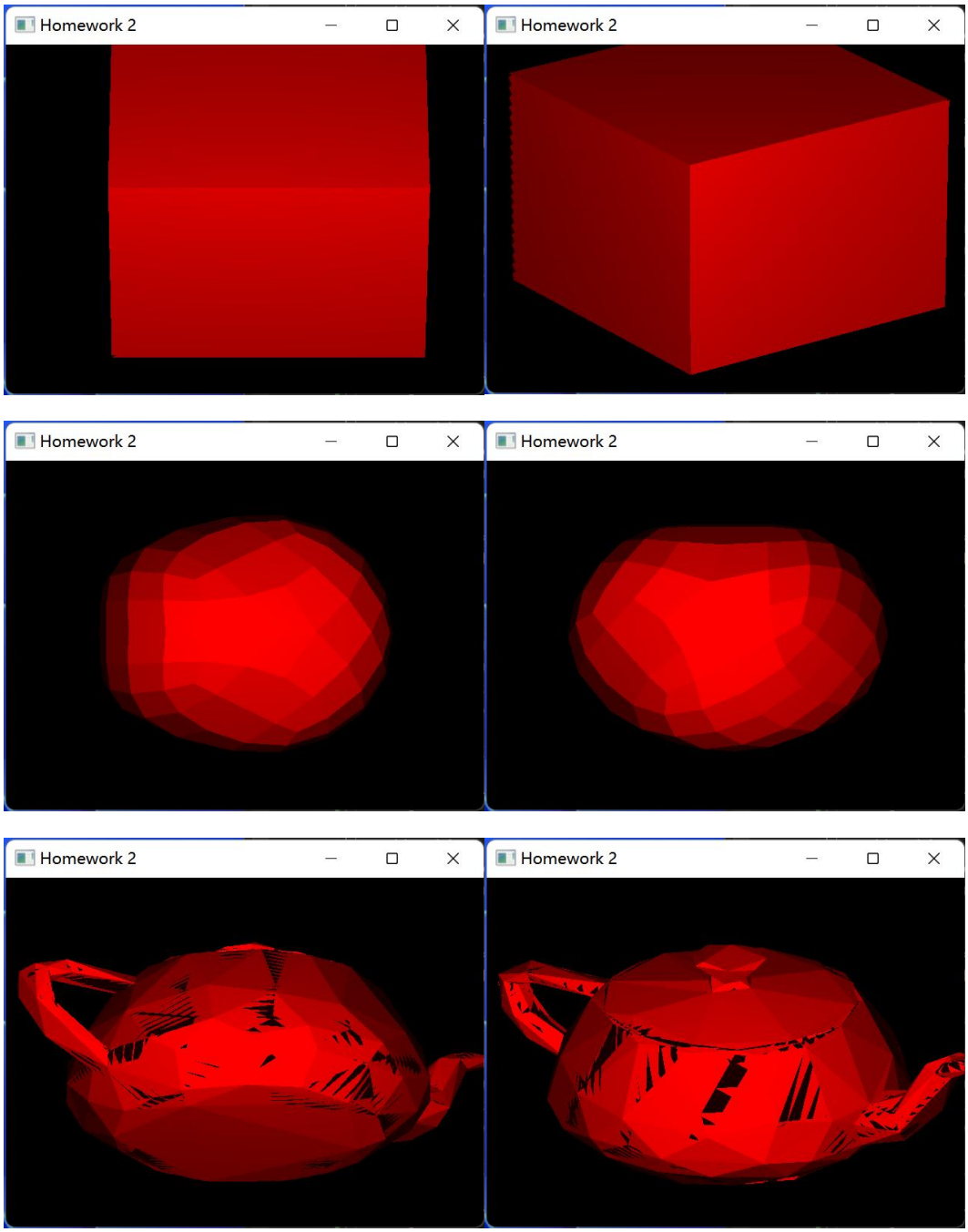
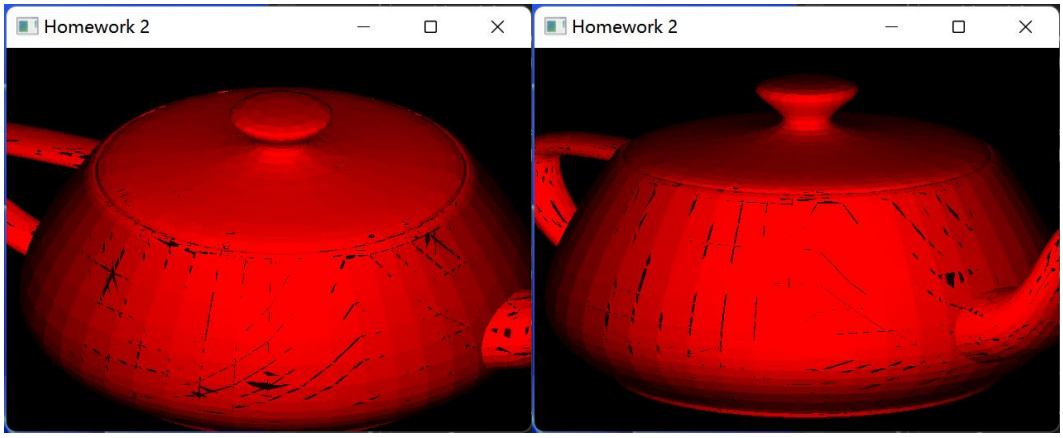
计算机图形学 作业三

