# 简单Phong模型实现

## 写在最前

在这次作业中, 我实现了基础的`phong`光照模型, 完成了基础功能, 并且尝试了两个升级功能

- obj读取
- BRDF材质

有两个视频来展示我的功能

- 纹理.mp4
- 渲染展示.mp4

其中,纹理对纹理功能单独做了展示, 渲染展示展示了整个从`.obj`文件读取, 指定材质, 到最后基于`phong`渲染的结果.



## 项目依赖和项目结构

程序基于`GLFW`实现的`openGL`接口

使用`glad`

使用数学库`glm`

使用`stb_image.h`读取纹理图片

`objLoader.h`和`objLoader.cpp`定义了`.obj`文件读取相关函数

`maLightShader.vs.glsl`和`maLightShader.fs.glsl`分别是顶点着色器和片段着色器, phong模型实现在其中

`CornellBox-Original.obj`是我读取的`.obj`文件

`container.jpg`是一个木制材料的纹理贴图

# 三维变换函数

我使用`glm`操作我的顶点矩阵, 以实现在三维空间中的变换. 因为底层数据结构也直接使用的`glm`提供的`vec`和`mat`, 我觉得自己重载一遍变换函数对我的学习和开发都不是很有帮助,更重要的是**理解函数可以被分解为什么样的矩阵乘法**, 所以我直接使用了`glm`提供的矩阵操作函数.以下主要标注出我使用的矩阵操作函数, 并且做出原理上的解释, 以展示我对使用矩阵乘法实现三维变换的理论的了解.

## 点乘

`glm`重载了`*`符号作为向量点乘符号.

## 四元数

我们谈论过为什么要使用四个坐标来描述一个三维坐标中的点, 简而言之, 它能把所有三维坐标上的四则运算操作变成优雅的四元数构成的矩阵乘法

## translate

```
glm::mat4 translated_mat = glm::translate(glm::mat4 origin, glm::vec3 tran_vec);
```

转移函数, 由vec构造三维空间内的转移矩阵, 乘以原矩阵. 用于在三维空间中**平移**坐标

有两个input:

- `glm::mat4 origin` 前一个矩阵
- `glm::vec3 tran_vec` 三维空间内一个向量, 表示**平移**的方向和位移.

一个output:

- `glm::mat4 translated_mat` 结果一个平移矩阵.

### 数学原理

构造转移矩阵

其中, 转移向量`tran_vec`决定`dx`, `dy`, `dz`三个量

转移向量如下

假设我们有一个坐标, 在三维空间下表示为四元组向量

那么平移后的新的点的坐标就可以表示为

即

这样就能完美使用矩阵乘法实现三维空间内点的平移了

### glm实现

```
template <typename T, precision P>
    GLM_FUNC_QUALIFIER tmat4x4<T, P> translate(tmat4x4<T, P> const & m, tvec3<T, P> const &
v)
    {
        tmat4x4<T, P> Result(m);
        Result[3] = m[0] * v[0] + m[1] * v[1] + m[2] * v[2] + m[3];
        return Result;
    }
```

`glm::translate`允许我们从一个变换矩阵创造出一个新的变化矩阵, 新的变换矩阵乘以点坐标, 等价于原来的变换矩阵对点坐标操作之后, 再对点坐标进行平移操作.

如果我们从头开始, 那么声明一个$$4 \times 4$$ 的对角坐标作为`origin`即可

具体数据结构还重载了乘号.

## scale

```
glm::scale glm::mat4 scaleMat = glm::scale(glm::mat4 origin,  glm::scaleVec)
```

### 数学原理

对目标三维空间内点的单位坐标的放缩.

