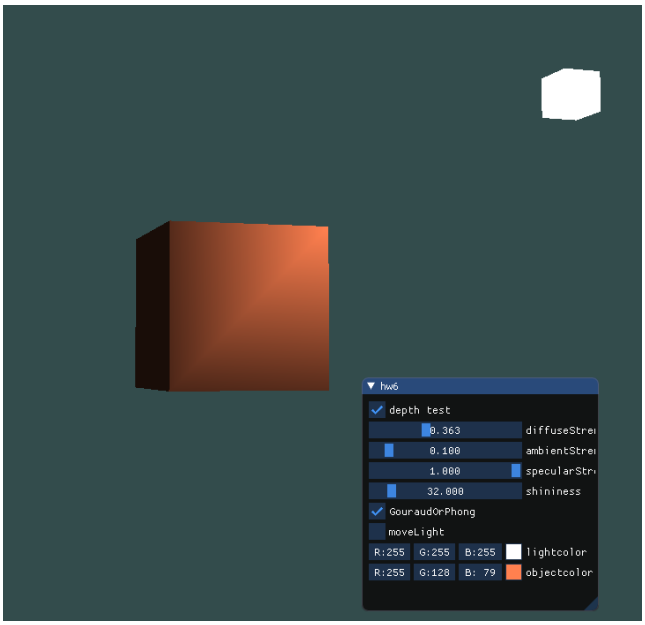
然后根据结果来对不同的 shader 源代码进行编译，（在 shader 类中实现了通过打开文件的方式来读取 shader 的源代码再进行编译，这样做能提供更为统一的接口）

```
Shader lightingShader("color1.vs", "color1.fs");
        Shader lampShader("lamp1.vs", "lamp1.fs");
        if (GouraudOrPhong)
        {
            lightingShader = Shader("GouraudColor.vs", "GouraudColor.fs");
        }
```