1、Phong shading 与 VBO 绘制三维物体

绘制效果如下所示





具体代码如下所示：

```
// 选择要加载的模型
objModel.loadModel("./objs/teapot_8000.obj");

// 自主设置变换矩阵

// 相机位置
camPosition = vec3(100 * sin(degree * 3.14 / 180.0) + objModel.centralPoint.x, 100 * cos(degree
* 3.14 / 180.0) + objModel.centralPoint.y, 100 + objModel.centralPoint.z);
// 增大相机到物体的距离
camPosition2 = vec3(1000 * sin(degree * 3.14 / 180.0) + objModel.centralPoint.x,
    1000 * cos(degree * 3.14 / 180.0) + objModel.centralPoint.y,
    1000 + objModel.centralPoint.z);

// 相机所看向的点
camLookAt = objModel.centralPoint;     // 例如，看向物体中心
// 相机的上方向
camUp = vec3(0, 1, 0);          // 上方向向量
// 用于生成透视投影矩阵
projMatrix = glm::perspective(radians(20.0f), 1.0f, 0.1f, 2000.0f);

// 单一点光源，可以改为数组实现多光源
lightPosition = objModel.centralPoint + vec3(30, 30, 30); // 光源的位置
// 输出光源位置
std::cout << "centralPoint Position: (" << objModel.centralPoint.x << ", " <<
objModel.centralPoint.y << ", " << objModel.centralPoint.z << ")" << std::endl;
std::cout << "nowTriangle Position: (" << objModel.getTriangleByID(0).triangleVertices[0].x << ",
" << objModel.getTriangleByID(0).triangleVertices[0].y << ", " <<
objModel.getTriangleByID(0).triangleVertices[0].z << ")" << std::endl;

// 顶点数
float vertices[30000 * 6];

int index = 0;
for (int i = 0; i < objModel.triangleCount; i++) {
    Triangle nowTriangle = objModel.getTriangleByID(i);

    // 遍历每个顶点和法线
    for (int j = 0; j < 3; j++) {
        // 计算在数组中的索引

        // 输入顶点数据
        vertices[index] = nowTriangle.triangleVertices[j].x;
        vertices[index + 1] = nowTriangle.triangleVertices[j].y;
        vertices[index + 2] = nowTriangle.triangleVertices[j].z;

        // 输入法线数据
        vertices[index + 3] = nowTriangle.triangleNormals[j].x;
        vertices[index + 4] = nowTriangle.triangleNormals[j].y;
        vertices[index + 5] = nowTriangle.triangleNormals[j].z;
        index += 6;
    }
}
```