构造放缩矩阵

其中`scaleVec` 指定$$S_x$$, $$S_y$$ , $$S_z$$

这样, 对于一个空间中的向量(也可以说是点但是`scale`的话说向量比较好理解), 有

即

对原向量的缩放完成

### glm实现

```
template <typename T, precision P>
GLM_FUNC_QUALIFIER tmat4x4<T, P> scale(tmat4x4<T, P> const & m, tvec3<T, P> const & v)
{
   tmat4x4<T, P> Result(uninitialize);
   Result[0] = m[0] * v[0];
   Result[1] = m[1] * v[1];
   Result[2] = m[2] * v[2];
   Result[3] = m[3];
   return Result;
}
```

如我们在上一小节所说, `glm`中的操作函数都写成了生成一个操作矩阵这样的形式, 除此之外和我们分析的数学原理完全一样

如果我们想要一个有一个缩放操作的操作矩阵, 那么声明一个$$4 \times 4$$ 的对角坐标作为`origin`即可

## rotate

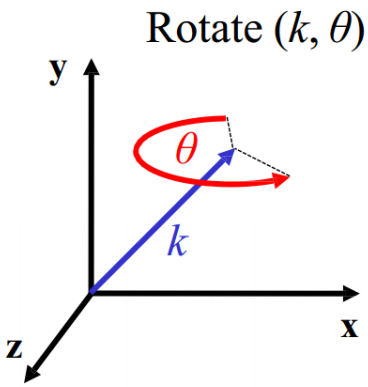
```
mat4 scaleMat = glm::rotate(mat4 origin, float angle, vec3 rotateVec)
```

实现围绕`rotateVec`旋转`angle`(弧度制)

### 数学原理

公式太难打了,直接放老师PPT

# 旋转变换




- 围绕单位向量 $(k_x, k_y, k_z)$ 的旋转变换可以写成如下的 Rodrigues 公式：

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} k_xk_x(1-c)+c & k_zk_x(1-c)-k_zs & k_xk_z(1-c)+k_ys & 0 \\ k_yk_x(1-c)+k_zs & k_zk_x(1-c)+c & k_yk_z(1-c)-k_xs & 0 \\ k_zk_x(1-c)-k_ys & k_zk_x(1-c)-k_xs & k_zk_z(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

where $c = cos\theta$ & $s = sin\theta$

**glm实现**

```
template <typename T, precision P>
GLM_FUNC_QUALIFIER tmat4x4<T, P> rotate(tmat4x4<T, P> const & m, T angle, tvec3<T, P> const
& v)
{
   T const a = angle;
   T const c = cos(a);
   T const s = sin(a);

   // 标准化
   tvec3<T, P> axis(normalize(v));
    // temp = (1 - c) * axis,即
    // temp[0] = ( 1 - c ) * kx
    // temp[1] = ( 1 - c ) * ky
    // temp[2] = (1 - c) * kz
   tvec3<T, P> temp((T(1) - c) * axis);

   // 构造矩阵Rotate
   tmat4x4<T, P> Rotate(uninitialize);
    //第一列，Rotate[0][3] = 0 初始化就决定了
   Rotate[0][0] = c + temp[0] * axis[0];
   Rotate[0][1] = temp[0] * axis[1] + s * axis[2];
   Rotate[0][2] = temp[0] * axis[2] - s * axis[1];
```

```
    Rotate[1][0] = temp[1] * axis[0] - s * axis[2];
    Rotate[1][1] = c + temp[1] * axis[1];
    Rotate[1][2] = temp[1] * axis[2] + s * axis[0];

    Rotate[2][0] = temp[2] * axis[0] + s * axis[1];
    Rotate[2][1] = temp[2] * axis[1] - s * axis[0];
    Rotate[2][2] = c + temp[2] * axis[2];

    tmat4x4<T, P> Result(uninitialize);
    // 还是glm实现思路，输入矩阵乘以Rotate
    Result[0] = m[0] * Rotate[0][0] + m[1] * Rotate[0][1] + m[2] * Rotate[0][2];
    Result[1] = m[0] * Rotate[1][0] + m[1] * Rotate[1][1] + m[2] * Rotate[1][2];
    Result[2] = m[0] * Rotate[2][0] + m[1] * Rotate[2][1] + m[2] * Rotate[2][2];
    Result[3] = m[3];
    return Result;
}
```

# 基于GLSL的Phong模型

我的Phong模型是写在片段染色器里的.

