HW3：GLSL及VBO入门

学号：19335109	课程：计算机图形学
姓名：李雪堃	学期：Fall 2021
专业：计算机科学与技术（超算）	教师：陶钧
邮箱：i@xkun.me	TA：席杨

Table of Contents

(一) 作业要求

- 绘制一系列沿固定轨迹运动的小球（如太阳系）
- 通过GLSL实现Phong-shading
  - 比较Phong-shading与OpenGL自带的flat与smooth shading的区别
  - 可选做：比较Blinn-Phong shading与Phong shading的区别
- 使用VBO对小球进行绘制
  - 使用足够的细分产生充足的顶点和三角面片，便于计算绘制时间
  - 比较不同细分层级下shading的区别
  - 讨论是否使用VBO及index array的效率区别

(二) 核心代码和过程

(1) 球体类 Sphere

`Sphere` 类的代码在 `include/Sphere.h` 和 `src/Sphere.cpp` 下。

在 `Sphere.h`，声明一个 `Sphere` 类，用于存储球体的顶点数据（位置向量、法向量）。构造函数根据细分层级构造球体，并绑定 VAO、VBO、EBO，生成顶点数据。析构函数会对 VAO、VBO、EBO 解绑。

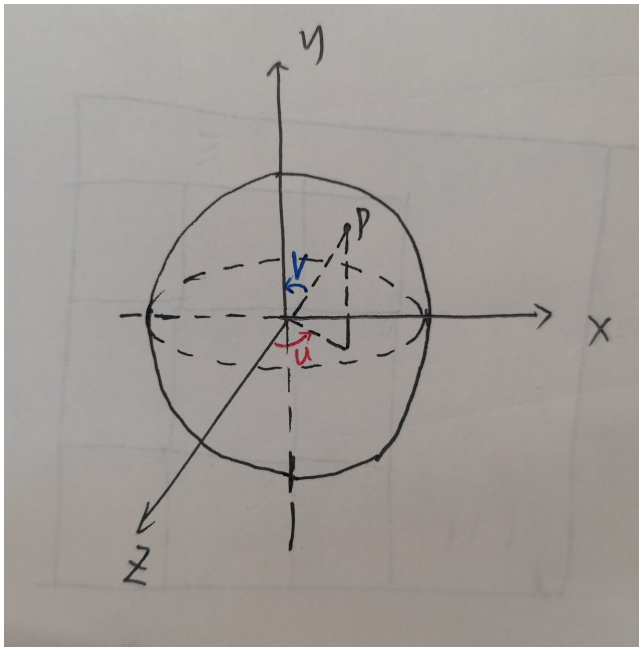
`draw()` 使用 VAO 绘制球体，而 `draw_oldschool()` 使用 `glBegin()` 和 `glEnd()` 绘制。

```
#ifndef SPHERE_H
#define SPHERE_H

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <vector>

#include "Shader.h"

const GLfloat PI = glm::pi <GLfloat> ();

class Sphere
{
private:
  GLuint VAO;
  GLuint VBO;
  GLuint EBO;

  std::vector<glm::vec3> positions; // position vectors
  std::vector<glm::vec3> normals;   // normal vectors

  std::vector<float> sphere_vertices;
  std::vector<int> sphere_indices;

public:
  Sphere(int X_SEGMENTS, int Y_SEGMENTS); // split into X_SEGMENTS x Y_SEGMENTS pieces
  ~Sphere();

  void draw();  // draw sphere using opengl core profile mode
  void draw_oldschool();  // draw sphere using deprectaed glBegin() and glEnd()
};

#endif
```

球体坐标按照球的参数方程来分解。对于球上一点 $p(x,y,z)$，可以用 (u,v) 参数方程来表示。

$$\begin{cases} x &= r\sin(\pi v)\sin(2\pi u) \\ y &= r\cos(\pi v) \\ z &= r\sin(\pi v)\cos(2\pi u) \end{cases}$$

其中，r 是球体半径，球心为 $(0,0,0)$。πv 是点 p 与球心（即原点）连线与 Y 轴正向的夹角，$2\pi u$ 表示 p 在 X-Z 平面上的投影与 Z 轴正向的夹角，且 u 和 v 满足 $u,v\in[0,1]$。那么，我们在细分球面时，就可以将 v 和 u 进行细分，带入参数方程就可以得到顶点的坐标。比如说设置 u 和 v 的细分层级为 32，那么 u 和 v 将被分成 $0,\frac{1}{32},\ldots,\frac{31}{32},1$。



