计算机图形学作业三

Phong shading 与 VBO 绘制三维物体：

【专业】计算机科学与技术　　【学号】22336259　　【姓名】谢宇桐

一、通过 fragment shader 实现 Phong shading：

1.1 在 vertex shader 中输出法向量；

算法原理：

vertex shader： 顶点着色器是图形渲染管线中的一部分，负责处理每个顶点的数据，包括位置、法线等，并执行必要的变换以准备后续的渲染过程。

```
// 顶点着色器
const char* vertexShaderSource = R"glsl(
#version 330 core
// 1.顶点输入：着色器接收两个输入属性，`position`和`normal`，分别代表顶点的位置和法线向量。
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
// 定义了两个输出变量，用于将变换后的法线和位置传递给片段着色器。
out vec3 fragNormal;
out vec3 fragPosition;
// 2.变换顶点位置：使用模型矩阵(`modelMatrix`)将顶点位置从模型空间转换到世界空间，然后使用视图矩阵(`viewMatrix`)和投影矩阵(`projMatrix`)将位置进一步转换到裁剪空间。
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;

uniform vec3 lightPosition;
uniform vec3 viewPosition;

void main() {
    // 3.计算变换后的顶点位置
    fragPosition = vec3(modelMatrix * vec4(position, 1.0));

    // 3.计算变换后的法向量
    fragNormal = mat3(modelMatrix) * normal;

    // 计算裁剪坐标
    gl_Position = projMatrix * viewMatrix * modelMatrix * vec4(position, 1.0);

    // 4.计算从顶点位置到光源位置的向量
    vec4 vertex_in_modelview_space = viewMatrix * modelMatrix * vec4(position,
1.0);
    vec3 vertex_to_light_vector = lightPosition - vertex_in_modelview_space.xyz;
//
}
)glsl";
```

1.2 GLSL 会自动插值并输入 fragment shader；

在init shader对着色器进行初始化：

```
// 初始化着色器
void MyGLWidget::initShader() {
    // 编译顶点着色器
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, 0);
    glCompileShader(vertexShader);

    // 编译片段着色器
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, 0);
    glCompileShader(fragmentShader);

    // 创建着色器程序并链接
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);

    // 这里添加顶点属性的绑定
    glBindAttribLocation(shaderProgram, 0, "position");
    glBindAttribLocation(shaderProgram, 1, "normal");

    glLinkProgram(shaderProgram);
    glUseProgram(shaderProgram);

    // 启用顶点属性数组
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    // 指定顶点属性数据的格式
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
(GLvoid*)0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (GLvoid*)
(3 * sizeof(float)));

    // 释放着色器资源
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
}
```

1.3 在 fragment shader 中通过 Phong shading 计算三类反射。

算法原理：

fragment shader： 片段着色器是图形渲染管线中处理每个像素颜色的阶段，它通常在顶点着色器之后执行。这个特定的片段着色器实现了Phong光照模型，这是一种广泛使用的光照模型，用于模拟物体在光源下的反射特性。仿照上一次作业的Phong Shading，代码如下：

```
// 片段着色器代码
const char* fragmentShaderSource = R"glsl(
```

```
#version 330 core

in vec3 fragNormal;
in vec3 fragPosition;

out vec4 fragColor;

uniform vec3 lightPosition;
uniform vec3 lightColor = vec3(0.5, 0.0, 0.8); // 光源颜色
uniform vec3 viewPosition; // 观察者位置
uniform float shininess = 250.0; // 镜面高光参数

void main() {
    // 基础颜色
    vec3 baseColor = vec3(1.0, 1.0, 1.0);
    vec3 ambientColor = vec3(0.1, 0.1, 0.1);  // 环境光颜色
    vec3 specularColor = vec3(0.2, 0.2, 0.2); // 镜面反射颜色

    // 环境光分量
    vec3 color = baseColor * ambientColor * 0.2 * lightColor;

    // 漫反射光
    vec3 diffuseColor = vec3(0.9, 0.9, 0.9); // 漫反射颜色
    vec3 lightDir = normalize(lightPosition - fragPosition); // 光源方向
    vec3 norm = normalize(fragNormal); // 法线
    float diff = max(dot(norm, lightDir), 0.0f); // 漫反射强度
    color += diff * diffuseColor;

    // 镜面反射分量
    vec3 viewDir = normalize(viewPosition - fragPosition); // 观察方向
    vec3 reflectDir = reflect(-lightDir, norm); // 反射方向
    float spec = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);
    color += spec * specularColor;

    fragColor = vec4(color, 1.0); // 输出最终颜色
}
)glsl";
```

二、使用 VBO 存储顶点与连接关系：

算法原理：

VBO：Vertex Buffer Object。VBO是OpenGL中用于存储顶点数据的缓冲区对象，在GPU中存储数据，可以提高渲染效率。在代码中创建一个VBO并将其绑定到当前的OpenGL上下文中，以便后续可以向其中写入顶点数据。

```
// 初始化VBO
void MyGLWidget::initVBO() {
    // 创建和绑定 VBO
    glGenBuffers(1, &vbo);
    // 将刚创建的VBO绑定到GL_ARRAY_BUFFER目标
```

```
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
}

// 传参、执行
void MyGLWidget::update_VBO() {
    // 将顶点数据上传到VBO中
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * vertexData.size(),
vertexData.data(), GL_STATIC_DRAW);

 // 传入着色参数
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "viewMatrix"), 1,
GL_FALSE, glm::value_ptr(viewMatrix));
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projMatrix"), 1,
GL_FALSE, glm::value_ptr(projMatrix));
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "modelMatrix"), 1,
GL_FALSE, glm::value_ptr(modelMatrix));
    glUniform3f(glGetUniformLocation(shaderProgram, "lightPosition"),
lightPosition.x, lightPosition.y, lightPosition.z);
    glUniform3f(glGetUniformLocation(shaderProgram, "viewPosition"),
camPosition.x, camPosition.y, camPosition.z);
    glUniform4f(glGetUniformLocation(shaderProgram, "lightColor"), 0.5, 0.0, 0.8,
1.0);
    // 画小球不用偏移量
    glUniform2f(glGetUniformLocation(shaderProgram, "offset"), offset.x,
offset.y);

    // 重新绑定VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    // 指定顶点坐标的格式
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // 指定法线的格式，假设法线在每个顶点数据的后面，每个法线有3个分量
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3
* sizeof(float)));
    glEnableVertexAttribArray(1);
    // 绘制
    glDrawArrays(GL_TRIANGLES, 0, vertexData.size());

}

// 使用完后解绑
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

2.1 可通过细分物体(如小球)产生足够多的三角面片。

算法原理：

1. **正四面体**：正四面体是一个由四个等边三角形组成的多面体，是球体的一个简单近似。
2. **递归细分**：通过递归地将每个三角形面片细分为四个更小的三角形，可以逐步逼近球面。每次细分都会增加球体的顶点数和面数，从而提高球体的平滑度。
3. **归一化**：在每次细分后，新生成的顶点需要归一化到单位球面上，即确保每个顶点到球心的距离为1。

自己画一个球心在原点半径为1的小球模型：

```
// 深度d可以控制球体的平滑度和顶点数
void MyGLWidget::Ball(int d) {
    // 定义正四面体的顶点，初始的四个点是中心为原点的正四面体
    glm::vec3 v[4] = {
        glm::vec3(0.0f, 0.0f, 1.0f),
        glm::vec3(0.0f, 0.942809f, -0.333333f),
        glm::vec3(-0.816497f, -0.471405f, -0.333333f),
        glm::vec3(0.816497f, -0.471405f, -0.333333f)
    };