`maLightShader.vs.glsl` 负责计算模型顶点的位置, 计算变换过的法向量

`maLightShader.fs.glsl` 负责计算光照和材料影响

## Phong模型理论知识回顾

Phong模型将一个物体显示的光分为三类, **镜面反射光, 漫反射光, 环境光**, 通过计算这三种光, 将结果**叠加**, 制造出真实的渲染效果

## 光照实现

这部分的代码实现在`maLightShader.fs.glsl`中

### 镜面反射光

$$I$$为光照, $$K_s$$为漫反射系数, 有RGB三个分量

$$R$$是光反射的方向(单位向量)

$$V$$是视角的方向(单位向量)

$$n$$是反射光系数, n越大, 高光越集中. (金属质感越浓)

```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
// 如公式，一点细节是如果向量点乘小于零(法向量夹角不对，这个面实际上没有照射到)，取零
float spec = pow(max(dot(viewDir, reflectDir), 0), material.shininess);
vec3 specular = light.specular * (spec * material.specular);
```

### 漫反射光

$$I$$为光照, $$K_d$$为漫反射系数, 有RGB三个分量

$$L$$是光的方向(单位向量)

$$N$$是面的法向量(单位向量)

实现

```
// diffuse
//面法向量N
vec3 norm = normalize(Normal);
//光方向 L
vec3 lightDir = normalize(light.position - FragPos);

// 如公式，一点细节是如果漫反射光算出来小于零(法向量夹角不对，这个面实际上没有照射到)，取零
float diff = max(dot(norm, lightDir), -dot(norm, lightDir));
vec3 diffuse = light.diffuse * (diff * material.diffuse);
```

### 环境光

$$I$$为光照, $$K_a$$为环境光系数, 有RGB三个分量 代表环境光为白色的时候物体显示什么颜色.

```
vec3 ambient = light.ambient * material.ambient;
```

## 法向量变换

问题是当我们使用之前提到的变换矩阵变换面之后, 面的法向量怎么变换呢？

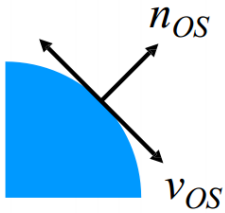
# 由切向量计算法向量

$v_{OS}$ 和 $n_{OS}$ 垂直：$n_{OS}^T v_{OS} = 0$

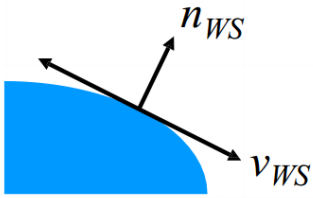
$$n_{OS}^T (M^{-1}M) v_{OS} = 0$$

$$(n_{OS}^T M^{-1})(M v_{OS}) = 0$$

$$(n_{OS}^T M^{-1}) v_{WS} = 0$$




$v_{WS}$ 和 $n_{WS}$ 垂直：$n_{WS}^T = n_{OS}^T (M^{-1})$




$$n_{WS} = (M^{-1})^T n_{OS}$$

法向量的变换矩阵是原变换矩阵的逆的转置

计算过程如上,我们只要最后结论就行, **法向量的变换矩阵是原变换矩阵的逆的转置**

我们在顶点着色器里就把法向量计算好, 保证片段着色器拿到正确的法向量数据.