另外，还需要求出我们细分的所有点的法向量，用于光照模型中计算反射光。

如果我们就将原点作为球心，法向量实际上就和点的坐标相同，不需要用叉积计算了。做平移和旋转变换就可以让球呈现在我们想要的位置。

但是，这种方法需要在 vertex shader 中重新计算法向量，传递给 fragment shader，因为变换后实际的法向量肯定会不一样。

下面是细分球面的代码，是 `Sphere` 类的构造函数。在细分时，按照我们的想法，位置向量和法向量就是一样的。然后在计算顶点索引时，要注意分奇偶行。最后将顶点坐标和法向量存储在 `sphere_vertices` 中，每六个一组绘制。

```cpp
Sphere::Sphere(int X_SEGMENTS, int Y_SEGMENTS)
{
  for (int y = 0; y <= Y_SEGMENTS; ++y)
  {
    for (int x = 0; x <= X_SEGMENTS; ++x)
    {
      float xSegment = (float)x / (float)X_SEGMENTS;
      float ySegment = (float)y / (float)Y_SEGMENTS;
      float xPos = std::cos(xSegment * 2.0f * PI) * std::sin(ySegment * PI);
      float yPos = std::cos(ySegment * PI);
      float zPos = std::sin(xSegment * 2.0f * PI) * std::sin(ySegment * PI);

      positions.push_back(glm::vec3(xPos, yPos, zPos));
      normals.push_back(glm::vec3(xPos, yPos, zPos));
    }
  }

  bool oddRow = false;
  for (int y = 0; y < Y_SEGMENTS; ++y)
  {
    if (!oddRow)
    {
      for (int x = 0; x <= X_SEGMENTS; ++x)
      {
        sphere_indices.push_back(y * (X_SEGMENTS + 1) + x);
        sphere_indices.push_back((y + 1) * (X_SEGMENTS + 1) + x);
      }
    }
    else
    {
      for (int x = X_SEGMENTS; x >= 0; --x)
      {
        sphere_indices.push_back((y + 1) * (X_SEGMENTS + 1) + x);
        sphere_indices.push_back(y * (X_SEGMENTS + 1) + x);
      }
    }
    oddRow = !oddRow;
  }

  for (int i = 0; i < positions.size(); ++i)
  {
    sphere_vertices.push_back(positions[i].x);
    sphere_vertices.push_back(positions[i].y);
    sphere_vertices.push_back(positions[i].z);
    if (normals.size() > 0)
    {
      sphere_vertices.push_back(normals[i].x);
      sphere_vertices.push_back(normals[i].y);
      sphere_vertices.push_back(normals[i].z);
    }
  }

  glGenVertexArrays(1, &VAO);
  glGenBuffers(1, &VBO);
  glGenBuffers(1, &EBO);

  glBindVertexArray(VAO);

  glBindBuffer(GL_ARRAY_BUFFER, VBO);
  glBufferData(GL_ARRAY_BUFFER, sphere_vertices.size() * sizeof(float), &sphere_vertices[0], GL_STATIC_DRAW);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sphere_indices.size() * sizeof(int), &sphere_indices[0], GL_STATIC_DRAW);

  glEnableVertexAttribArray(0);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void *)0);
  glEnableVertexAttribArray(1);
  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void *)(3 * sizeof(float)));
}
```

用 VAO 绘制球面。

```cpp
void Sphere::draw()
{
  glBindVertexArray(VAO);
  glDrawElements(GL_TRIANGLE_STRIP, sphere_indices.size(), GL_UNSIGNED_INT, 0);
}
```

用 `glBegin` 和 `glEnd` 绘制球面。

```cpp
void Sphere::draw_oldschool()
{
  glBegin(GL_TRIANGLE_STRIP);
  for (int i = 0; i < sphere_indices.size(); i++)
  {
    glVertex3f(positions[sphere_indices[i]].x, positions[sphere_indices[i]].y, positions[sphere_indices[i]].z);
    glNormal3f(normals[sphere_indices[i]].x, normals[sphere_indices[i]].y, normals[sphere_indices[i]].z);
  }
  glEnd();
}
```

(2) 不同 Shading 的实现

(2-1) Phong Shading and Blinn-Phong Shading

vertex shader 中传入 mvp 三个矩阵，然后对坐标做变换后传递给 fragment shader。

法线需要用法线矩阵来计算，即 model 矩阵左上角逆矩阵的转置。

```glsl
#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);
}
```

fragment shader 根据处理后的顶点坐标和法向量计算光照。

Phong 模型：$\mathbf{I} = k_a\mathbf{L}_a + k_d\mathbf{L}_d(\mathbf{n}\cdot\mathbf{l}) + k_s\mathbf{L}_s\cdot\max\{(\mathbf{n}\cdot\mathbf{h})^\alpha, 0\}$

