School of Data and Computer Science, Sun Yat-sen University

作业 3：使用 GLSL

18364066 Lu Yanzuo

December 13, 2020

Contents

1 Introduction

绘制一个小球并使用 GLSL 实现不同着色模型，要求如下：

- 实现环境光反射，漫反射以及镜面反射
- 比较三种着色模型的效果，包括 Flat, Gouraud, Phong 等
- 比较其他绘制参数变化时，绘制结果的变化，如不同反射类型的光照强度，镜面反射中的指数 α 等
- 比较小球细分程度不同时，绘制结果的变化
- 在报告中详细讨论以上变化

2 Sphere & Subdivision

在本次作业中我使用了上次 HW2 中提到的 obj 模型，主要原因是我在搜索和学习 GLSL 相关资料时发现，在 GLSL 的使用中几乎没有实例会通过手动 glVertex 等函数来构建需要操作的模型，绝大多数都使用了 VAO(vertex buffer object) 和 VBO(vertex array object) 两个 OpenGL 的新特性来将数据传输到 GPU 以渲染图形。

VAO 把对象信息直接存储在 GPU 中，而不是在当我们需要的时候才传输，这意味着我们的应用程序不需要将数据传输到 GPU 或者是从 GPU 输出，这样也就获得了额外的性能提升。使用 VAO 并不困难，不需要大量的 glVertex 调用，而是将顶点数据存储在数组中，然后放进 VBO，最后在 VAO 中存储相关的状态。于是我在相关网站上下载了一个球体 Sphere 的 3d- obj 模型，在 Windows 自带软件下预览如下图所示。

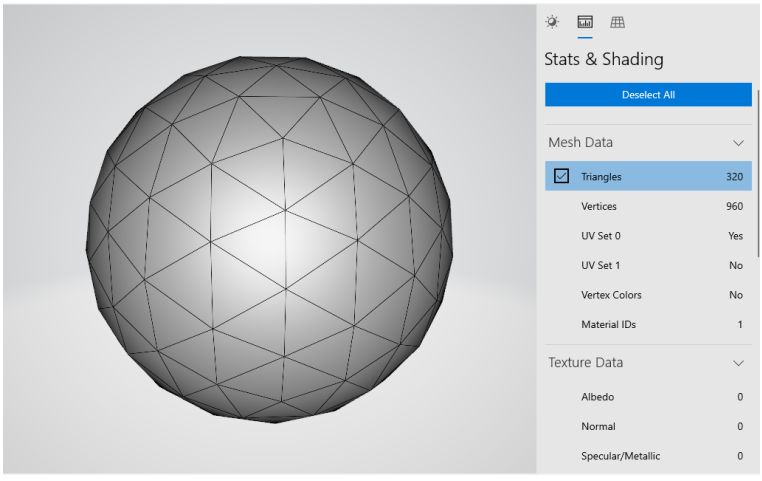

Figure 1: Sphere 球体 obj 模型

从中可以看到该球体总共包含 320 个三角形 Triangles 和 960 个顶点 Vertices，下面的光照模型都基于该球体展开，其导入方法在许多网站上都有所提及，算是一种较为普遍的操作方式，也基于此没

有办法对该球体进一步的细分，增加和减少三角形都变得相当困难，所以以下所有绘制结果都统一使用了该球体。

3 Phong Shading

3.1 Concept

提到 Phong Shading 就不得不提到老师在课上讲到的另一个相近的名词 Phong Lighting Model，后者是说物体被光照产生的效果，而前者考虑的是如何在三个顶点中填充颜色，Phong Shading 是在三种着色模型中最接近真实的一种，但其开销也是最大的。Gouraud Shading 是每个顶点 Vertex 计算一次光照，而 Phong Shading 是每个片元 Fragment 或者每个点（注意此处不是顶点 Vertex）计算一次光照，点的法向量是通过顶点的法向量插值得到的，还有一个优势是 Phong Shading 能够在减少三角面数的情况下达到看起来一样的效果，因为插值后的法向量会光滑变化。