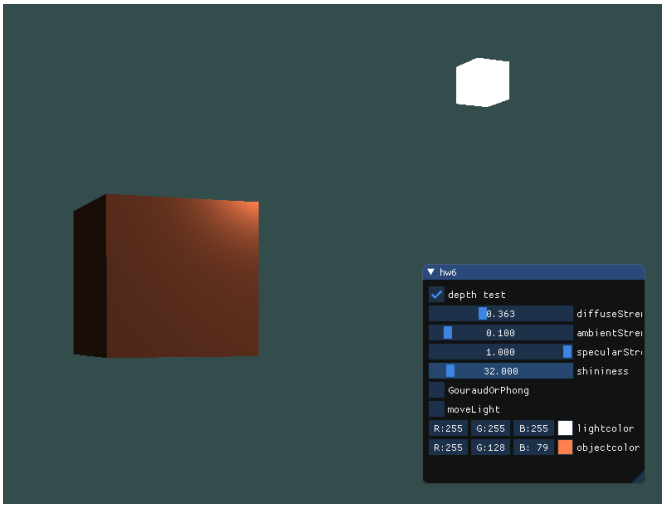
不同 shading 的对比效果图：

Gouraud

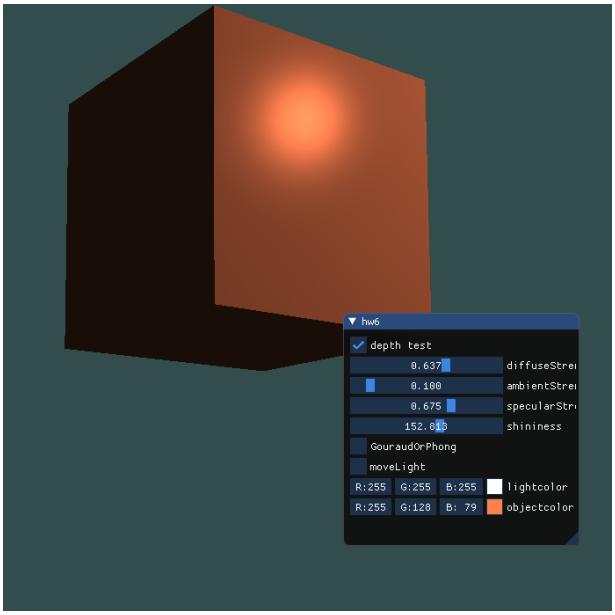
高光发生在顶点，然后出现了一条直线



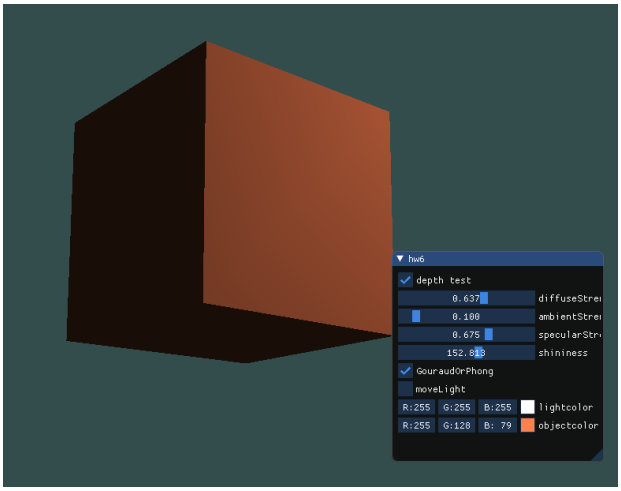
而对比 phong 模型则只会出现在顶点周围



当 phong 模型中高光出现在平面里而不是顶点上的时候:



切换成 Gouraud shading 在同样的位置就看不到了



b. 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

在 imgui 中添加 slider 来进行选择

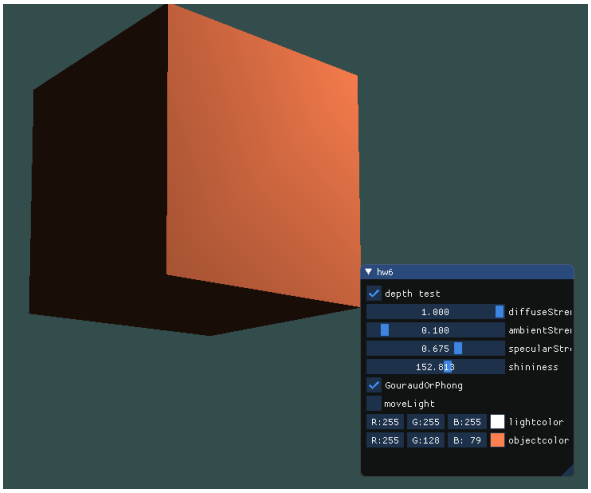
```
1.  ImGui::SliderFloat("diffuseStrength", &diffuseStrength, 0.0f, 1.0f);
```

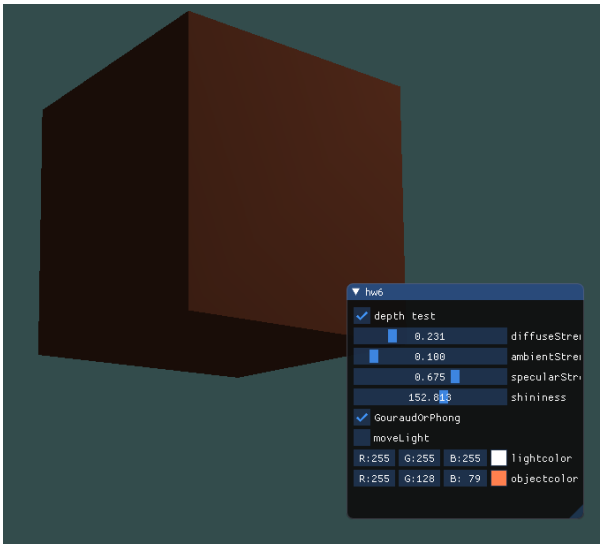
然后根据之前上面说过的方法将参数传入到顶点着色器或者片段着色器中去。

```
1.  //设置 diffuse 因子
2.          lightingShader.setFloat("diffuseStrength", diffuseStrength);
```