- 环境光系数设置为 0.1，环境光强度 `lightColor` 需要设置
- 漫反射光的光线方向向量通过顶点坐标和光源位置得到
- 镜面反射系数设置为 1.0，高光系数设置为 32

```glsl
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 FragPos;

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

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

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

  // specular
  float specularStrength = 1.0;
  vec3 viewDir = normalize(viewPos - FragPos);
  vec3 reflectDir = reflect(-lightDir, norm);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
  vec3 specular = specularStrength * spec * lightColor;

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

对于Blinn-Phong 模型，只需要修改 fragment shader 中的反射向量为半角向量，用半角向量与法向量的夹角近似反射角。

```glsl
vec3 halfwayDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(norm, halfwayDir), 0.0), 32);
```

然后，在 `src/main.cpp` 中设置相机和视角、shader 的参数。

```
glm::mat4 model = glm::mat4(1.0f);
glm::vec3 camera_position(0.0f, 0.0f, 5.0f);
glm::mat4 view = glm::lookAt(camera_position, glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));

int width, height;
glfwGetWindowSize(window, &width, &height);
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float) width / (float) height, 0.1f, 100.0f);

shader.use();
shader.setMat4("model", model);
shader.setMat4("view", view);
shader.setMat4("projection", projection);

shader.setVec3("lightPos", glm::vec3(-10.0f, 10.0f, 10.0f));
shader.setVec3("viewPos", camera_position);
shader.setVec3("lightColor", glm::vec3(1.0f, 1.0f, 1.0f));
shader.setVec3("objectColor", glm::vec3(0.118f, 0.565f, 1.0f));

sphere.draw();
```

(2-2) Flat Shading and Smooth Shading

Flat shading 和 Smooth shading 由 OpenGL 提供。

将光源位置、视角和相机位置、光源强度等全部设置为和 Phong 与 Blinn-Phong 模型相同。

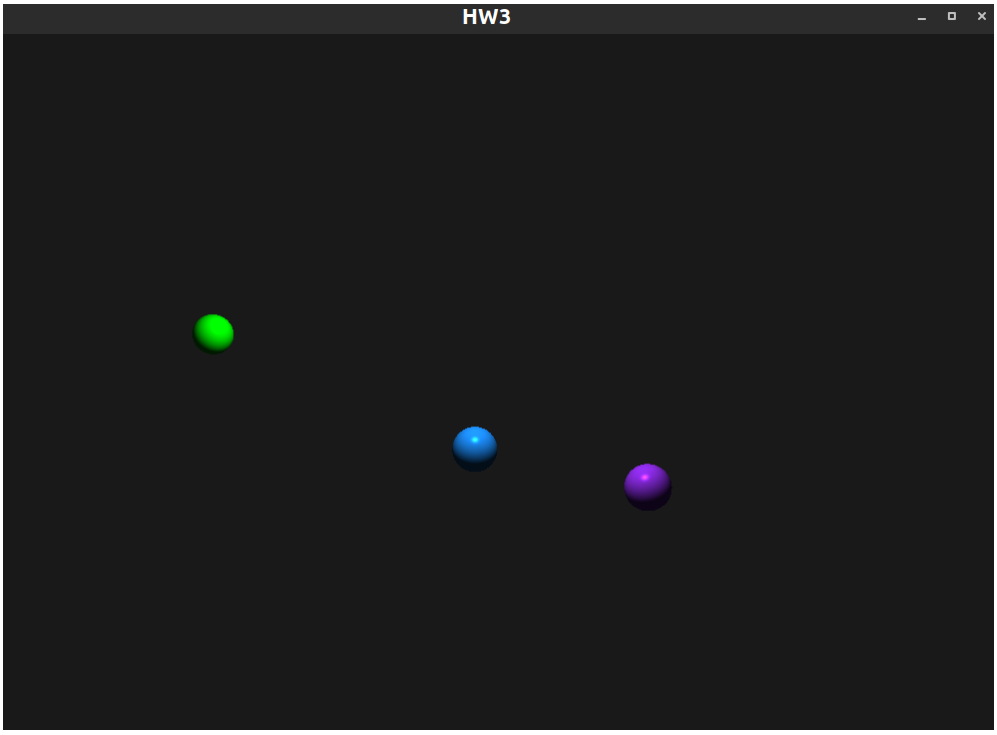
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
int width, height;
glfwGetWindowSize(window, &width, &height);
gluPerspective(45.0, (double) width / (double) height, 0.1, 100.0);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

GLfloat light_pos[] = {-10.0f, 10.0f, 10.0f, 0.0f};
GLfloat light_ambient[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat light_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};

glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

GLfloat material_ambient[] = {0.118f, 0.565f, 1.0f, 1.0f};
GLfloat material_diffuse[] = {0.118f, 0.565f, 1.0f, 1.0f};
GLfloat material_specular[] = {0.118f, 0.565f, 1.0f, 1.0f};
GLfloat material_shininess[] = {16};

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, material_ambient);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, material_specular);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, material_shininess);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);

sphere.draw_oldschool();
```