整个着色的计算过程首先是通过对顶点法向量线性插值，计算内部片元 Fragment 的法向量，然后通过对顶点位置线性插值计算片元 Fragment 对应的三维位置，由此计算光照方向以及观察方向，最后通过光照模型计算片元 Fragment 的颜色。

3.2 Implementation

在顶点着色器中，我们首先计算顶点在裁剪空间 clip space 的坐标，并赋值到 gl_Position 变量，gl_Position 是 GLSL 中默认的归一化裁剪空间坐标，仅能在顶点着色器中使用，目的是对顶点完成用户自定义的 Model、View 和 Projection 三种矩阵变换。

```
// Vertex position in clip space.
gl_Position =  projection * view * model * vec4(vertex_model, 1);
```

接下来计算在观察空间 eye space 中，顶点 vertex 到相机 camera 和光源 light 的两个向量，再加上顶点的法线向量，将这几个向量计算出来后传递给片元着色器使用。

```
// Eye space vector from vertex to camera.
vec3 vertex_eye = (view * model * vec4(vertex_model, 1)).xyz;
eyedir_eye = vec3(0,0,0) - vertex_eye;

// Eye space vector from vertex to light.
vec3 light_eye = (view * vec4(light_world, 1)).xyz;
lightdir_eye = light_eye + eyedir_eye;
```

```
// Eye space normal vector of the vertex.
normal_eye = (normal * vec3(normal_model)).xyz;
```

在片元着色器中，需要做的工作就是分别计算环境光、漫反射光和镜面反射光三种成分，环境光比较简单不需要过多的操作，而在漫反射光和镜面反射光中则需要增加一些新的运算，光线的入射角决定了表面的亮度，这实际上是基于兰伯特余弦定理，在漫反射中我们简单地使用光线与表面法线的夹角的余弦值作为光的强度。

而在镜面反射中多考虑了观察者位置这一因素，其效果就是当观察者以某一个角度看观察对象时会比其他角度亮，并且当你偏离这个角度时亮度会减弱，最直接的例子就是当我们看向某种金属时往往等看到一束闪亮的白光而不是金属本身的颜色。

基于以上，我们只需利用在顶点着色器中已经计算好的法向量以及顶点到观察者（即相机）和光源的两个向量，计算出漫反射光的角度以及镜面反射光的角度随后完成插值操作 clamp。

```
// Diffuse angle calculation.
vec3 n = normalize(normal_eye); // normal
vec3 l = normalize(lightdir_eye); // to light
float diffuse_angle = clamp(dot(n, l), 0, 1);

// Specular angle calculation
vec3 e = normalize(eyedir_eye); // to eye
vec3 r = reflect(-l, n); // light reflection
float specular_angle = clamp(dot(e, r), 0, 1);

vec3 ambient = ambient_mat;
vec3 diffuse = diffuse_mat * light_mat * diffuse_angle;
vec3 specular = specular_mat * light_mat * pow(specular_angle,
   specular_power);

color = ambient + diffuse + specular;
```

4 Gouraud Shading & Flat Shading

4.1 Concept

简单来说，Flat Shading 就是一个三角面共用一个颜色，如果一个三角面的代表顶点（一般是最后一个或第一个）刚好被光照成了白色，那么整个面就是白的。而在此基础上，Gouraud Shading 会考虑三个顶点的信息，中间的颜色用一种二维的插值，假设一个点被光照成了白色，另外两个点都是红色，那么在这个面上颜色就是从白到红逐渐变化。