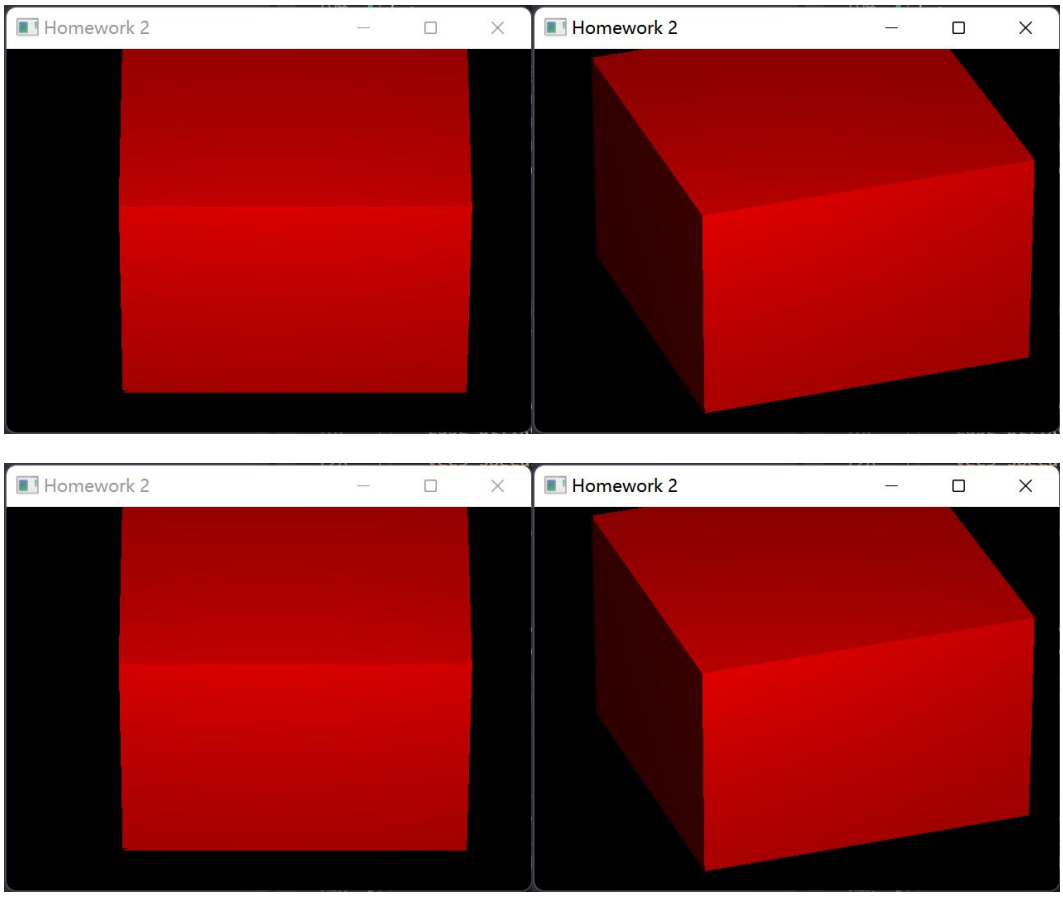
```
// 创建并绑定顶点缓冲对象
vbo.create();
vbo.bind();
vbo.allocate(vertices, sizeof(float) * index);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// 绑定着色器程序
shaderProgram.bind();

// 创建并设置模型视图投影矩阵
QMatrix4x4 modelViewProjection;
// modelViewProjection.perspective(45.0f, width() / float(height()), 0.1f, 100.0f); // 设置透视
投影矩阵
modelViewProjection.perspective(glm::radians(180.0f), 1.0f, 0.1f, 5000.0f);
modelViewProjection.lookAt(QVector3D(camPosition2.x, camPosition2.y, camPosition2.z),
    QVector3D(camLookAt.x, camLookAt.y, camLookAt.z),
    QVector3D(camUp.x, camUp.y, camUp.z)); // 设置视图矩阵

QMatrix4x4 modelView;
modelView.lookAt(QVector3D(camPosition.x, camPosition.y, camPosition.z),
    QVector3D(camLookAt.x, camLookAt.y, camLookAt.z),
    QVector3D(camUp.x, camUp.y, camUp.z));
// 将模型视图投影矩阵传递给顶点着色器
shaderProgram.setUniformValue("modelViewProjection", modelViewProjection);
shaderProgram.setUniformValue("modelView", modelView);

lightPosition2 = QVector3D(lightPosition.x, lightPosition.y, lightPosition.z); // 光源位置
objectColor = QVector3D(1.0, 0.0, 0.0);    // 物体颜色（红色）
lightColor = QVector3D(1, 1, 1);     // 光照颜色（白色）
ambientColor = QVector3D(0.1, 0.1, 0.1);   // 环境光颜色
viewPos = QVector3D(camPosition.x, camPosition.y, camPosition.z); // 光源位置
// 将光源位置、物体颜色、光照颜色和环境光颜色传递给片段着色器
shaderProgram.setUniformValue("lightPosition2", lightPosition2);
shaderProgram.setUniformValue("objectColor", objectColor);
shaderProgram.setUniformValue("lightColor", lightColor);
shaderProgram.setUniformValue("ambientColor", ambientColor);

// 绑定顶点缓冲对象
vbo.bind();

// 获取顶点位置的属性位置

// 启用顶点位置和法线的属性数组
shaderProgram.enableAttributeArray(0);
shaderProgram.enableAttributeArray(1);

// 设置顶点位置和法线的属性数组指针
shaderProgram.setAttributeBuffer(0, GL_FLOAT, 0, 3, 6 * sizeof(float));
shaderProgram.setAttributeBuffer(1, GL_FLOAT, 3 * sizeof(float), 3, 6 * sizeof(float));

// 绘制三角形
auto start_time = std::chrono::high_resolution_clock::now();
glDrawArrays(GL_TRIANGLES, 0, objModel.triangleCount * 3 * 3);
auto end_time = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
// 输出时间差异，单位为毫秒

// 禁用属性数组
shaderProgram.disableAttributeArray(0);
shaderProgram.disableAttributeArray(1);

// 解绑顶点缓冲对象
vbo.release();
// 解绑着色器程序
shaderProgram.release();
```