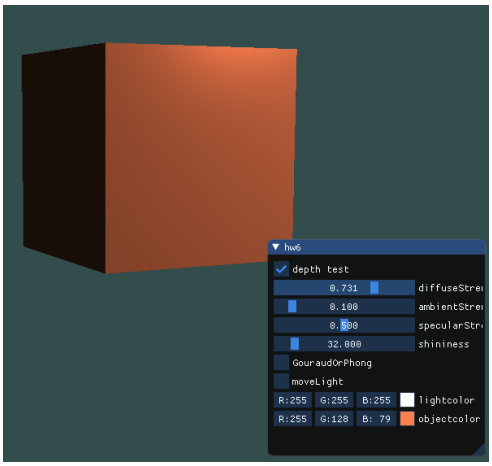
不同的因子效果对比图:

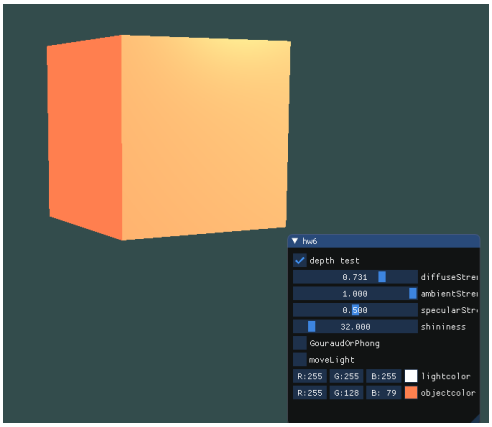
Diffuse 因子为 1 和 0.2 对比



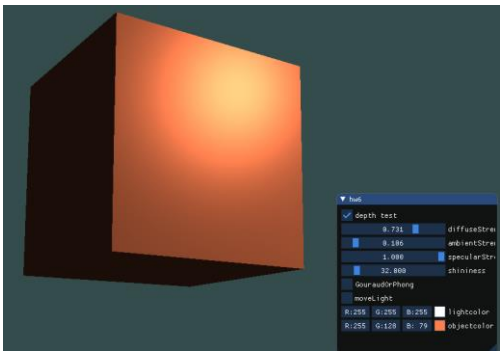


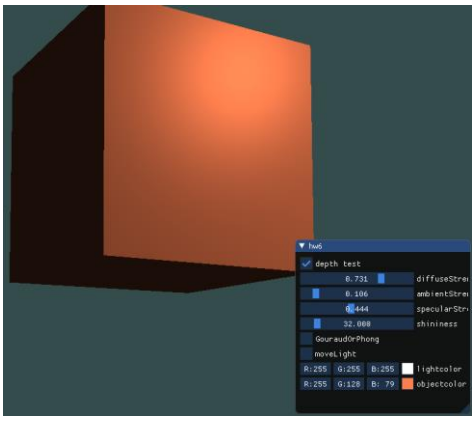
Ambient 因子为 1 和 0.1 对比



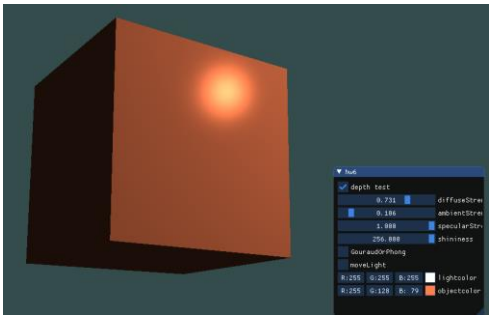


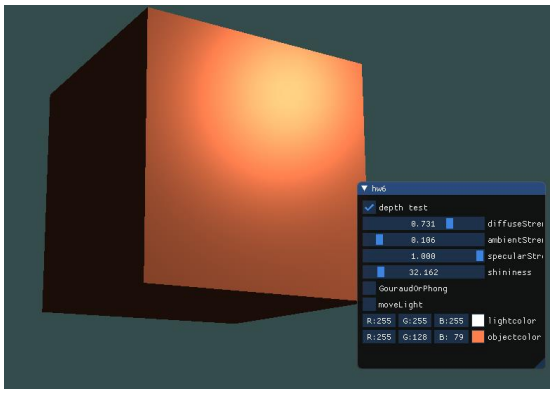
Specular 因子为 0.4 和 1 时高光效果对比