4.2 Implementation

在这两种着色模型的实现中，我们只需要实现 Gouraud Shading 一种就可以了，因为 Flat Shading 可以直接在 Gouraud Shading 的基础上加一个 flat 关键字就完成共用颜色。而 Gouraud Shading 的实现方式其实相较于 Phong Shading 简单了不少，因为 OpenGL 的渲染管线将自动地在顶点之间以正确的方式进行插值（声明了 flat 关键字除外），我们只需要在顶点着色器中正确地计算每个顶点的颜色即可，再直白一点说就是把在 Phong Shading 中片元着色器计算颜色的部分全部转移到顶点着色器来完成就完事了。

```
// Vertex position in clip space.
gl_Position =  projection * view * model * vec4(vertex_model, 1);

// Eye space vector from vertex to camera.
vec3 vertex_eye = (view * model * vec4(vertex_model, 1)).xyz;
vec3 eyedir_eye = vec3(0,0,0) - vertex_eye;

// Eye space vector from vertex to light.
vec3 light_eye = (view * vec4(light_world, 1)).xyz;
vec3 lightdir_eye = light_eye + eyedir_eye;

// Eye space normal vector of the vertex.
vec3 normal_eye = (normal * vec3(normal_model)).xyz;

// Diffuse angle calculation.
vec3 n = normalize(normal_eye); // normal
vec3 l = normalize(lightdir_eye); // to light
float diffuse_angle = clamp(dot(n, l), 0, 1);
```

```
// Specular angle calculation
vec3 e = normalize(eyedir_eye); // to eye
vec3 r = reflect(-l, n); // light reflection
float specular_angle = clamp(dot(e, r), 0, 1);

vec3 ambient = ambient_mat;
vec3 diffuse = diffuse_mat * light_mat * diffuse_angle;
vec3 specular = specular_mat * light_mat * pow(specular_angle,
   specular_power);

color = ambient + diffuse + specular;
```

这样我们在片元着色器中只需要一行代码，其中 gl_FragColor 也是和 gl_Position 一样是 GLSL 语言的关键字，和直接输出 color 没什么不同，只是为了更加简洁。

```
gl_FragColor = vec4(color, 1.0);
```

以上就是三种着色模型的 GLSL 语言代码，因为 Gouraud Shading 是比 Phong Shading 更为简单的操作，且可以进行代码的复用，所以将 Phong Shading 的部分放在了前面，其实只要 OpenGL 的渲染管线能够实现自动插值对我们编写 GLSL 是十分容易的，无非就是计算法向量以及几个角度相乘就可以，但在学习的过程中还是遇到了诸多困难，例如一些特别函数的使用方法。

至于 main 函数部分载入 shader 和构建 program 等操作，从作业的要求来看并不是主要考察内容，且实现方式千篇一律，在传递参数时比较困难的一个点应该是实验环境需要 glm 库的使用，复现方法将在报告的末尾进行展示，下面一节将展示三种着色模型的不同效果。

5 Results

下面第一小节会是基于一组普通且视觉效果较好的参数观察三种着色模型的差异，然后第二到第四小节是在 Phong Shading 的基础上分别观察这三种光对视觉效果的影响。

5.1 baseline

首先基础的参数分别是环境光 (0.5, 0.0, 0.0)、漫反射光 (0.5, 0.0, 0.0) 和镜面反射光 (0.3, 0.3, 0.3)，目标是构建一个红色的小球且光斑类似于金属上看到的那种白色。从图中我们可以看到 Flat Shading 的每个三角面的颜色都是相同的，而在 Gouraud Shading 中则能够看出视觉上相对来说更好了一些，

但其实仔细观察还是能看到三角面之间不可消除的马赫带，插值效果虽有但并不理想，至于 Phong Shading 则是最为逼真的一种了，也能更好地看到我们的白色光斑对应于镜面反射光 (0.3, 0.3, 0.3)。