具体代码在`maLightShader.vs.glsl`

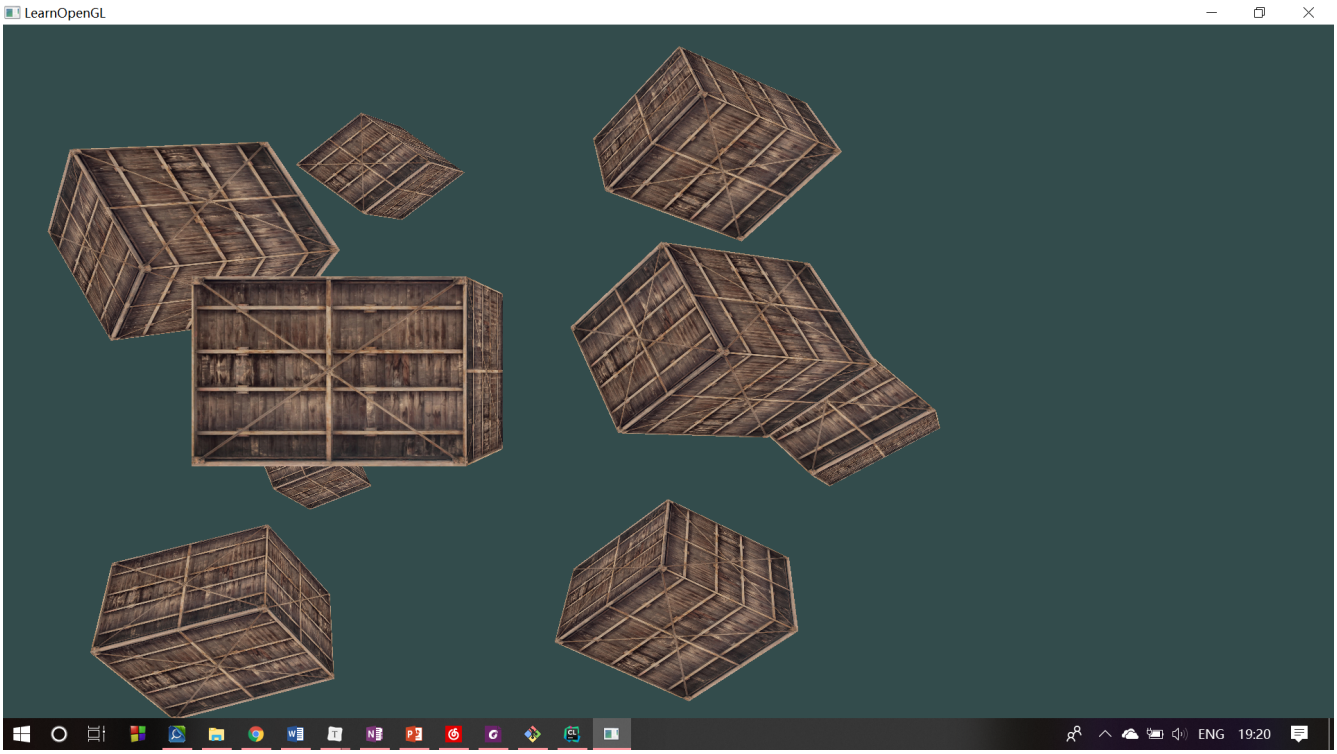
```
// 计算顶点的位置, 中间经过了 model , view , projection三个变换矩阵
// model : 把载入的模型坐标放在我们指定的世界坐标的某个位置上, 决定这个模型最初的位置(坐标和旋转情况)
// view : 随着我们观看视角的变换变换顶点, 主要是做出可以用鼠标旋转物体的功能
// projection : 最后投影到二维平面
gl_Position = projection * view * model * vec4(aPos, 1.0);
FragPos = vec3(model * vec4(aPos, 1.0));
Normal = mat3(transpose(inverse(model))) * aNormal;
```

## 纹理

简单调用`opengl`库接口, 方式基本是以下三步

- 读入纹理图片
- 指定纹理坐标(gsgl中)
- 指定纹理插值方法

在这里简单展示结果, 我给一堆方箱子贴上了箱子的纹理图片, 纹理图片是`content.jpg`




这部分单独有视频, 就不作为最后的展示的一部分了.