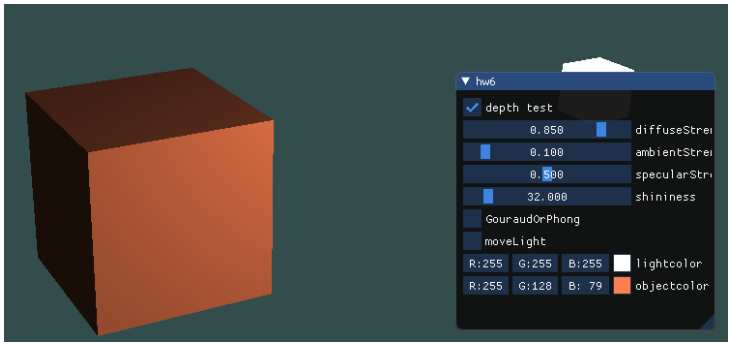


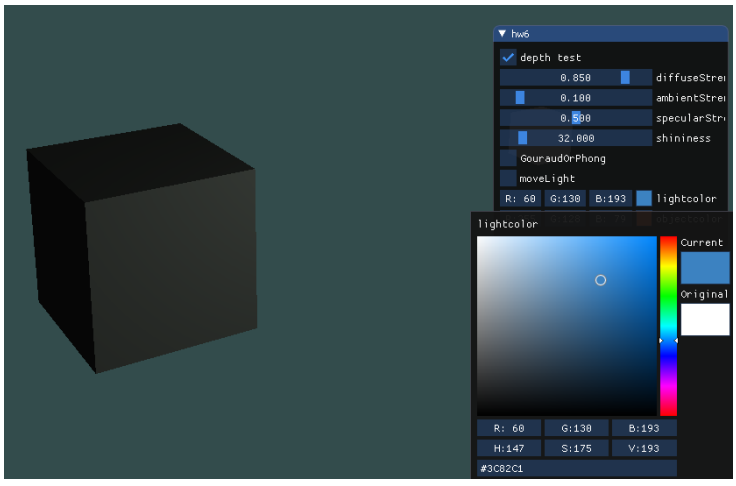
Shininess 为 32 和 256 高光效果对比

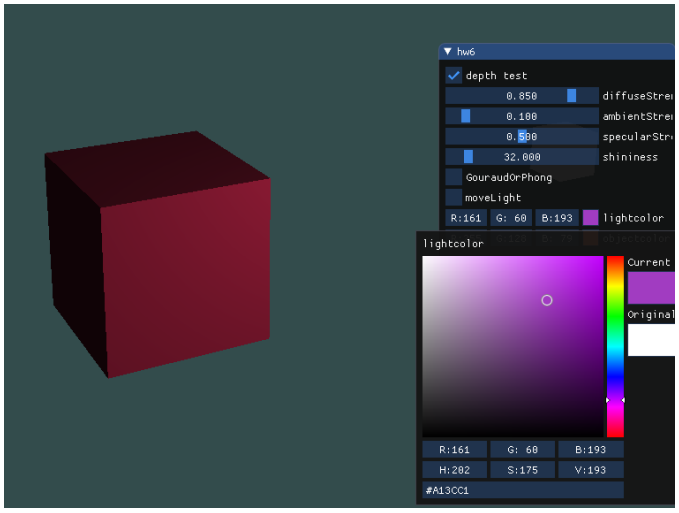




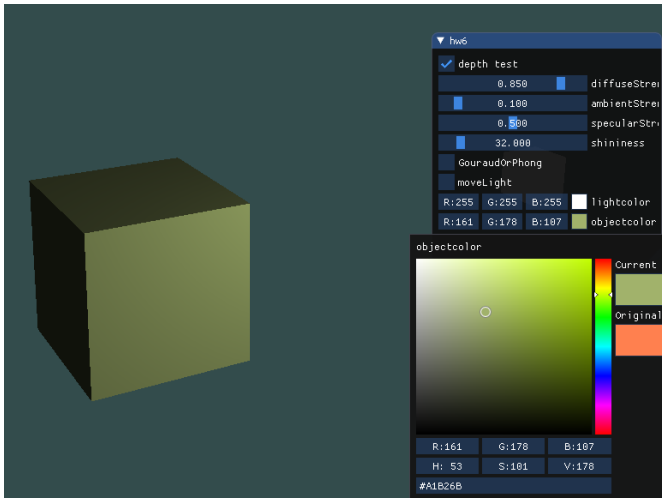
光源颜色为白色和不同颜色对珊瑚红颜色物体的照明对比

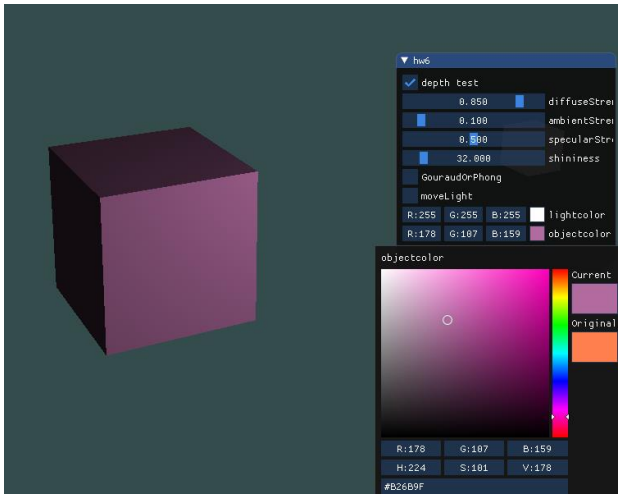






同样白色光源对不同颜色物体照明对比





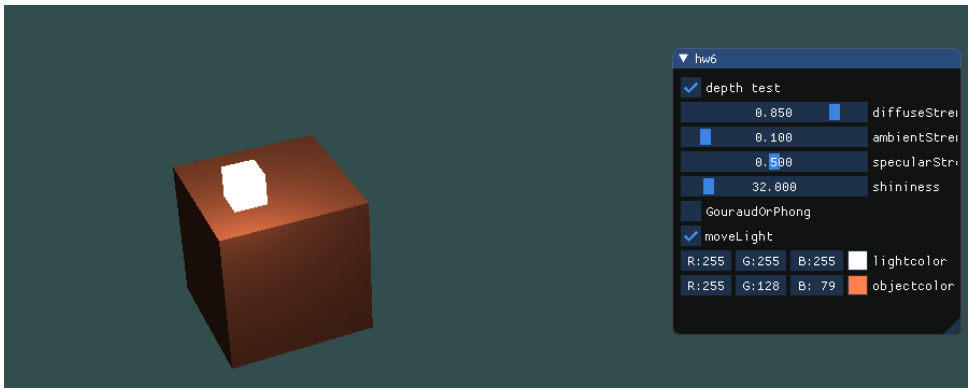