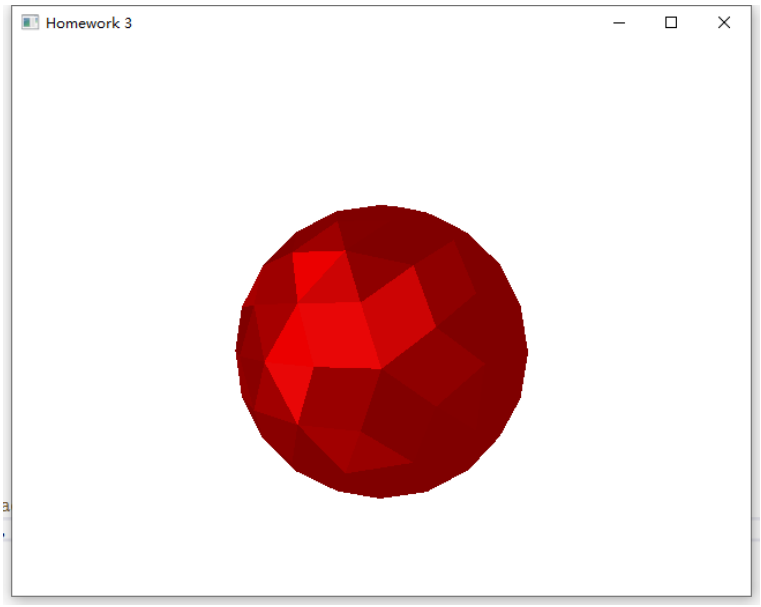

Figure 2: Flat Shading: baseline

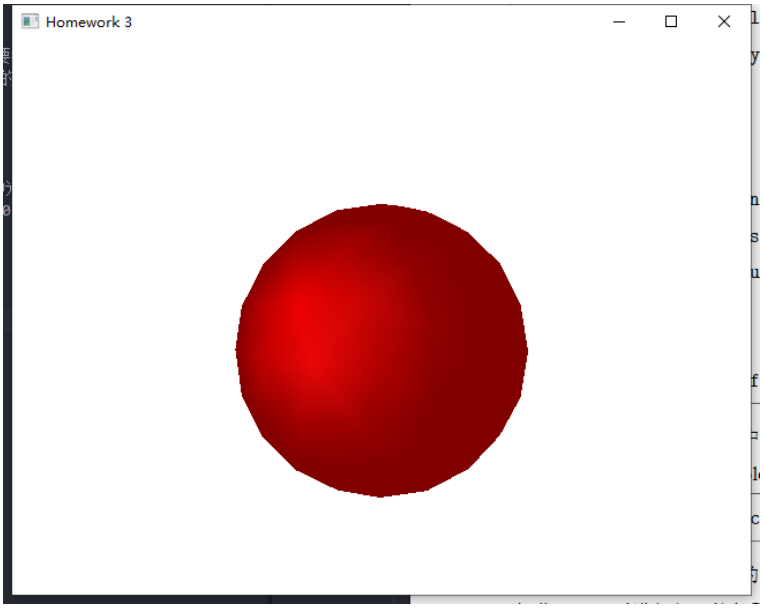

Figure 3: Gouraud Shading: baseline

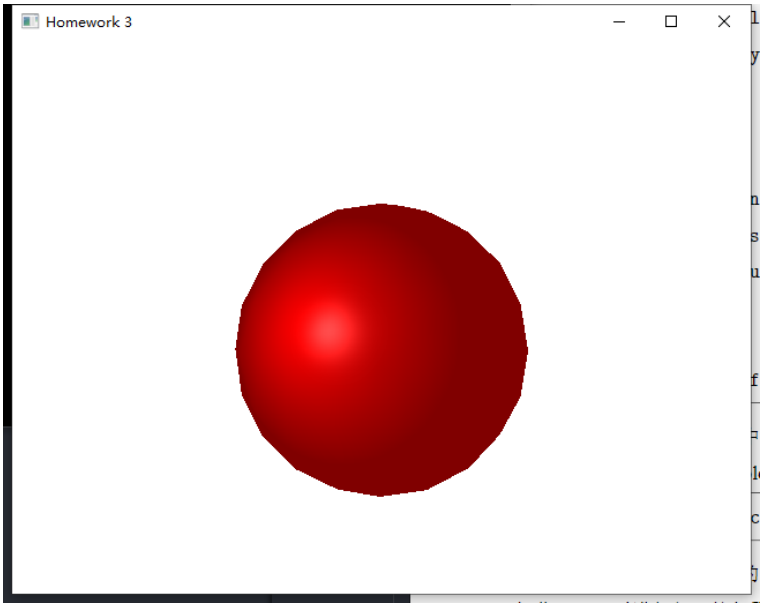

Figure 4: Phong Shading: baseline

5.2 ambient

下面是将环境光的红色分量分别调小到 0.2 和调大到 0.8 的效果，可以看到效果还是十分显著的，调小和调大其实就相当于在房间中开灯和关灯一个道理，整体而不是局部的亮度会得到降低或上升。