##升级分: 任意OBJ读取

本着学习 `.obj` 文件结构的目的, 我没有使用 `.obj` 读取库,

我写了自己的obj读取程序, 可以识别 `.obj` 文件的以下关键字:

- v
- f
- g

并且编写了几个函数, 将 `.obj` 文件转化为渲染需要的格式

# 渲染核心 -- Group类

对 `.obj` 文件里每一个 `g` 标签生成一个group对象, 把顶点数据填入GPU中, 渲染.

具体的架构在 `objLoader.h` 和 `objLoader.cpp` 中.

## 多边形分裂

`.obj` 文件的格式是 `顶点` - `面` 格式. 例子如下

```
## Object backwall
v  -0.99  0.00  -1.04
v   1.00  0.00  -1.04
v   1.00  1.99  -1.04
v  -1.02  1.99  -1.04

usemtl backwall
f -4 -3 -2 -1
material chrome
g backwall
```

但是`opengl`是以三角形为绘画单元的. 所以要将多边形分裂为三角形

主要是利用`f -4 -3 -2 -1`这一行, 这一行指定了顶点的顺序, 只需要拿出第一个顶点A, 然后顺序选取每两个顶点B和C, 和A组成一个三角形

## 法向量计算

我找到的`.obj`没有顶点法向量, 所以我写了一个算法计算法向量.

思路顺着刚刚的分裂算法说, 计算$$\overrightarrow{AB} $$ , $$\overrightarrow{AC}$$ ,得到三角形ABC的法向量

具体实现如下

```
// 本函数形成一个面的三角形分裂， 并且把三角面片的法向量存在顶点向量后
// 最后送入顶点染色器的向量格式 [x, y, z, a. b, c]
// [x, y, z] 是顶点坐标 , [a, b ,c]是对应的面法向量
std::vector<std::vector<float> > split2triangle (std::vector<std::vector<float> > polygon)
{
    std::vector<std::vector<float> > result;
    std::vector<float> pointFirst = polygon[0];
    // 加速 : 一个面应该只有一个法向量
    polygon.erase(polygon.begin());
    while (polygon.size() > 1) {
        std::vector<float> point1 = pointFirst;
        std::vector<float> point2 = polygon.back();
        polygon.pop_back();
        std::vector<float> point3 = polygon.back();
        // 求叉乘计算法向量
        int test_l = polygon.size() % 2;
        std::vector<float> normal = cross_product(sub(pointFirst, point3), sub(pointFirst,
point2));
        if (test_l == 1) {
            normal = cross_product(sub(pointFirst, point2), sub(pointFirst, point3));
        }
        // 法向量append加入每一个点
        point1.insert(point1.end(), normal.begin(), normal.end());
        point2.insert(point2.end(), normal.begin(), normal.end());
        point3.insert(point3.end(), normal.begin(), normal.end());
        // result append 三个点
        result.push_back(point1);
        result.push_back(point2);
        result.push_back(point3);
```

```
    }
    return result;
}
```

# 升级分: BRDF材料

BRDF材料有三种不同的模型建立方式

- 经验模型
- 物理模型
- 数据驱动

其中物理模型有更复杂的参数和计算关系, 但是也能得到更好的效果.经验模型因为计算量小和实现简单而应用广泛.