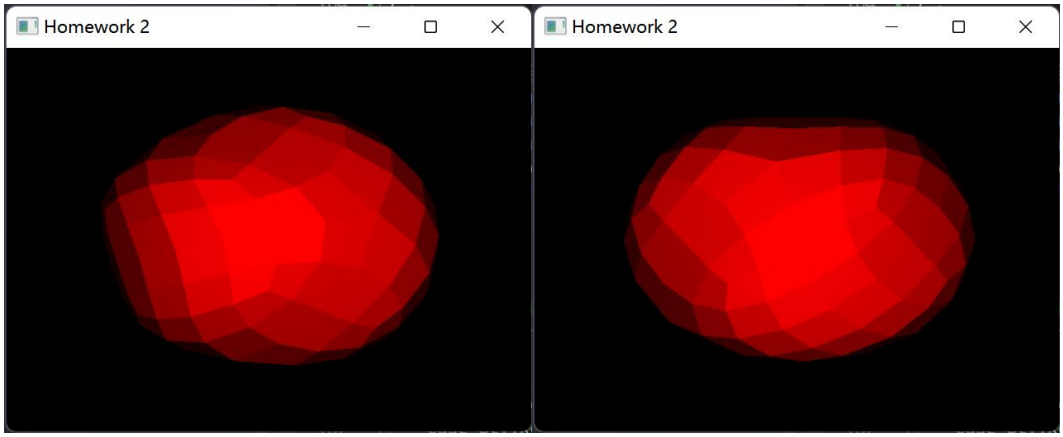
代码在作业 2 的基础上进行改进，我们首先加载一个模型，接着通过设置相机位置、观察点和上方向，以及生成透视投影矩阵，确定了场景的视角和投影方式。然后定义了光源的位置，并输出了光源位置和模型中某个三角形的位置。随后，创建了一个包含顶点和法线数据的数组，并上传到 OpenGL 顶点缓冲对象 VBO 中。在渲染过程中，设置了模型视图投影矩阵、光照和材质属性，包括光源位置、物体颜色、光照颜色和环境光颜色。最后，启用顶点

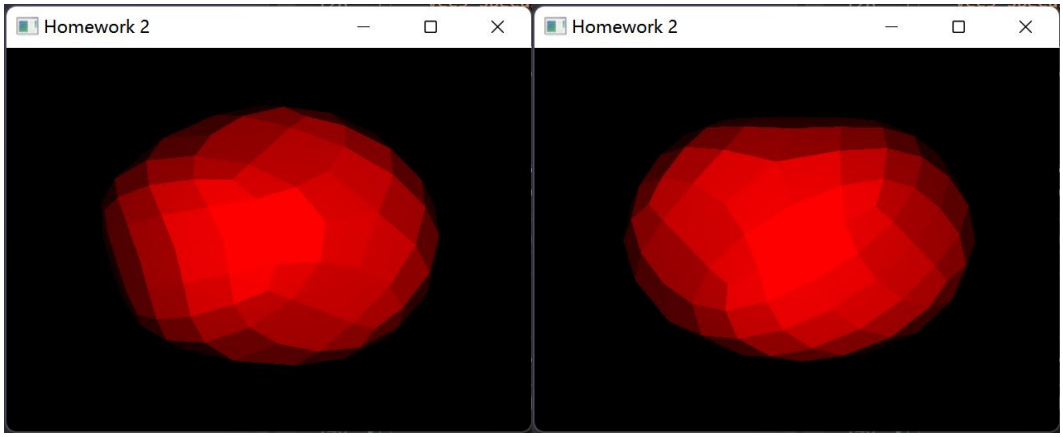
属性数组，使用 OpenGL 函数 glDrawArrays 绘制三角形，并在绘制完成后禁用属性数组、解绑顶点缓冲对象和着色器程序。