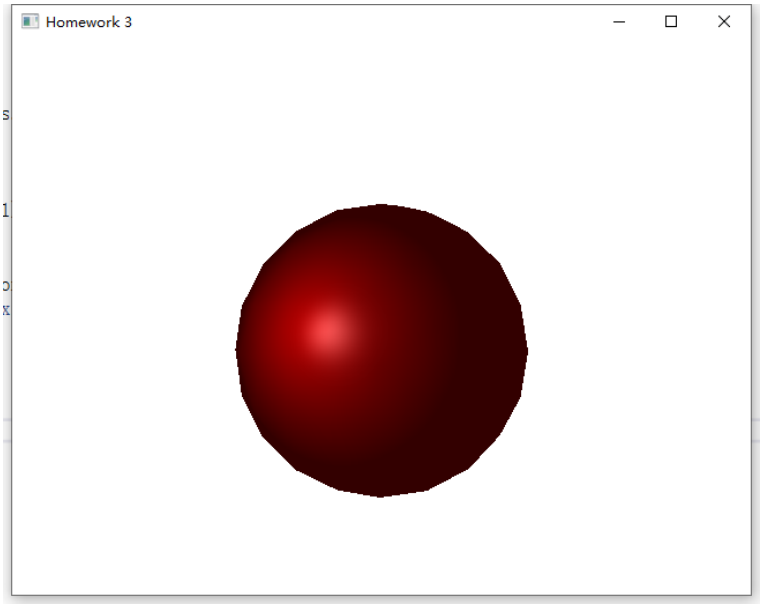

Figure 5: ambient: lower

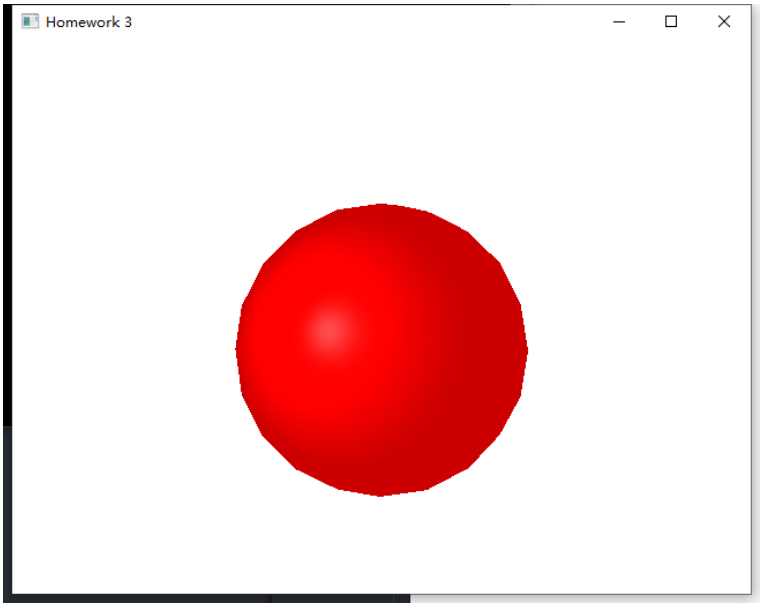

Figure 6: ambient: higher

5.3 diffuse

下面是将漫反射光的红色分量分别调小到 0.2 和调大到 0.8 的效果，可以看到这种变换和环境光是有所不同的，从视觉上来看明显的能够感受到相较于 baseline，分量较小时暗的地方会变得更暗，分量较大时亮的地方会变得更亮，这就是因为漫反射光反映的是物体本身的一个反射光强度。