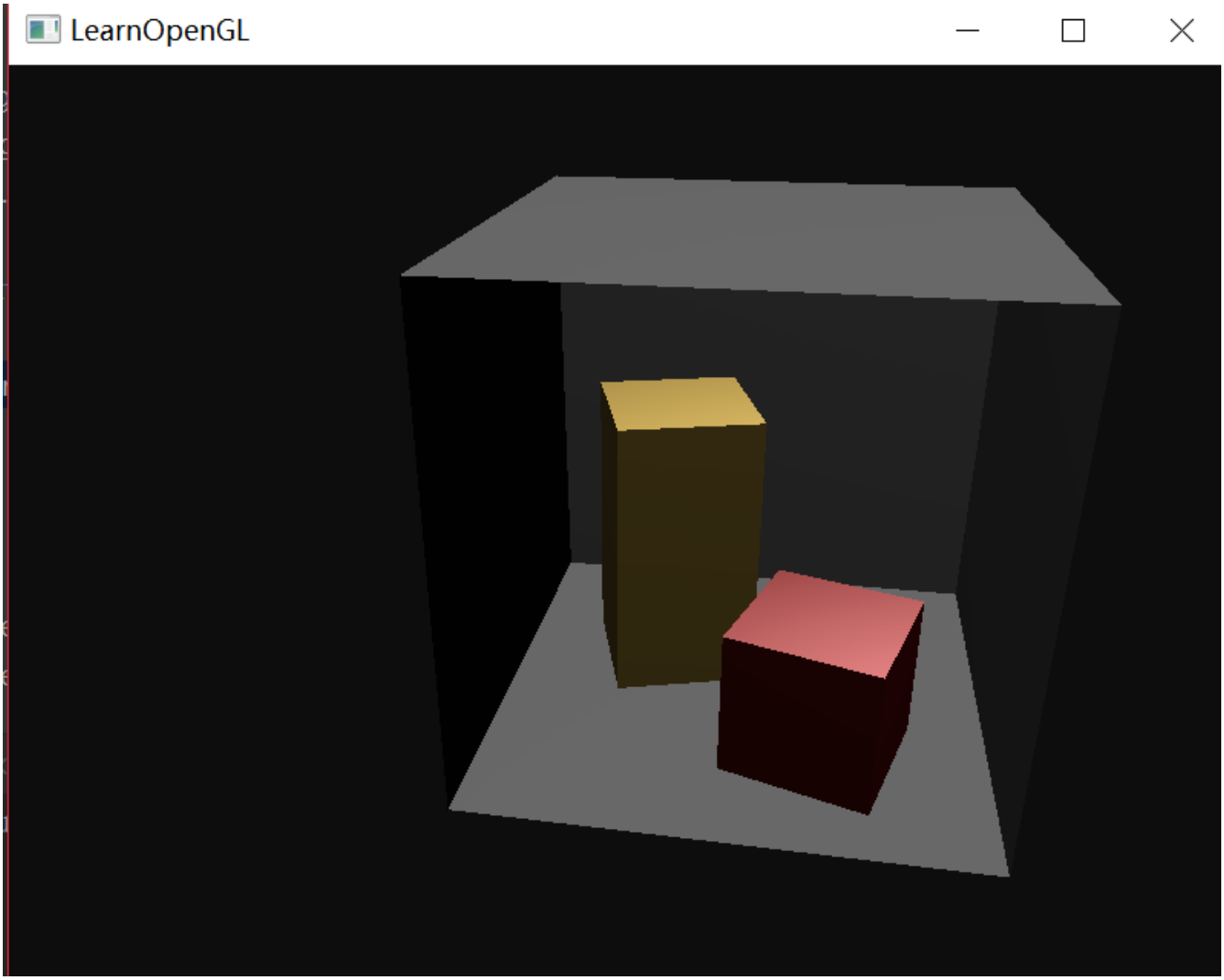
我目前支持`phong`模型支持的材料, 即通过改变物体的**漫反射常数**, **镜面反射常数** 和 **颜色**来改变物体看起来的材质.
我目前支持如下的材料(材料参数来源, 需要翻墙)

- emerald
- jade
- obsidian
- pearl
- ruby
- turquoise
- brass
- bronze
- chrome
- copper
- gold
- silver
- black_plastic
- cyan_plastic
- green_plastic
- red_plastic
- white_plastic
- yellow_plastic
- black_rubber
- cyan_rubber
- green_rubber
- red_rubber
- white_rubber
- yellow_rubber