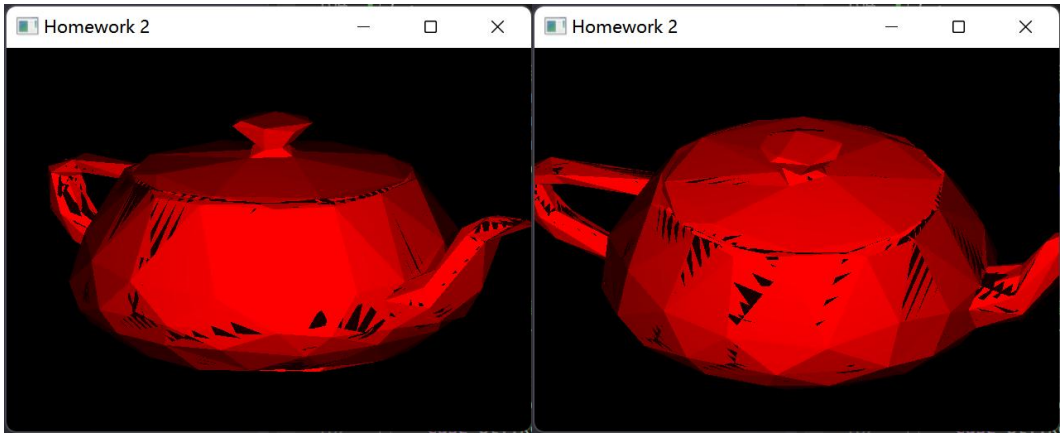
2、对比 Phong shading 与 OpenGL 自带的 smoothing shading 的区别；

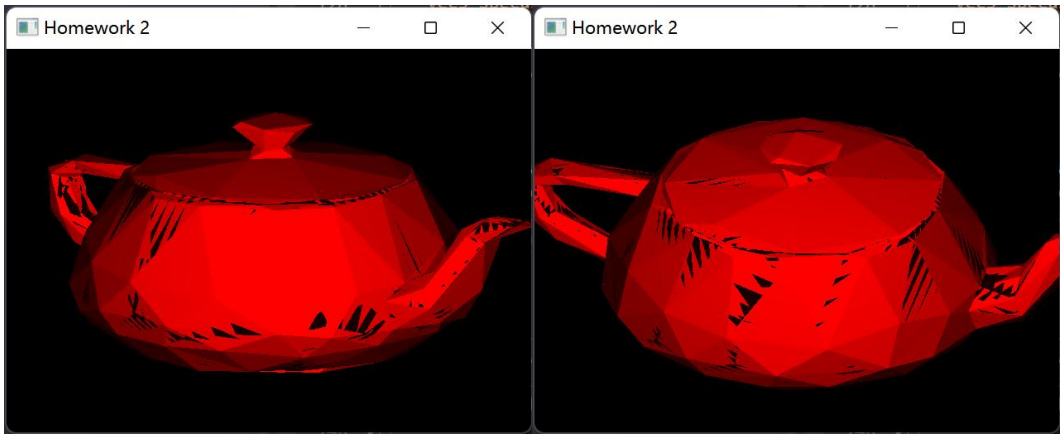
下面对两种算法进行对比：
位于上面一行的是 Phong shading
位于下面一行的是 smoothing shading