Bonus：当前光源为静止状态，尝试使光源在场景中来回移动，光照效果实时更改。

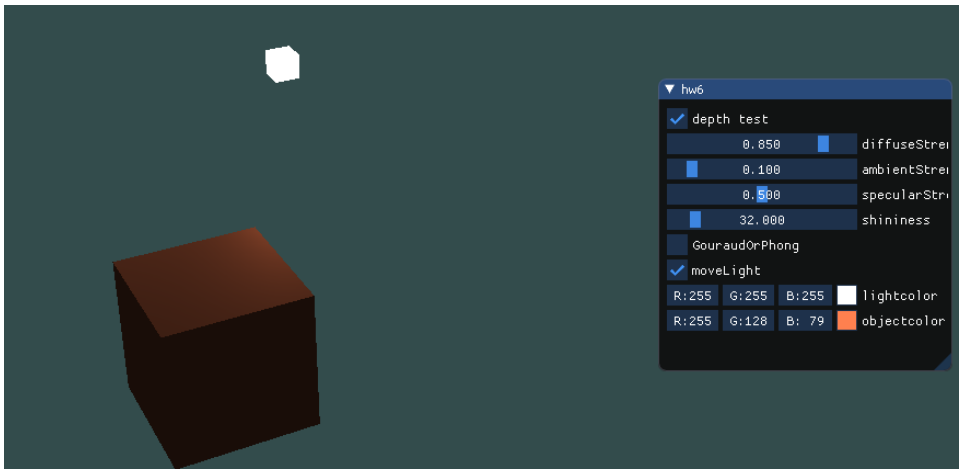
代码的实现的话就将代表光源位置的 cube 的 module 矩阵加一个随时间变化的就行。

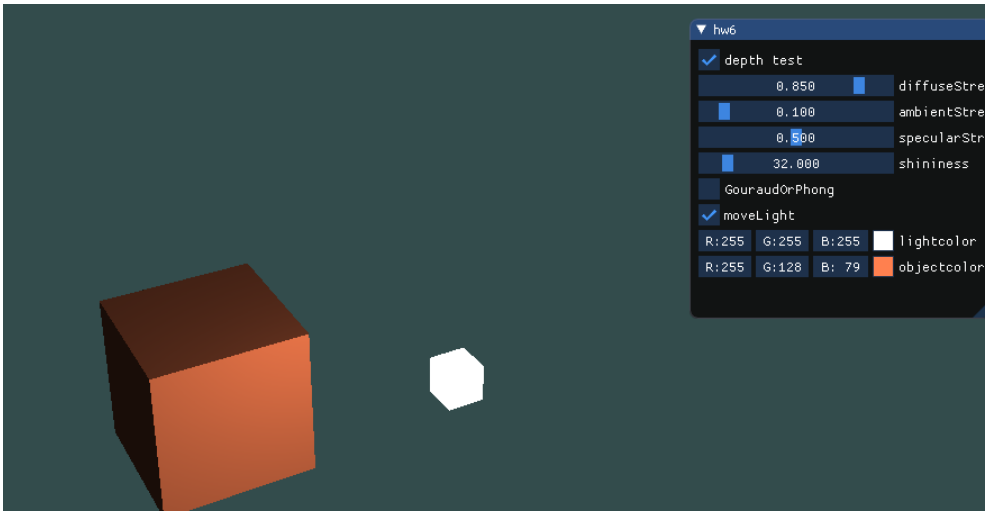
```
if (moveLight)
        {
            lightPos.x = 1.0f + sin(glfwGetTime()) * 2.0f;
            lightPos.z = sin(glfwGetTime() / 2) * 2.0f;
        }
```

```
model = glm::mat4(1.0f);
        model = glm::translate(model, lightPos);
        model = glm::scale(model, glm::vec3(0.2f));
        lampShader.setMat4("model", model);
```

运行截图：






hw6
depth test
0.850
diffuseStre
0.100
ambientStre
0.500
specularStr
32.000
shininess
GouraudOrPhong
moveLight
R:255
G:255
B:255
lightcolor
R:255
G:128
B: 79
objectcolor