    // 清空三角形列表
    triangles.clear();

    // 递归细分四面体的每个面
    divide_triangle(v[0], v[1], v[2], d);
    divide_triangle(v[0], v[1], v[3], d);
    divide_triangle(v[0], v[2], v[3], d);
    divide_triangle(v[1], v[2], v[3], d);

    // 处理每个细分后的三角形
    for (int i = 0;i < triangles.size();i++) {
        // 添加顶点位置
        vec3 vertex = normalize(triangles[i]);
        vertexData.push_back(vertex.x);
        vertexData.push_back(vertex.y);
        vertexData.push_back(vertex.z);
        // 计算并添加法线
        vertexData.push_back(vertex.x);
        vertexData.push_back(vertex.y);
        vertexData.push_back(vertex.z);
    }
}
```

细分三角面片：

因为模型为球心在原点半径为1的球体，法向量和坐标值一样，所以这里计算新的点的坐标就是将三个顶点的坐标之和然后归一化（即投影到球上），一分四

```
void MyGLWidget::divide_triangle(vec3 v1, vec3 v2, vec3 v3, int d) {
    if (d > 0) {
        vec3 v_new = normalize(v1 + v2 + v3) ;
        divide_triangle(v1, v2, v_new, d - 1);
        divide_triangle(v1, v3, v_new, d - 1);
        divide_triangle(v2, v3, v_new, d - 1);
    }
    else {
        triangles.push_back(v1);
        triangles.push_back(v2);
        triangles.push_back(v3);
    }
}
```

效果如后图。

三、使用多个细分迭代次数讨论以下内容：

3.1 对比 Phong shading 与 OpenGL 自带的 smoothing shading 的区别；

smooth shading：

在平滑着色中，每个顶点都被赋予一个颜色值。这些颜色值在顶点着色器中计算，并传递给片段着色器。片段着色器会根据顶点颜色对每个像素的颜色进行插值，使颜色在顶点之间平滑过渡。对于法向量，它先计算三角形三顶点处的法向量，然后将这三个法向量线性插值的结果作为整个三角形的法向量。

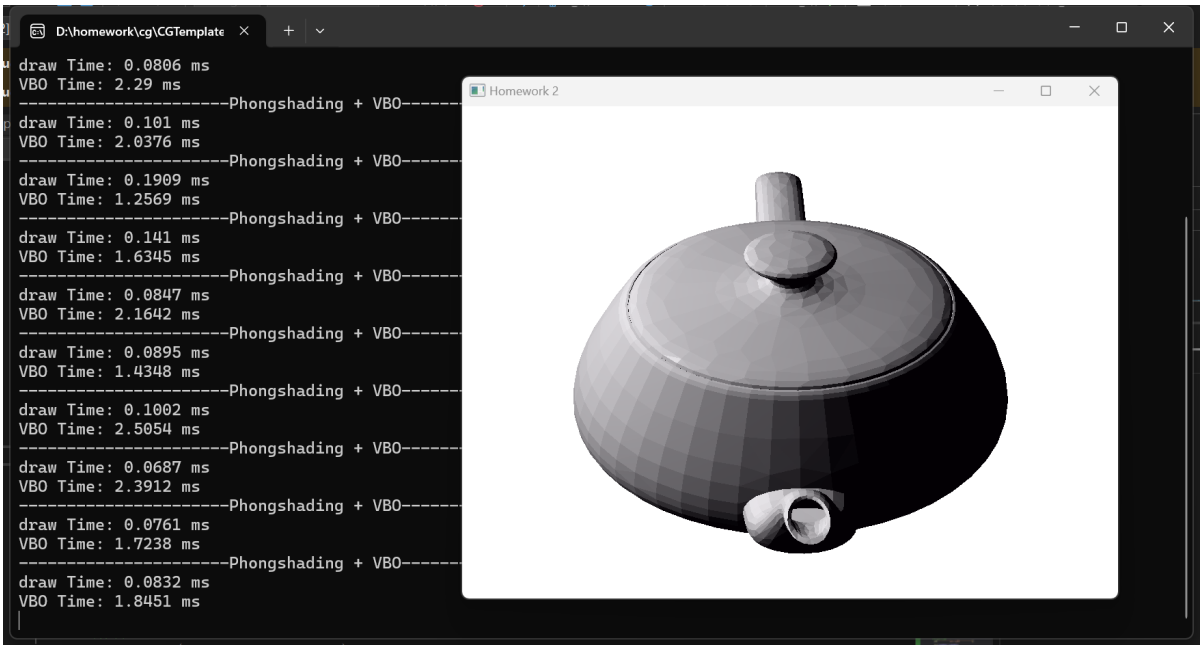
```
// 模型：  glvertex + GL_SMOOTH
void MyGLWidget::scene_2() {

    printf("-----------------------Glvertex + GL_SMOOTH--------------------------
-\n");

    glUseProgram(0);
    // 初始化
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glShadeModel(GL_SMOOTH);

    objModel.loadModel("./objs/teapot_8000.obj");

    // 获取顶点数据
    vertexData.clear();
    for (int j = 0; j < objModel.triangleCount; j++) {
        Triangle nowTriangle = objModel.getTriangleByID(j);
        getTriangleData(nowTriangle);
    }

    // 计算摄像机、光照点信息
    float dist = 200.0;

    // 自主设置变换矩阵
    camPosition = vec3(100 * sin(degree * 3.14 / 180.0) +
objModel.centralPoint.y, 100 * cos(degree * 3.14 / 180.0) +
objModel.centralPoint.x, 10 + objModel.centralPoint.z);
    camLookAt = objModel.centralPoint;     // 例如，看向物体中心
    camUp = vec3(0, 1, 0);          // 上方向向量
    // 单一点光源，可以改为数组实现多光源
    lightPosition = objModel.centralPoint + vec3(-100, 0, 20);