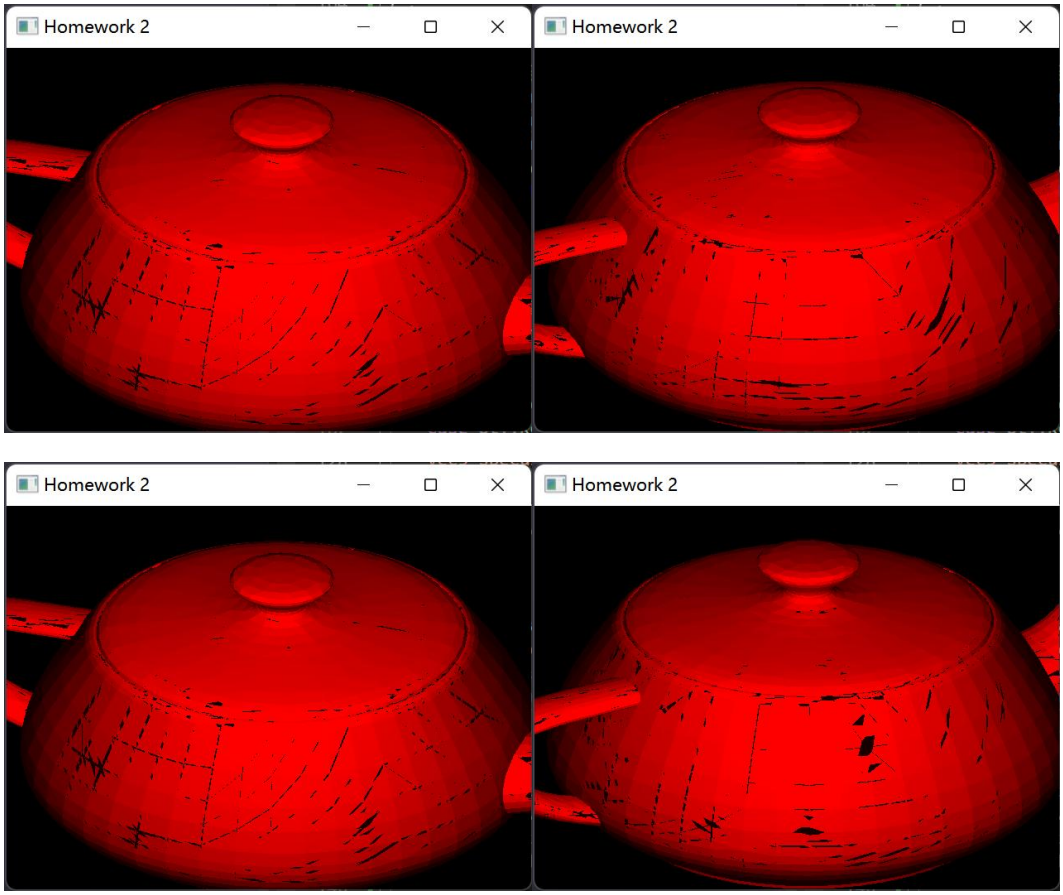



Homework 2
Homework 2


Homework 2
Homework 2


Homework 2
Homework 2


Homework 2
Homework 2



整体来看二者的渲染差异并不是非常大，同时这里因为截图后画面信息有一些丧失，在显示屏中如果将显示屏幕亮度调到最高，可以看出 smoothing shading 在高光处效果更差，smoothing shading 高光处容易完全变成一样的颜色，导致完全无法看出物体表面的信息，而 Phong Shading 还是能够看出物体的外形轮廓的，不同平面的光照效果会更接近现实。这种差异的主要原因可能是 Phong Shading 包含包括环境光、漫反射和镜面反射三个分量，考虑了物体表面的镜面高光反射，可以呈现出更明显的高光部分，使物体看起来更有光泽感。而 smoothing Shading 更加平滑，没有明显的高光反射，对于表面光滑的物体效果不好，可能更适合渲染表面比较粗糙的物体。

3、使用 VBO 进行绘制及通过 glVertex 进行绘制的区别。

	glVertex	VBO
teapot_600	5976ms	47us
rock	3153ms	45us
cube	235ms	36us
teapot_8000	无法绘制	45us

可以看到使用 VBO 之后绘制速度大幅提升，原本无法绘制的 teapot_8000 也能够在非常短的时间内画出来，相比 glVertex ，VBO 绘制不同模型的时间差异并不大，说明实验中还远未达到性能瓶颈，主要的耗时可能来自于数据的传输等操作。