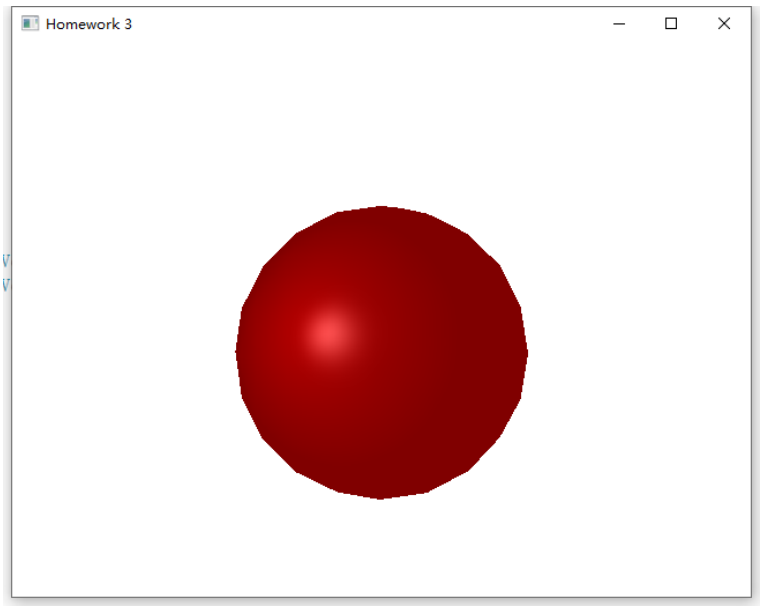

Figure 7: diffuse: lower

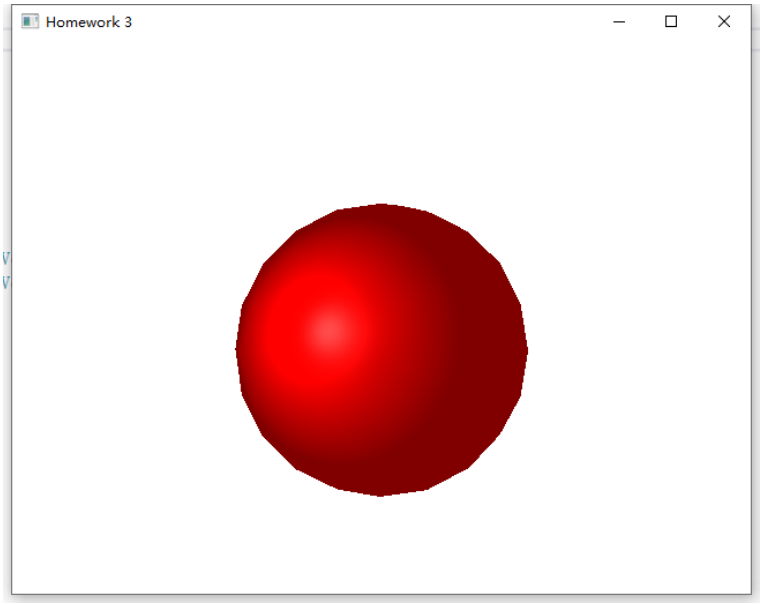

Figure 8: diffuse: higher

5.4 specular

下面是将镜面反射光三个分量同时分别调小到 0.1 和调大到 0.5 的效果，可以看到最明显的变化就是，亮斑的强度是随分量的大小正比变化的，当镜面反射光分量越小时，亮斑就会变得越稀松，越大时亮斑就会变得越耀眼。