    // 设置光照颜色
    GLfloat lightColor[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat lightPosition[] = { objModel.centralPoint.x,   dist +
objModel.centralPoint.y, dist + objModel.centralPoint.z };
    // 设置模型本身颜色
    GLfloat modelAmbient[] = { 0.1f, 0.1f, 0.1f, 1.0f }; // 环境光
    GLfloat modelDiffuse[] = { 0.9, 0.9, 0.9, 1.0 };   // 漫反射光
    GLfloat modelSpecular[] = { 0.75f, 0.75f, 0.75f, 1.0f };  // 镜面反射光
```

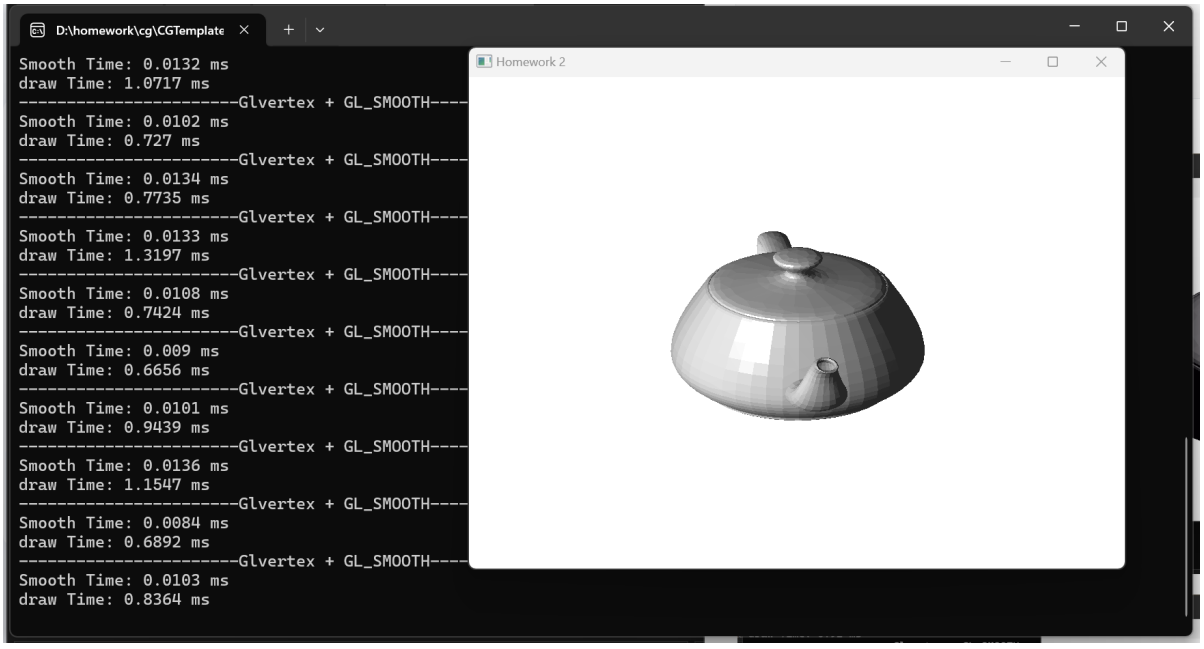
```
    GLfloat modelShininess = 250.0f; // 反光度

    auto smooth_start_time = std::chrono::high_resolution_clock::now();

    glMaterialfv(GL_FRONT, GL_AMBIENT, modelAmbient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, modelDiffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, modelSpecular);
    glMaterialf(GL_FRONT, GL_SHININESS, modelShininess);

    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightColor);
    glEnable(GL_LIGHTING);//启用光照
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);//启用颜色追踪
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // 设置投影矩阵
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    int width = 100;
    int height = 100;
    int farness = 500;
    glOrtho(-width, width, -height, height, 0.1, farness);

    // 设置视图矩阵
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(camPosition.x, camPosition.y, camPosition.z, camLookAt.x,
camLookAt.y, camLookAt.z, camUp.x, camUp.y, camUp.z);
    auto smooth_end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> smoothTime = smooth_end_time -
smooth_start_time;
    std::cout << "Smooth Time: " << smoothTime.count() << " ms" << std::endl;


    auto draw_start_time = std::chrono::high_resolution_clock::now();
    // 执行绘制
    glPushMatrix();
    glBegin(GL_TRIANGLES);
    for (size_t i = 0; i < vertexData.size(); i += 6) {
        glColor3f(0.75f, 0.75f, 0.75f);
        glNormal3f(vertexData[i + 3], vertexData[i + 4], vertexData[i + 5]);
        glVertex3f(vertexData[i], vertexData[i + 1], vertexData[i + 2]);
    }
    glEnd();
    glPopMatrix();
    auto draw_end_time = std::chrono::high_resolution_clock::now(); // 结束时间
    // 计算并输出算法的执行时间
    std::chrono::duration<double, std::milli> drawTime = draw_end_time -
draw_start_time;
    std::cout << "draw Time: " << drawTime.count() << " ms" << std::endl;

}
```

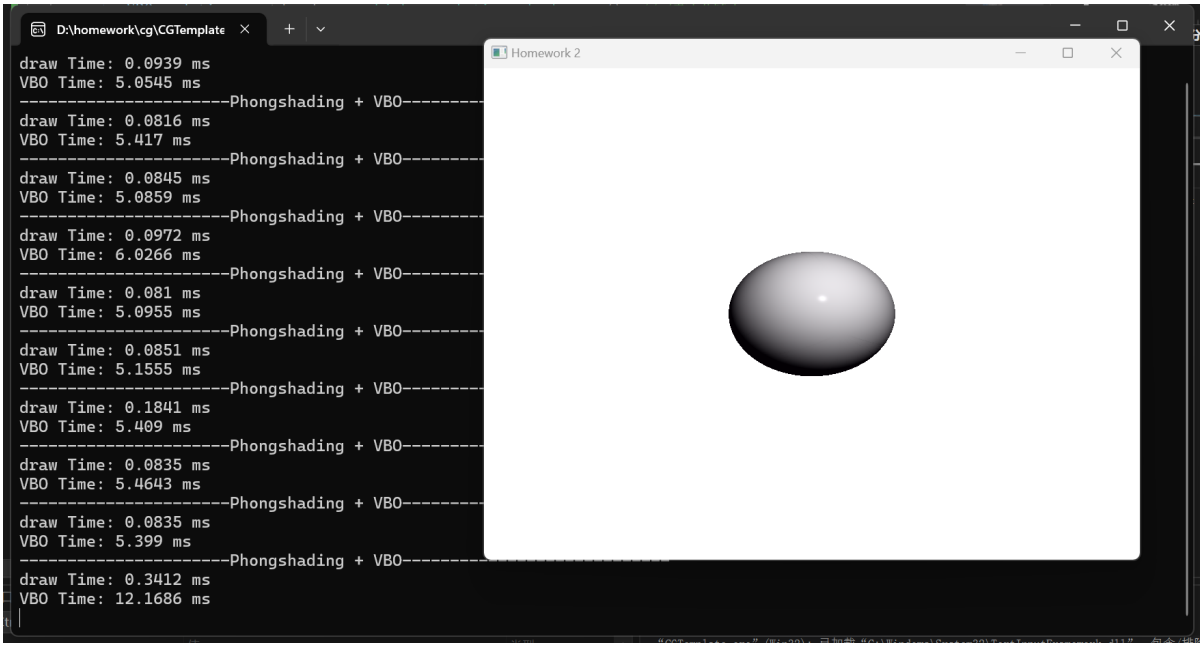
Phong shading实现代码如下：

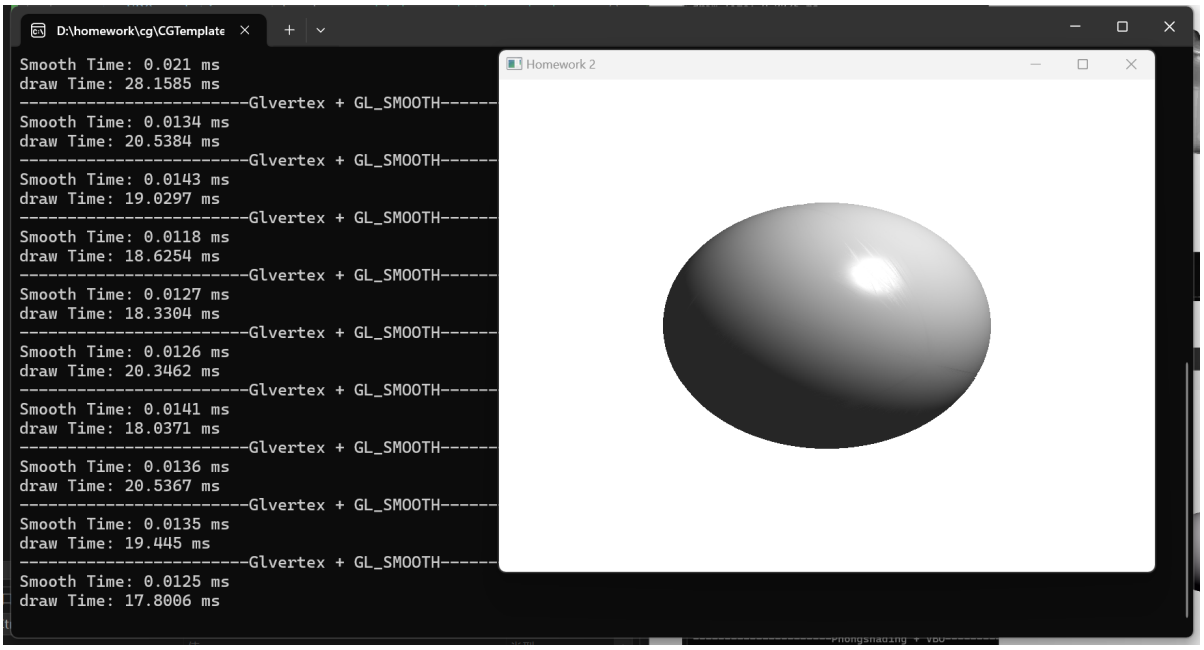
```
// 模型：phongshading + VBO
void MyGLWidget::scene_0() {

