**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

**ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи №4

# на тему: “Проектування статичних тривимірних об’єктів на екран монітора за допомогою графічних бібліотек OpenGL”

# з дисципліни: *“*Засоби програмування комп'ютерної графіки*”*

**Лектор:**

проф. кафедри ПЗ

Журавчак Л. М.

**Виконав:**

студент групи ПЗ-33

Шлом’як Д. В.

**Прийняв:**

проф. кафедри ПЗ

Журавчак Л. М.

Львів – 2025

# Тема: проектування статичних тривимірних об’єктів на екран монітора за допомогою графічних бібліотек OpenGL.

**Мета:** навчитися будувати тривимірні об’єкти, проектувати їх на екран монітора, застосовуючи функції OpenGL.

**Теоретичні відомості**

Для задання різних перетворень об’єктів сцени в OpenGL використовують операції над матрицями, при цьому розрізняють кілька типів матриць, зокрема вигляду і проекції. Перша матриця визначає перетворення об’єкта у світових координатах, такі як паралельний перенос, зміна масштабу і поворот. Матриця проекцій задає, як будуть проектуватися тривимірні об’єкти на площину екрана (у віконні координати). Для того, щоб вибрати матрицю для опрацювання, використовують команду glMatrixMode(mode), виклик якої зі значенням параметра GL\_MODELVIEW чи GL\_PROJECTION вмикає режими роботи з матрицею виду чи проекцій відповідно.

**Робота з матрицями проекції та вигляду**

Отож, щоб працювати з матрицею проекції, треба активувати відповідний режим командою glMatrixMode(GL\_PROJECTION). Параметри ортогональної проекції вибирають за допомогою функції glOrtho(…), для отримання перспективної проекції сцени використовують функцію gluPerspective(…) або glFrustum(…). Якщо команда проектування не задана явно, OpenGL виконує ортогональну проекцію сцени за замовчуванням. Параметри спостереження задають за допомогою функції gluLookAt(…).

Перед роботою з об’єктами необхідно задати режим роботи з матрицею вигляду, виконавши команду glMatrixMode(GL\_MODELVIEW). Прямокутну область виведення результатів визначають функцією glViewport(…).

**Тривимірні примітиви**

Для створення тривимірних примітивів використовують відповідні функції додаткових бібліотек OpenGL – gluSphere(), gluCylinder(), gluDisk() із GLU, glutSolidSphere(), glutSolidCube(), glutSolidCone() тощо із GLUT, auxSolidCube(), auxSolidBox(), auxSolidTorus(), auxSolidCylinder(), auxSolidCone() тощо із GLAUX і т.д. Для коректної побудови цих об’єктів необхідно видаляти невидимі лінії та поверхні, увімкнувши відповідний режим командою glEnable(GL\_DEPTH\_TEST).

**Освітлення**

Важливою деталлю у тривимірних сценах є освітлення: без нього об’єкти здаватимуться плоскими. За замовчуванням освітлення вимкнуто, його потрібно увімкнути командою glEnable(GL\_LIGHTING). Джерело освітлення вмикають командою glEnable(GL\_LIGHTn), де n може набувати значень від 0 до 7, та задають його властивості за допомогою функції glLight(…). Джерел можна використовувати кілька.