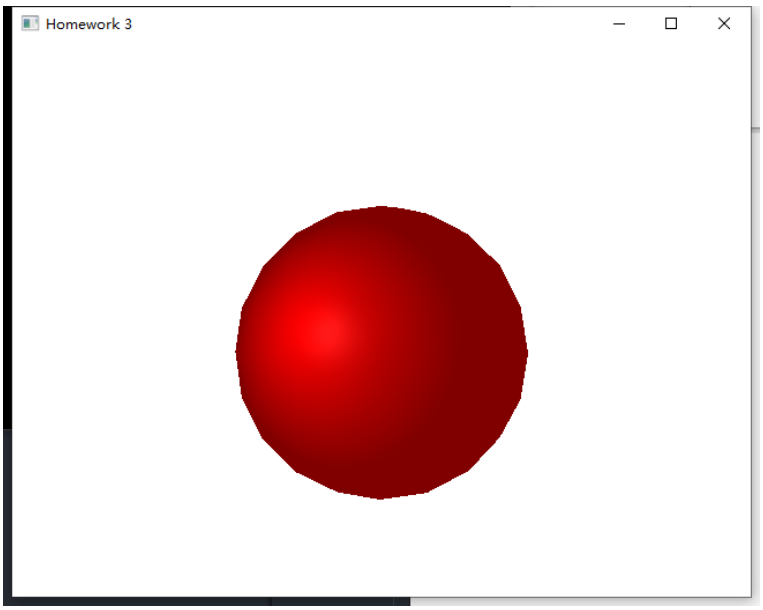

Figure 9: specular: lower

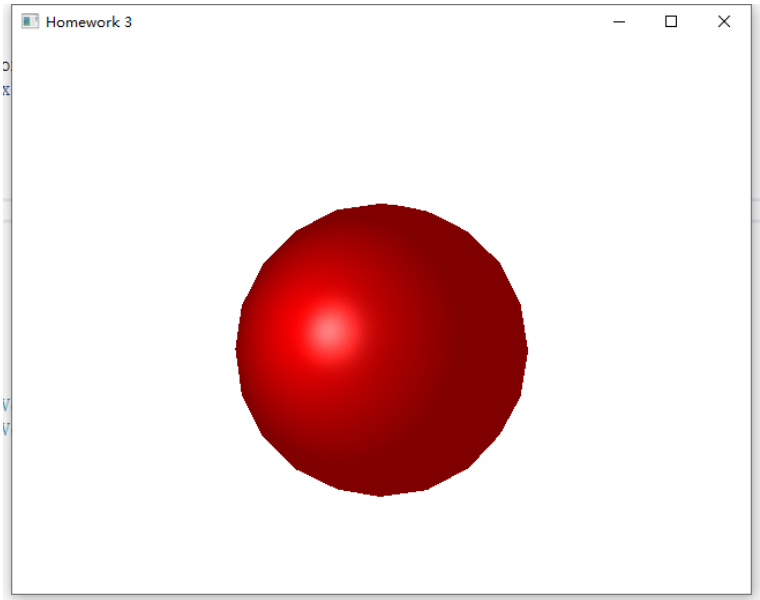

Figure 10: specular: higher

6 Summary & Usage

正如前面所提到的，由于使用了 obj 模型，在此基础上对小球进行细分十分困难，主要原因是没有学习过相关的 obj 模型知识，也很难找到不同细分程度的 obj 文件，但其实很容易地脑测出不同细分程度对视觉效果的影响。当球面细分程度越高，也就是模型包含的三角面越多时，球体看上去也会更加圆润和光滑（至少是在 Phong Shading 下），随着计算开销的增大球面也会更加细腻，而当球面细分程度越低，也就是模型包含的三角面越少时，球体看上去就会更加粗糙和棱角分明，在 Gouraud Shading 下马赫带应该也会更加地清晰，计算开销减小的同时球面也会更加网格化。

若要复现以上的着色模型，比较**重要**的一点就是要将 src/glm 这个库复制到本机路径中 glew 安装的 include 文件夹中，例如 $C:\backslash glew-2.1.0\backslash include\backslash glm$，然后若要使用不同的着色模型，需要在 main.cpp 中第 125 行和第 126 行将顶点着色器和片元着色器的文件名字改为需要的即可，例如 flat.vs, flat.fs 等。