    printf("----------------------Phongshading + VBO----------------------------
\n");
    initVBO();

    // 初始化
    initShader();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    objModel.loadModel("./objs/teapot_8000.obj");

    // 获取顶点数据
    vertexData.clear();
    for (int i = 0; i < objModel.triangleCount; i++) {
        Triangle nowTriangle = objModel.getTriangleByID(i);
        getTriangleData(nowTriangle);
    }

    // 自主设置变换矩阵
    camPosition = vec3(300 * sin(degree * 3.14 / 180.0) +
objModel.centralPoint.y, 400 * cos(degree * 3.14 / 180.0) +
objModel.centralPoint.x, 10 + objModel.centralPoint.z);
    camLookAt = objModel.centralPoint;     // 例如，看向物体中心
    camUp = vec3(0, 1, 0);          // 上方向向量
    projMatrix = glm::perspective(radians(20.0f), 1.0f, 0.1f, 2000.0f);
    viewMatrix = glm::lookAt(camPosition, camLookAt, camUp);

    // 单一点光源，可以改为数组实现多光源
    lightPosition = objModel.centralPoint + vec3(0, 200, 200);
    modelMatrix = glm::mat4(1.0f); // 使用单位矩阵，不进行任何变换

    auto VBO_start_time = std::chrono::high_resolution_clock::now();
    update_VBO();
    auto VBO_end_time = std::chrono::high_resolution_clock::now(); // 结束时间
    // 计算并输出算法的执行时间
    std::chrono::duration<double, std::milli> VBOTime = VBO_end_time -
VBO_start_time;
    std::cout << "VBO Time: " << VBOTime.count() << " ms" << std::endl;

    // 解绑VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

渲染效果对比如下：


D:\homework\cg\CGTemplate
draw Time: 0.0806 ms
VBO Time: 2.29 ms
----------Phongshading + VBO----------
draw Time: 0.101 ms
VBO Time: 2.0376 ms
----------Phongshading + VBO----------
draw Time: 0.1909 ms
VBO Time: 1.2569 ms
----------Phongshading + VBO----------
draw Time: 0.141 ms
VBO Time: 1.6345 ms
----------Phongshading + VBO----------
draw Time: 0.0847 ms
VBO Time: 2.1642 ms
----------Phongshading + VBO----------
draw Time: 0.0895 ms
VBO Time: 1.4348 ms
----------Phongshading + VBO----------
draw Time: 0.1002 ms
VBO Time: 2.5054 ms
----------Phongshading + VBO----------
draw Time: 0.0687 ms
VBO Time: 2.3912 ms
----------Phongshading + VBO----------
draw Time: 0.0761 ms
VBO Time: 1.7238 ms
----------Phongshading + VBO----------
draw Time: 0.0832 ms
VBO Time: 1.8451 ms
Homework 2


D:\homework\cg\CGTemplate
Smooth Time: 0.0132 ms
draw Time: 1.0717 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0102 ms
draw Time: 0.727 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0134 ms
draw Time: 0.7735 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0133 ms
draw Time: 1.3197 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0108 ms
draw Time: 0.7424 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.009 ms
draw Time: 0.6656 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0101 ms
draw Time: 0.9439 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0136 ms
draw Time: 1.1547 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0084 ms
draw Time: 0.6892 ms
----------Glvertex + GL_SMOOTH----
Smooth Time: 0.0103 ms
draw Time: 0.8364 ms
Homework 2


D:\homework\cg\CGTemplate
draw Time: 0.0939 ms
VBO Time: 5.0545 ms
----------Phongshading + VBO----------
draw Time: 0.0816 ms
VBO Time: 5.417 ms
----------Phongshading + VBO----------
draw Time: 0.0845 ms
VBO Time: 5.0859 ms
----------Phongshading + VBO----------
draw Time: 0.0972 ms
VBO Time: 6.0266 ms
----------Phongshading + VBO----------
draw Time: 0.081 ms
VBO Time: 5.0955 ms
----------Phongshading + VBO----------
draw Time: 0.0851 ms
VBO Time: 5.1555 ms
----------Phongshading + VBO----------
draw Time: 0.1841 ms
VBO Time: 5.409 ms
----------Phongshading + VBO----------
draw Time: 0.0835 ms
VBO Time: 5.4643 ms
----------Phongshading + VBO----------
draw Time: 0.0835 ms
VBO Time: 5.399 ms
----------Phongshading + VBO----------
draw Time: 0.3412 ms
VBO Time: 12.1686 ms
Homework 2



通过对比phongshading中的**draw time**和smooth shading中的**smooth Time**，我们可以看到phong shading的渲染时间普遍比smooth time的渲染时间长。Phong Shading由于需要**在每个像素上计算光照**，能**提供更高质量的光照效果**，因此其性能消耗通常比Smooth Shading要大。Smooth Shading**通过顶点颜色插值来实现平滑效果**，而不需要在像素级别进行复杂的光照计算，因此在性能上更优。

所以总结来说，Phong Shading尤其适合需要精确模拟高光和复杂光照的场景，但性能消耗较大。而OpenGL的Smooth Shading则在性能和效果之间取得了平衡，适合于对性能要求较高的应用场景。

3.2 使用 VBO 进行绘制及通过 glVertex 进行绘制的区别；

VBO

算法已经在上方写出，我们在绘制前后加上时间代码：

```
auto draw_start_time = std::chrono::high_resolution_clock::now();
// 执行绘制
glDrawArrays(GL_TRIANGLES, 0, vertexData.size());

auto draw_end_time = std::chrono::high_resolution_clock::now(); // 结束时间
// 计算并输出算法的执行时间
std::chrono::duration<double, std::milli> drawTime = draw_end_time -
draw_start_time;
std::cout << "draw Time: " << drawTime.count() << " ms" << std::endl;
```