你可以在`.obj`文件中使用`material`关键字指定材质,比如这样

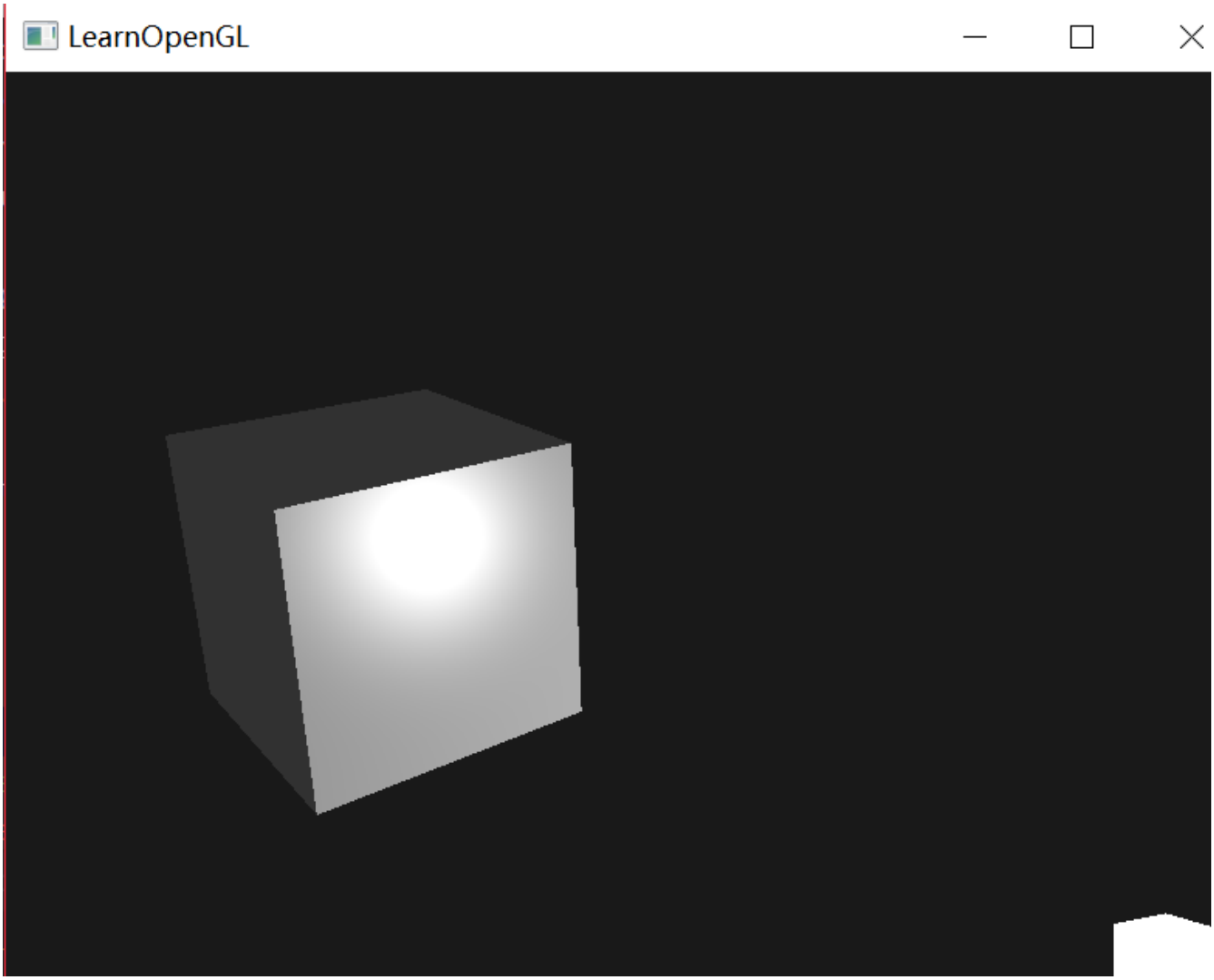
```
material chrome
g backWall
```

我把backwall部分的材质设置为铬, 长方体是金, 正方体是红宝石.你可以看到他们的高光情况确实比较不一样.