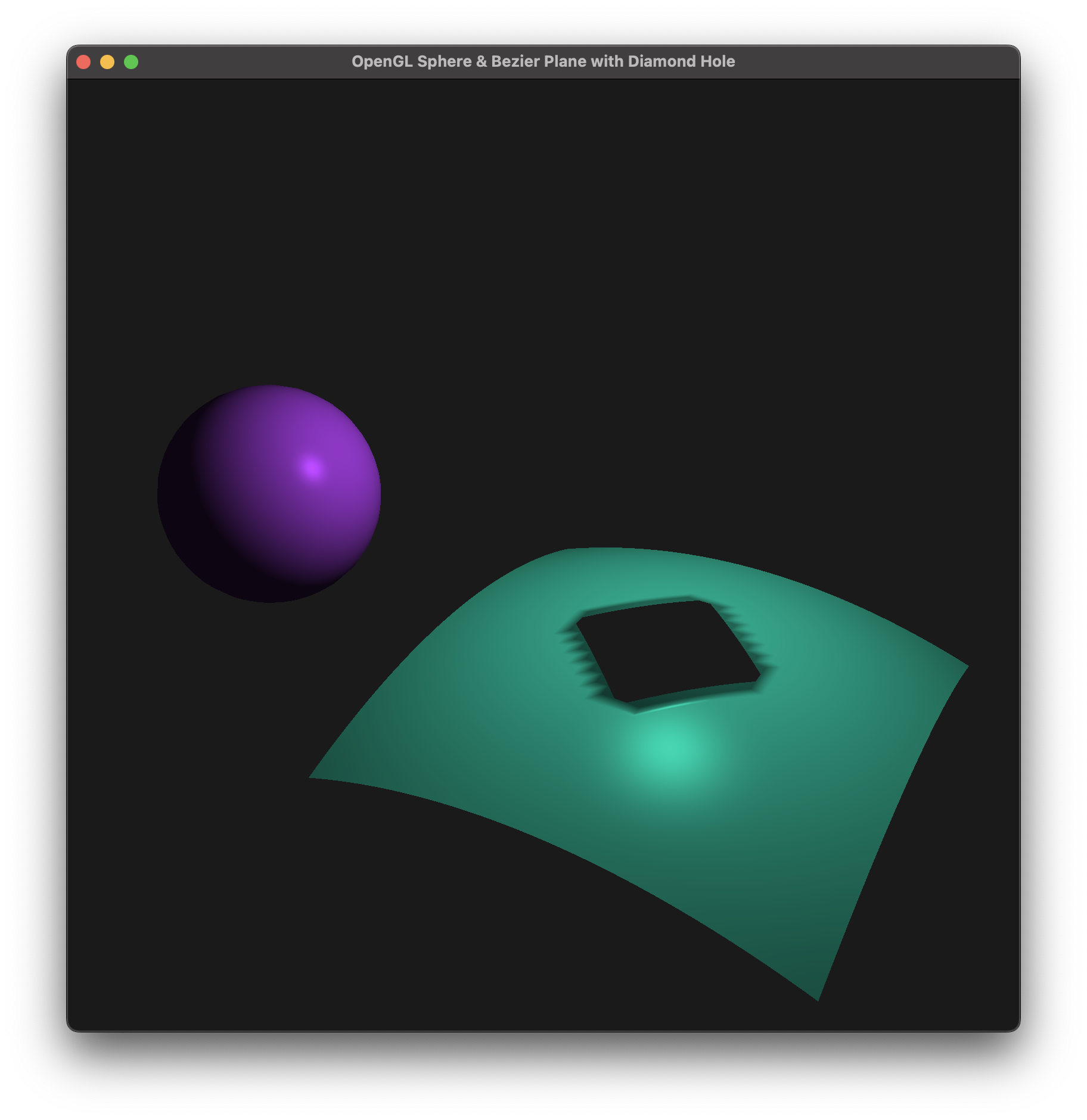
**Завдання**

10. Згенерувати за допомогою функцій бібліотеки GLU сферу. У поверхні Без’є вирізати отвір у формі ромба.