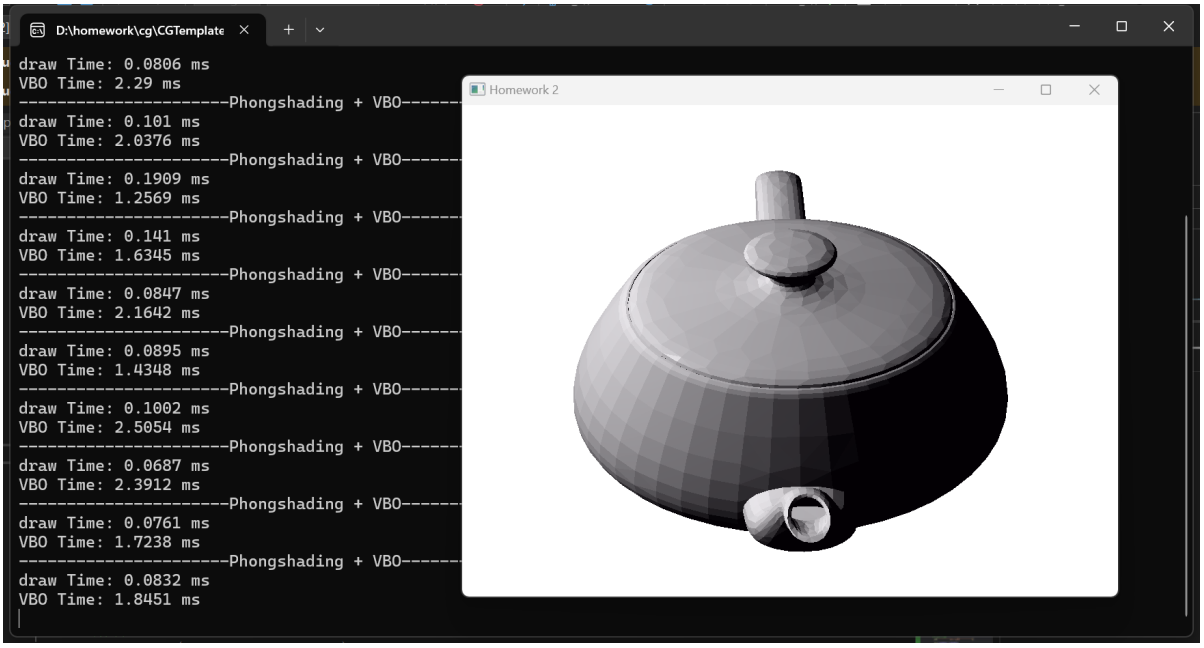
glVertex

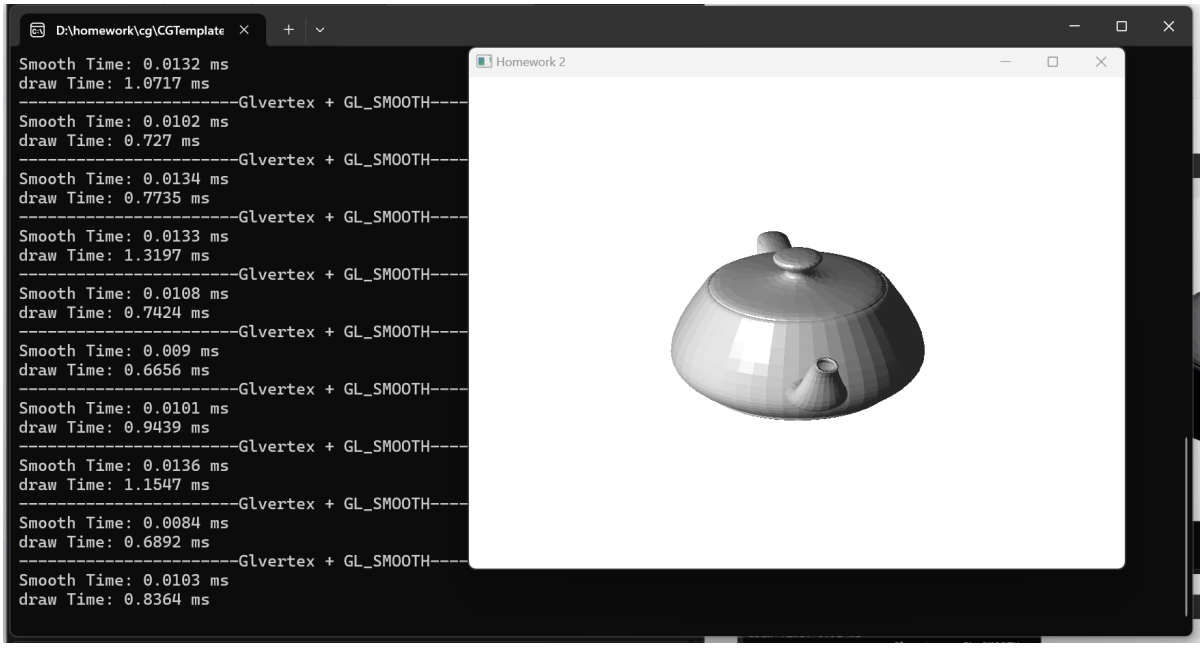
OpenGL Vertex Arrays 是一种用于存储和渲染图形数据的方法，它允许你将顶点数据（如位置、颜色、法线等）存储在一个数组中，并一次性发送给GPU进行渲染。这种方式比单独发送每个顶点的数据要高效得多，因为它减少了CPU到GPU的数据传输次数。

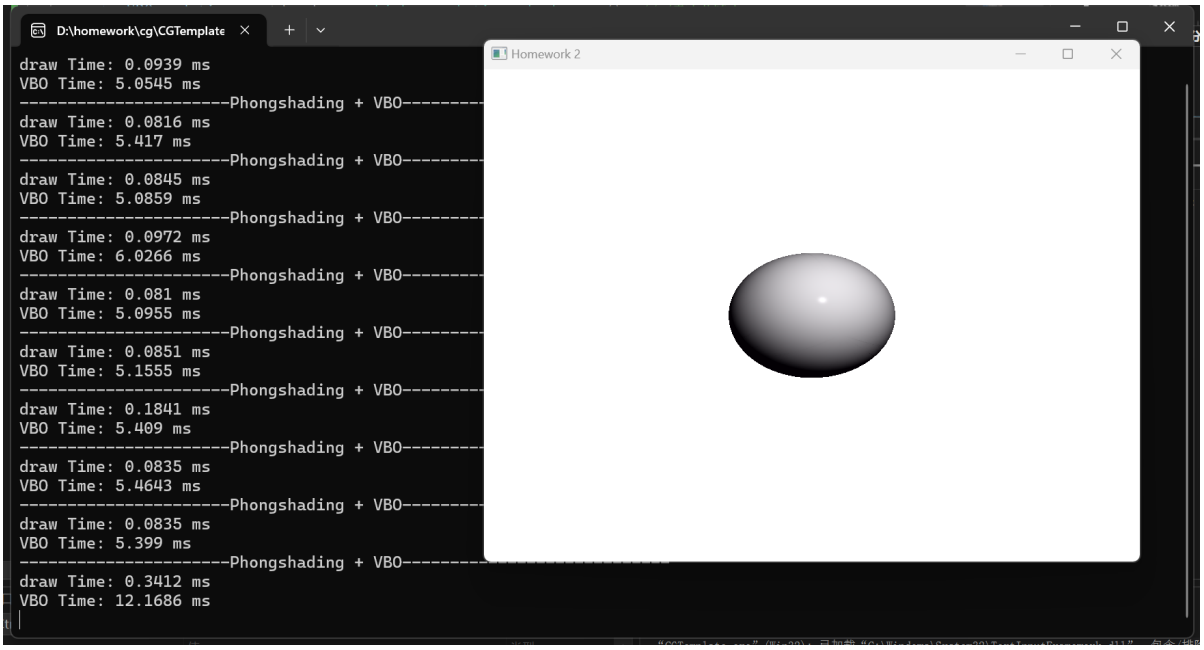
```
auto draw_start_time = std::chrono::high_resolution_clock::now();
// 执行绘制
glPushMatrix();
glBegin(GL_TRIANGLES);
```