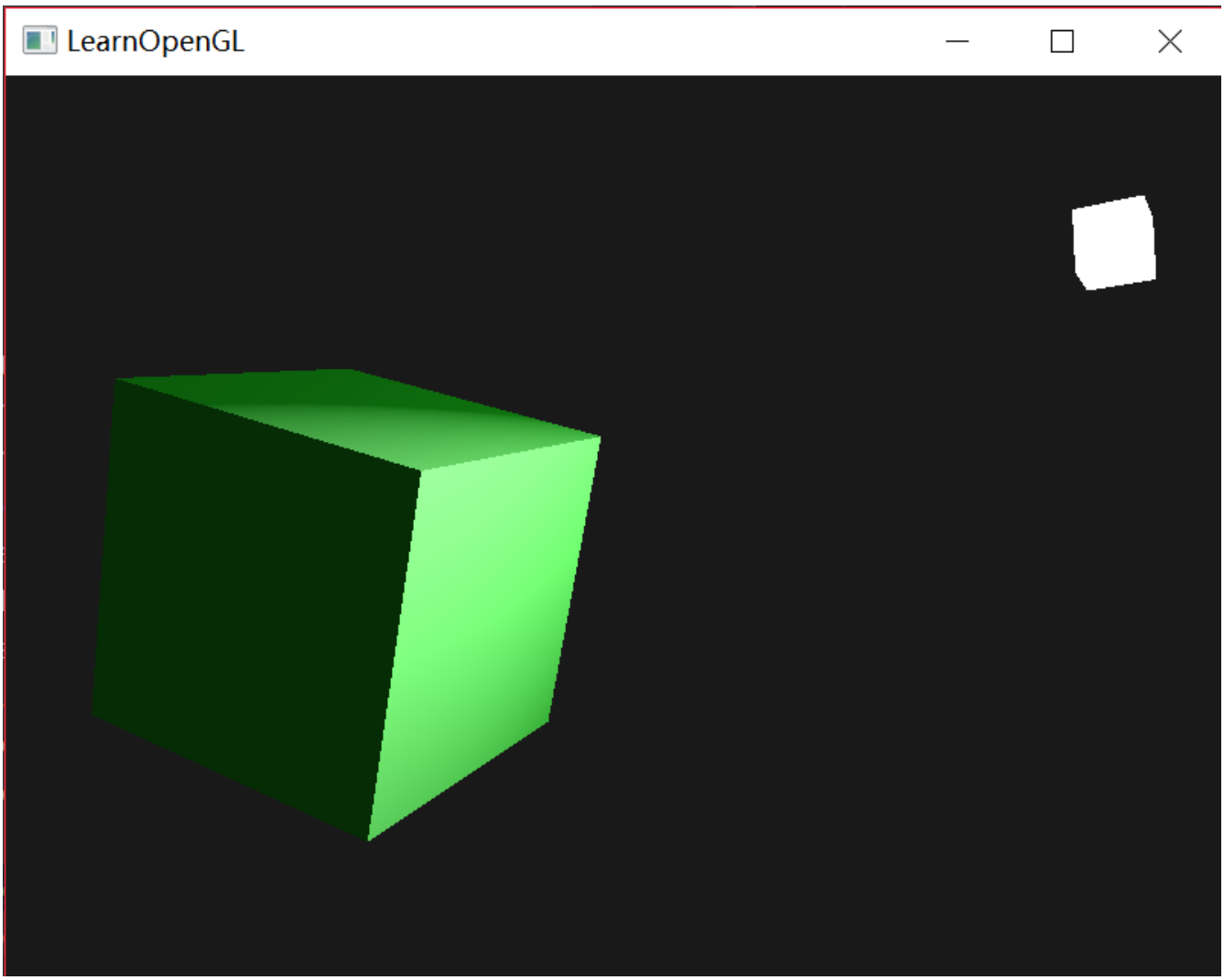



以下是更多效果, 在单个物体的情况下更容易通过设置光源位置展示材质效果, 白色的正方体是光源

银




翡翠


LearnOpenGL