这种使用 `glBegin` 和 `glEnd` 的方法效率远低于使用 VAO 的方法，因为 VAO 是将顶点数据一次性拷贝到显存，而 `glBegin/glEnd` 中使用 `glVertex` 都会对顶点数据进行一次拷贝，内存与显存间的通信耗时大。

(三) 实验结果

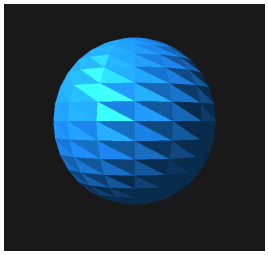
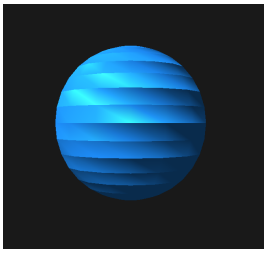
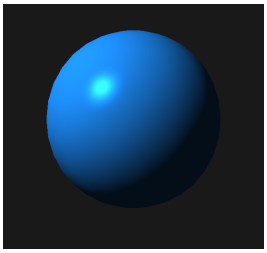
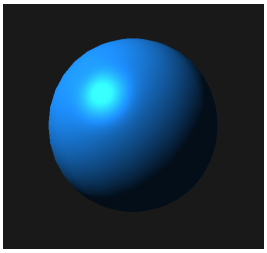
(1) 多个小球旋转的场景

设置 3 个小球绕同一中心旋转，速度不同，使用 Phong 光照模型。下面是俯视的视角。

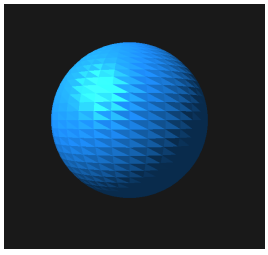
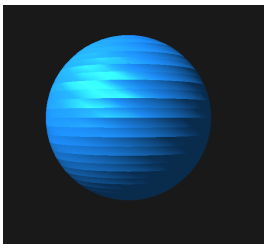
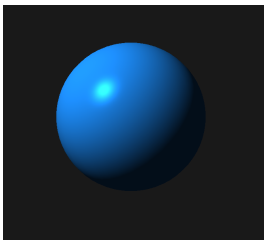
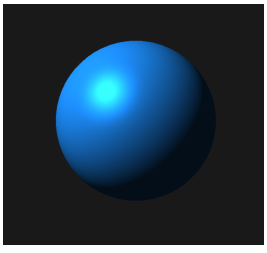


(2) 不同细分层级下不同 Shading 的区别

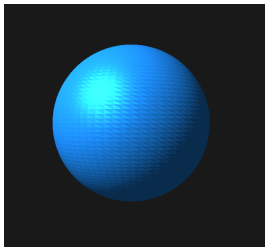
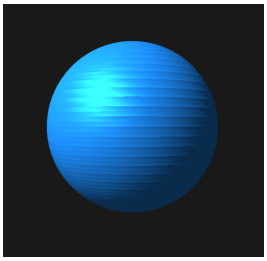
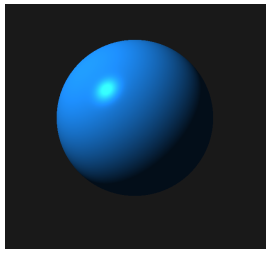
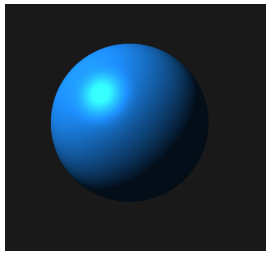
16x16:

Flat	Smooth	Phong	Blinn-Phong
			

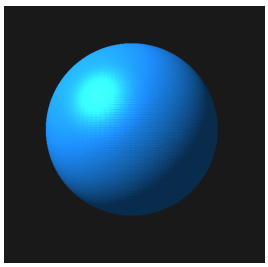
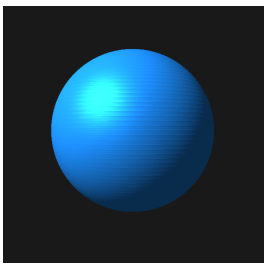
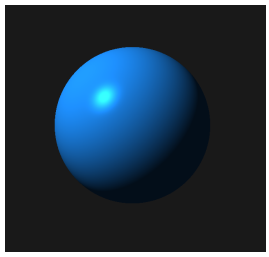
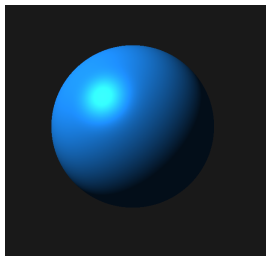
32x32:

Flat	Smooth	Phong	Blinn-Phong
			

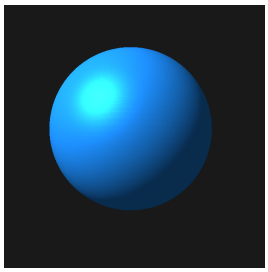
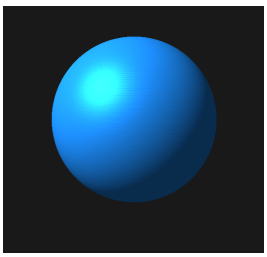
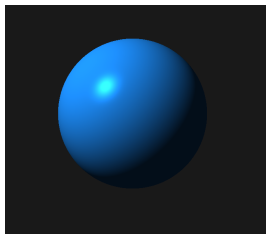
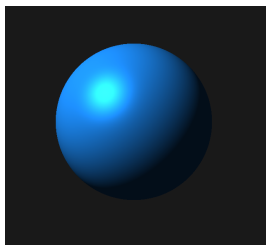
64x64:

Flat	Smooth	Phong	Blinn-Phong
			

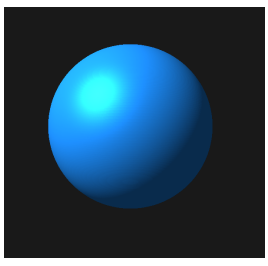
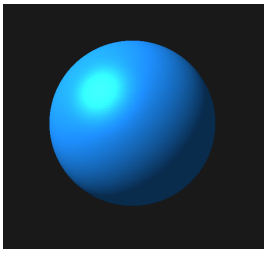
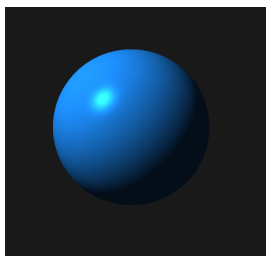
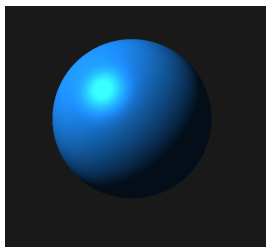
128x128:

Flat	Smooth	Phong	Blinn-Phong
			

256x256:

Flat	Smooth	Phong	Blinn-Phong
			

512x512:

Flat	Smooth	Phong	Blinn-Phong
			

可以看到，当细分层级较小时 (< 128x128)，Flat shading 和 Smooth shading 都有非常明显的马赫带，当细分层级增加时，Flat 和 Smooth 的马赫带基本消失，效果相似。

而对于 Phong 和 Blinn-Phong 模型，两者在 16x16 ~ 512x512 的细分层级下的效果各自的变化都不大，而 Phong 与 Blinn-Phong 可以看到明显的高光区别，Phong 模型高光较为集中，而 Blinn-Phong 高光有一定模糊。这是因为镜面反射光的计算不同，Blinn-Phong 使用半角向量的优点是可以减少计算量，不用计算反射方向 $\mathbf{r}$，而且可以避免 $\mathbf{r} \cdot \mathbf{v} = 0$ 不连续的情况。

(四) 实验总结

这次作业我学习了基本的 GLSL 着色器的编写和语法，学会了如何编写 vertex shader 和 fragment shader，而且学到了使用 VAO 来绘制图形的 OpenGL 核心模型，这种模式与立即模式相比，虽然写起来麻烦，但是与 OpenGL 渲染管线对应，需要我们了解更多渲染管线的知识。

自己使用 shader 实现了 Phong 和 Blinn-Phong 光照模型，对于两者对光照的计算和区别有了更深的理解和掌握。

(五) 参考资料

- https://blog.csdn.net/majing19921103/article/details/45017547
- https://cloud.tencent.com/developer/article/1686214
- https://learnopengl-cn.github.io/