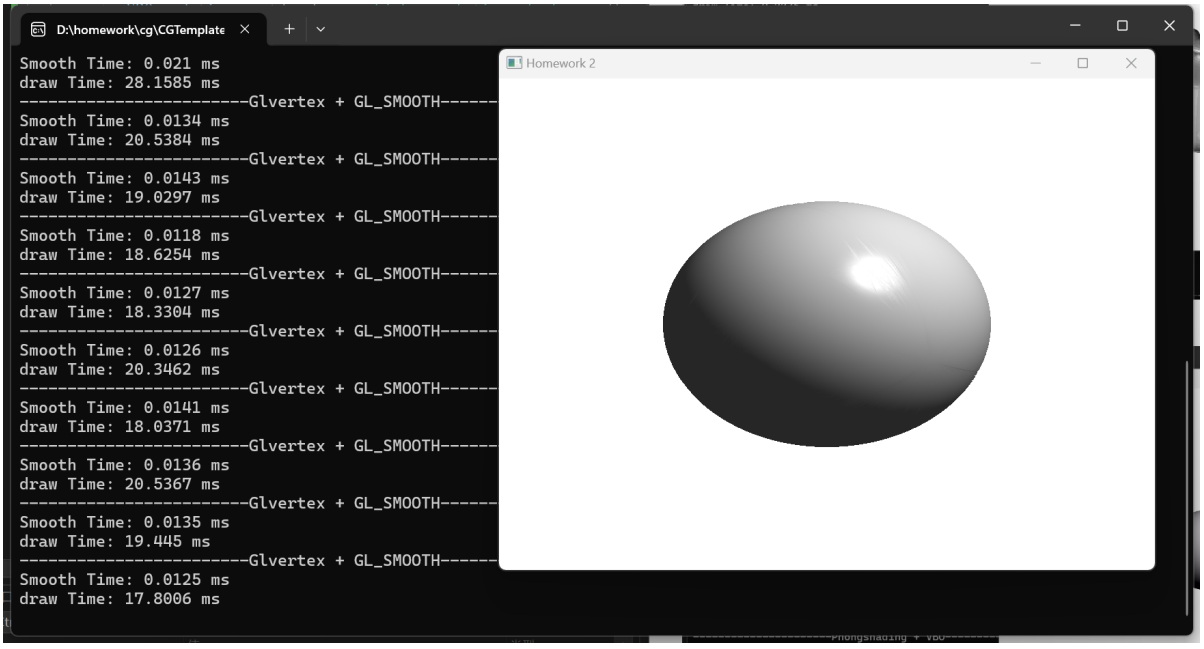
```
for (size_t i = 0; i < vertexData.size(); i += 6) {
    glColor3f(0.75f, 0.75f, 0.75f);
    glNormal3f(vertexData[i + 3], vertexData[i + 4], vertexData[i + 5]);
    glVertex3f(vertexData[i], vertexData[i + 1], vertexData[i + 2]);
}
glEnd();
glPopMatrix();
auto draw_end_time = std::chrono::high_resolution_clock::now(); // 结束时间
// 计算并输出算法的执行时间
std::chrono::duration<double, std::milli> drawTime = draw_end_time -
draw_start_time;
std::cout << "draw Time: " << drawTime.count() << " ms" << std::endl;
```

我们同样使用上面的运行结果，多次切换：









从图上我们可以看到，对比draw time，从平均绘画时间上说，glVertex的绘制时间普遍比VBO长。尤其当面数越多时，差距越明显。

从理论上来讲，由于VBO数据存储在GPU显存中，代码通常更复杂，一旦上传数据，**重复渲染时不需要再次传输数据**，减少了CPU到GPU的数据传输，提高了性能。因此它适合于复杂模型和实时渲染，尤其是在需要频繁渲染大量顶点数据的情况下。而glVertex代码相对简单直观，**但每次渲染都需要将数据从CPU传输到GPU**，这在数据量较大时会导致显著的性能下降。适合于数据量小、渲染不频繁的场景，或者在性能要求不高的简单应用中。所以从结论上说VBO在性能上通常优于glVertex算法。

3.3 讨论 VBO 中是否使用 index array 的效率区别

Index array

索引数组（Index Array）是用于高效绘制图形的一种数据结构。它包含了一系列的整数，每个整数都是一个索引，指向顶点数组（Vertex Array）中的一个特定顶点。它允许定义多个几何形状（如三角形）重用相同的顶点数据，而不是为每个形状发送完整的顶点数据。这种方法可以减少数据传输量，提高渲染效率，尤其是在处理大型模型或复杂场景时。下面是需要在原VBO代码中修改的地方：

初始化：

```
void MyGLWidget::initVBOVAO() {
    // 生成并绑定VAO
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // 生成并绑定VBO
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    // 生成并绑定EBO
    glGenBuffers(1, &ebo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
}
```