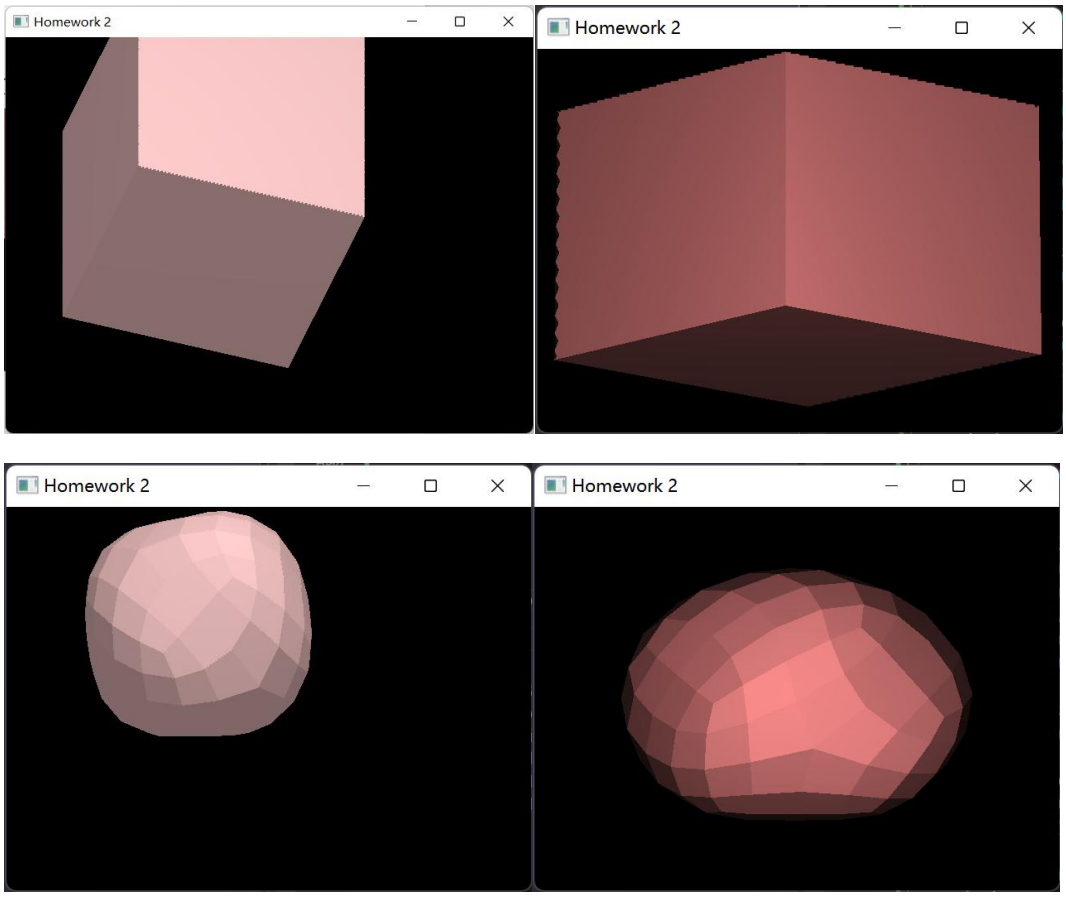
4、讨论 VBO 中是否使用 index array 的效率区别。

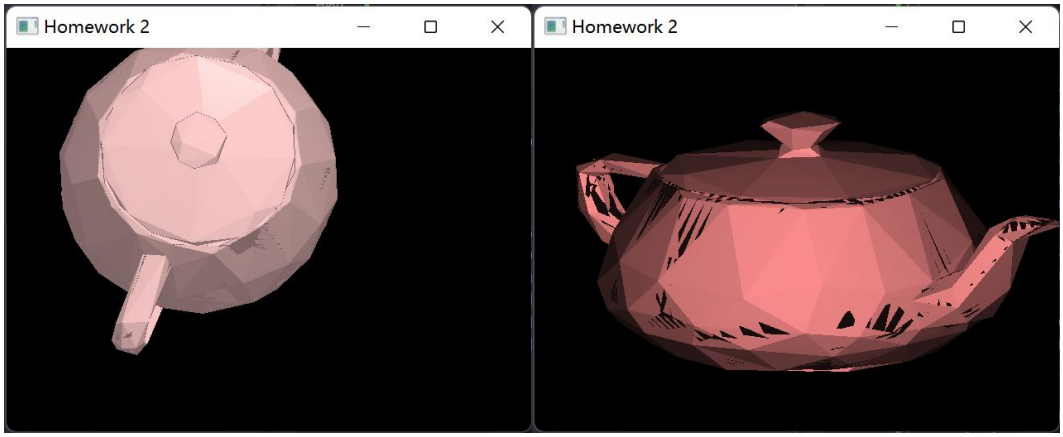
由于模型比较小，所以实验中我们比较难测量出使用 index array 后的耗时和没有使用 index array 的耗时之间的差异，二者的绘制速度都非常快，不过我们可以从理论上进行一些分析。Index Buffer 通过使用索引缓冲对象，绘制时可以指定一个索引数组，这样只需要传输一组索引，而不是重复传输相同的顶点数据。这在场景中有很多共享顶点的情况下能够显著减少数据传输量。同时如果模型包含许多重复的顶点数据，使用索引缓冲对象可以节省内存，因为只需要存储一份顶点数据，并通过索引引用它们。另外索引缓冲对象允许 GPU 按索引顺序访问顶点数据，这有助于提高缓存的命中率，减少内存访问的随机性。而如果不使用索引缓冲对象，那么在每次渲染时都需要传输所有的顶点数据，这可能导致冗余的数据传输，特别是对于重叠的顶点。每个顶点的数据都需要单独存储，而没有复用共享的顶点数据，这可能导致占用更多的内存。同时由于没有索引缓冲对象，GPU 可能需要随机访问顶点数据，这可能导致缓存性能的降低。

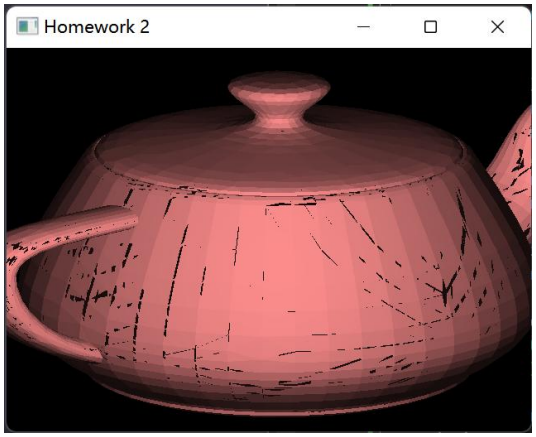
所以在大多数情况下，使用索引缓冲对象可以减少数据传输量、节省内存，并有助于提高缓存性能。然而，对于简单的模型或者数据量较小的情况，这种优势可能并不明显。在实际中我们需要综合考虑内存占用、数据传输和缓存性能等因素，选择是否使用索引缓冲对象。