填充索引数组：

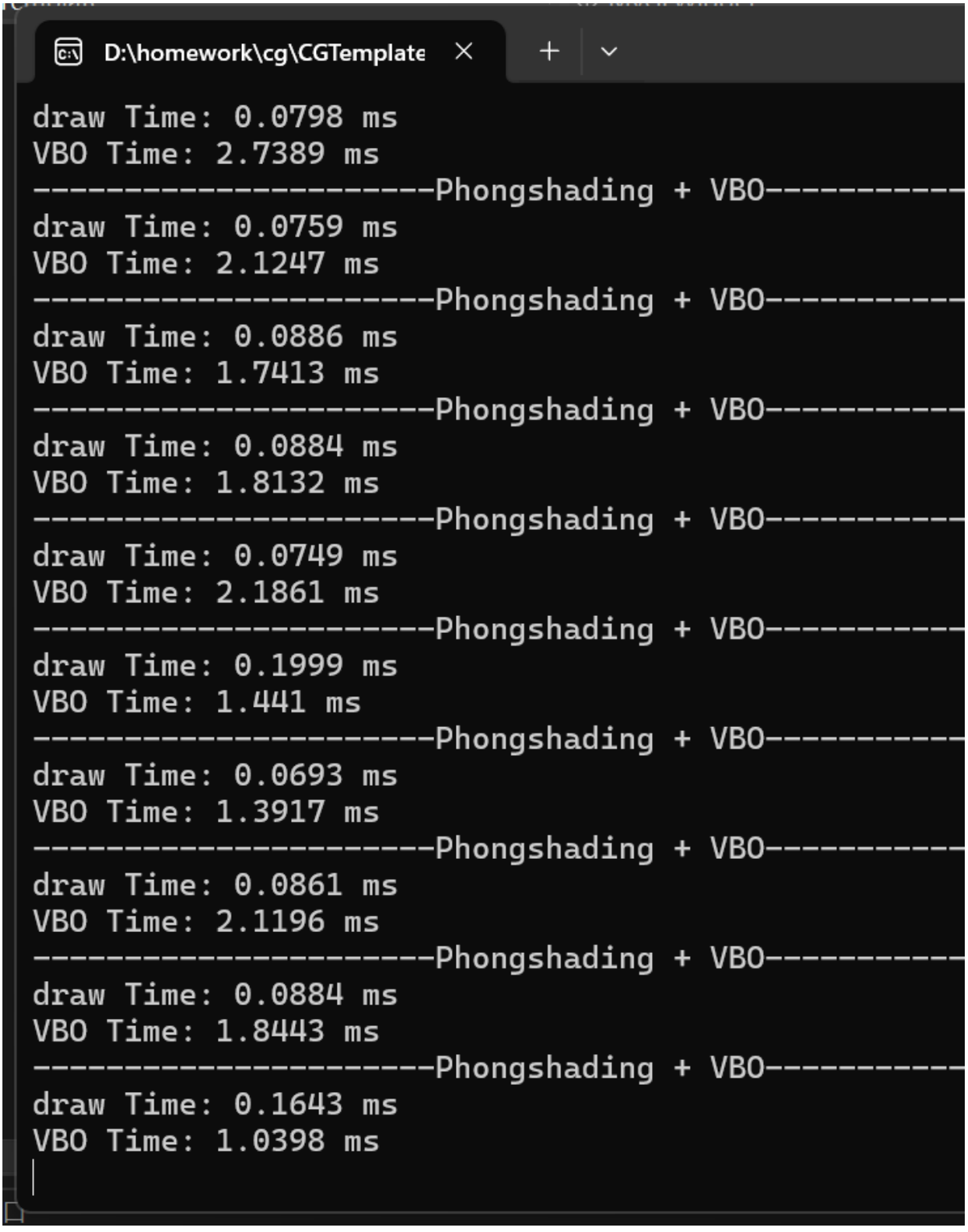
```
class MyGLWidget::update_VBO(){
    // 将顶点数据上传到 VBO
    // ...

    if (4 <= draw_id <= 5) {//
        // 填充索引数组
        for (int i = 0; i < objModel.triangleCount; ++i) {
            for (int j = 0; j < 3; ++j) {
                indices.push_back(static_cast<GLuint>(i * 3 + j));
            }
        }
    }

    // 上传索引数据到EBO
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint),
indices.data(), GL_STATIC_DRAW);

    // ...接着下面的代码

    // 执行绘制，使用glDrawElements而不是glDrawArrays
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    //...
}
```

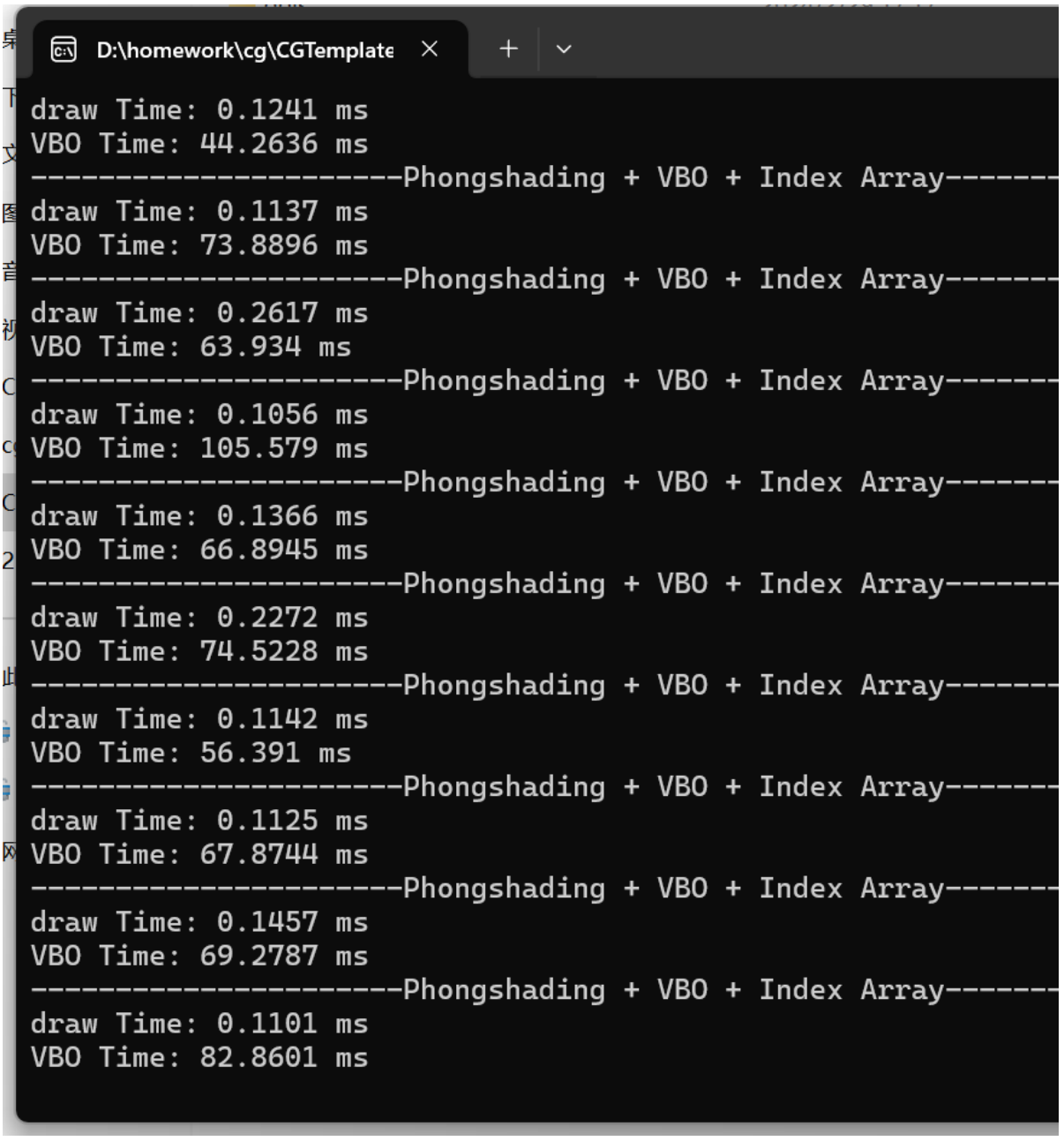
使用后需要解绑。

在小球里需要生成索引：

```
void MyGLWidget::Ball(int d) {
    // ...
    // 生成索引
    for (int i = 0; i < triangles.size(); i += 3) {
        indices.push_back(i);
        indices.push_back(i + 1);
        indices.push_back(i + 2);
    // ...
}
```

效率对比如下： 茶壶：


D:\homework\cg\CGTemplate
draw Time: 0.0798 ms
VBO Time: 2.7389 ms
----------Phongshading + VBO----------
draw Time: 0.0759 ms
VBO Time: 2.1247 ms
----------Phongshading + VBO----------
draw Time: 0.0886 ms
VBO Time: 1.7413 ms
----------Phongshading + VBO----------
draw Time: 0.0884 ms
VBO Time: 1.8132 ms
----------Phongshading + VBO----------
draw Time: 0.0749 ms
VBO Time: 2.1861 ms
----------Phongshading + VBO----------
draw Time: 0.1999 ms
VBO Time: 1.441 ms
----------Phongshading + VBO----------
draw Time: 0.0693 ms
VBO Time: 1.3917 ms
----------Phongshading + VBO----------
draw Time: 0.0861 ms
VBO Time: 2.1196 ms
----------Phongshading + VBO----------
draw Time: 0.0884 ms
VBO Time: 1.8443 ms
----------Phongshading + VBO----------
draw Time: 0.1643 ms
VBO Time: 1.0398 ms