5、对比、讨论 HW3 和 HW2 的渲染结果、效率的差别。

下面左边是 HW2，右边是 HW3







可以看出 HW3 使用 VBO 的效果中，图形的边缘绘制的没有 HW2 中用 edge walking 得到的效果好，主要原因可能是我们没有启用 VBO 的抗锯齿，图形的边缘显示出锯齿状，导致整体效果不如 HW2，如果启用抗锯齿可以平滑边缘，提高图形质量。刚才我们已经对比了使用 VBO 进行绘制和通过 glVertex 进行绘制的效率区别，在 HW2 中我们的设备无法支持绘制 teapot_8000，但是 HW3 中我们可以非常快的进行绘制。另一方面在 HW2 中，我们是无法处理模型图像超出左右边缘的情况的，因为这需要在 edge walking 中做出额外的控制条件才行，但是使用 VBO 我们则不用担心物体的图像超出左右边缘，底层函数会对其自动进行处理。整体来看使用 VBO 绘制效率高了非常多，使用可编程着色器后有更大的自由度来定义自己的渲染算法和效果，还可以更灵活地组织和管理顶点数据，通过灵活的数据组织方式轻松存储不同属性的顶点数据，比如位置、法线、纹理坐标等。可以看出 VBO 是现代图形学中的一项重要改变，其提供了更高效、更灵活的顶点数据管理方式，并与可编程着色器等现代图形学技术协同工作，使得我们能够更好地控制图形渲染流程，在处理更加复杂和真实的场景，同时获得更好的性能。