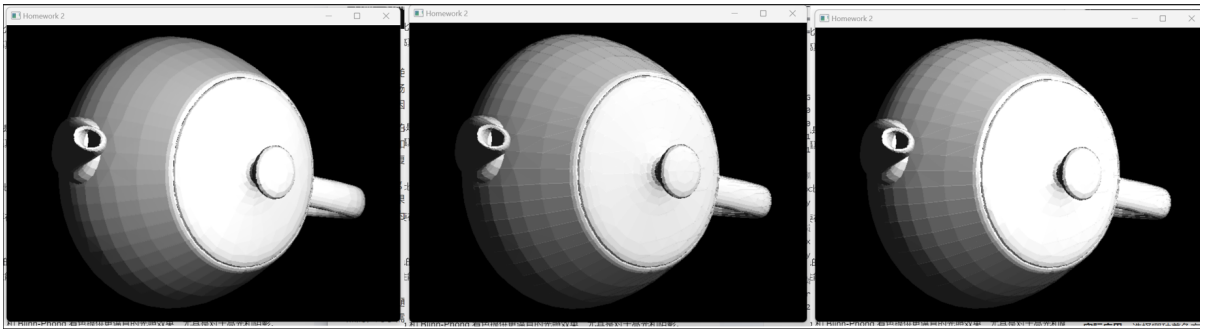
理论上来说，使用索引数组可以提高渲染效率，减少绘制调用，优化缓存使用，并减少带宽需求。通过上面draw time的对比，我们可以看到使用索引数组会对效率有显著提升效果。

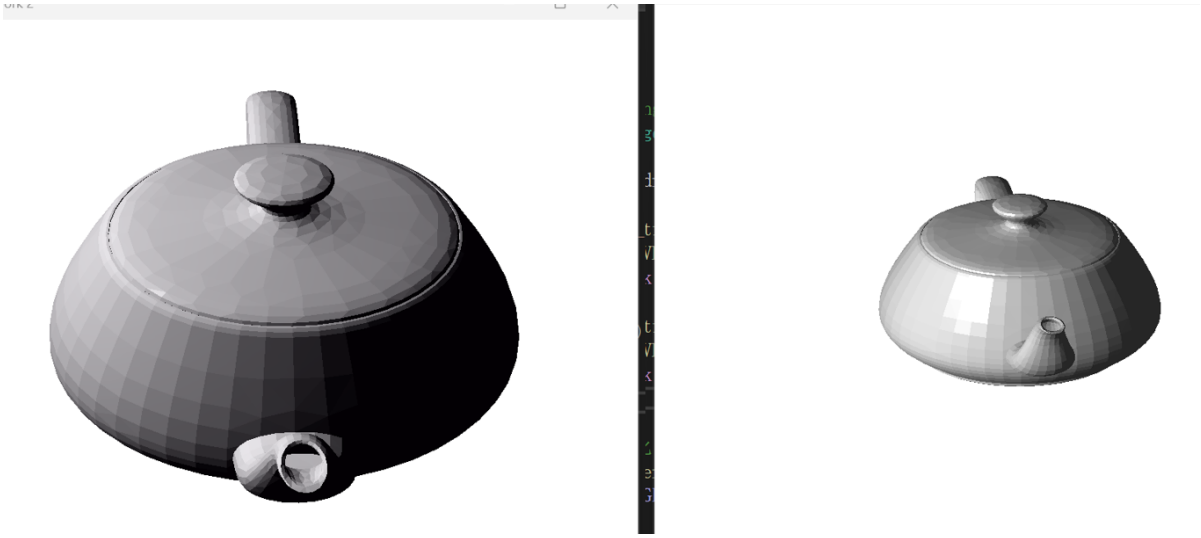
（因为此时讨论index array对渲染效率的影响。VBO这里的时间包括初始化、上传数据等流程，所以这里不参考）

3.4 对比、讨论 HW3 和 HW2 的渲染结果、效率的差别。

渲染效果：

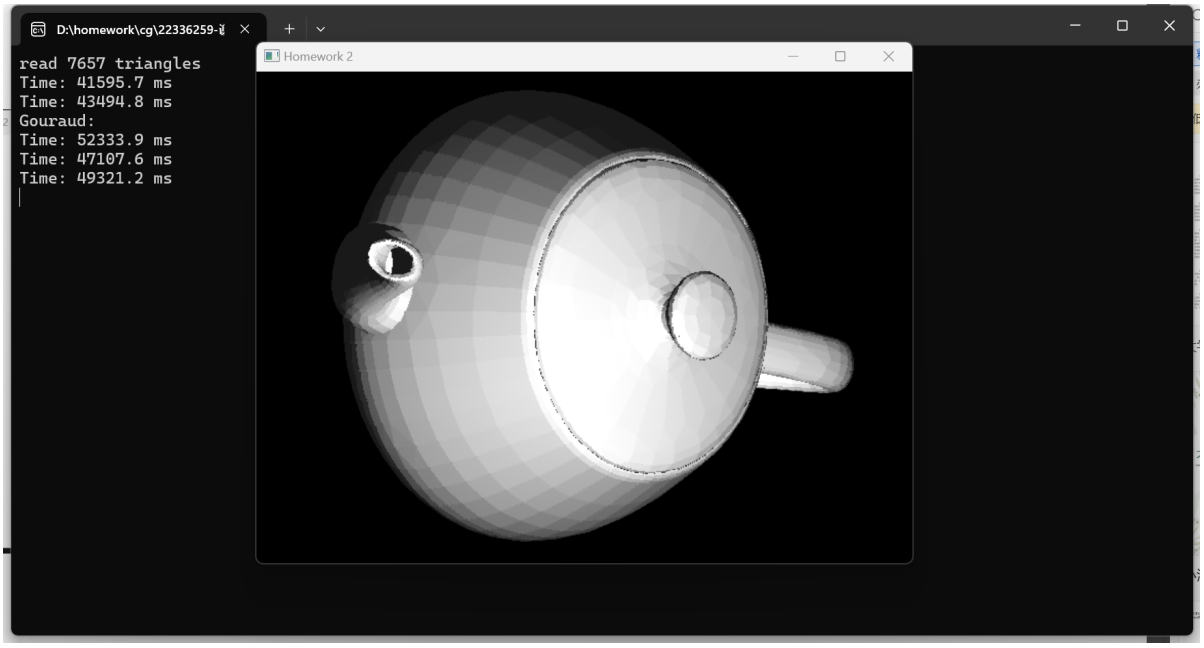
HW2





可以看到，HW2无论是哪种方法，其渲染效果都不如HW3好。因为HW3使用了着色器，渲染结果更加丰富，能够实现更复杂的光照效果和材质效果。而HW2受限于固定管线的能力，无法实现与着色器相同的光照和材质效果，所以渲染效果较差。

而从效率来说更加明显。因为我是轻薄本，HW2运行8000面茶壶非常困难。电脑风扇声音非常大，时间也很久，用gouraud最快的方法也需50000ms左右。而从上面的报告可以看到HW3运行一个8000面茶壶只需不到100ms，快的2ms就可以生成。



所以我们可以得出，无论是从渲染效果还是渲染效率来说，HW3的方法都强于HW2。