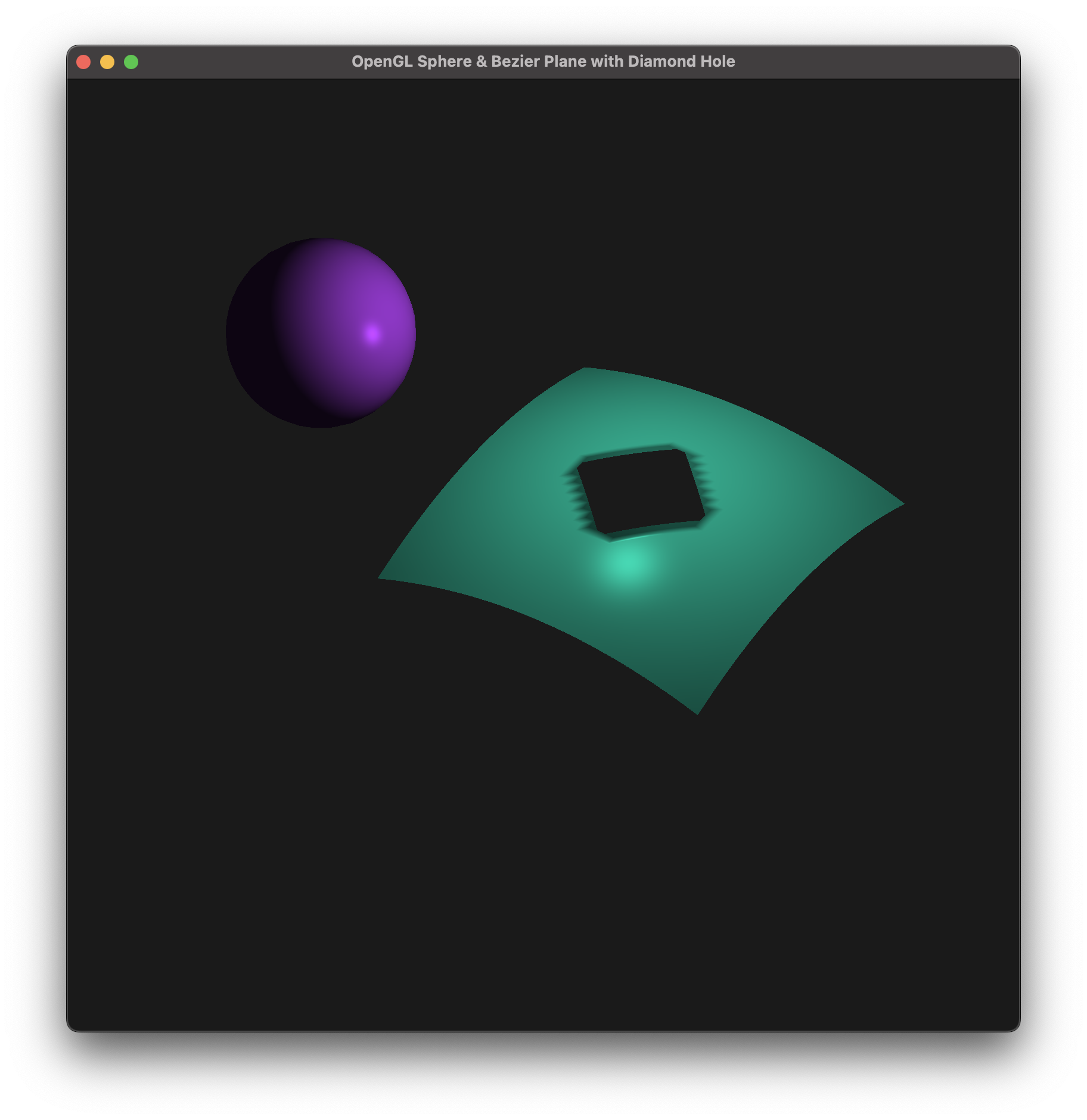
**Код програми**

#include "GL/glew.h"  
#include "GLFW/glfw3.h"  
#include <glm/glm.hpp>  
#include <glm/gtc/matrix\_transform.hpp>  
#include <glm/gtc/type\_ptr.hpp>  
#include <iostream>  
#include <cmath>  
#include <map>  
#include <vector>  
  
const int SCR\_WIDTH = 800;  
const int SCR\_HEIGHT = 800;  
  
const char\* vertexShaderSource = R"(  
 #version 330 core  
 layout (location = 0) in vec3 aPos;  
 layout (location = 1) in vec3 aNormal;  
  
 out vec3 FragPos;  
 out vec3 Normal;  
  
 uniform mat4 projection;  
 uniform mat4 view;  
 uniform mat4 model;  
  
 void main() {  
 FragPos = vec3(model \* vec4(aPos, 1.0));  
 Normal = mat3(transpose(inverse(model))) \* aNormal;  
  
 gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);  
 }  
)";  
  
const char\* fragmentShaderSource = R"(  
 #version 330 core  
 out vec4 FragColor;  
  
 in vec3 FragPos;  
 in vec3 Normal;  
  
 uniform vec3 lightPos;  
 uniform vec3 viewPos;  
 uniform vec3 lightColor;  
 uniform vec3 objectColor;  
  
 void main() {  
 // Ambient lighting  
 float ambientStrength = 0.1;  
 vec3 ambient = ambientStrength \* lightColor;  
  
 // Diffuse lighting  
 vec3 norm = normalize(Normal);  
 vec3 lightDir = normalize(lightPos - FragPos);  
 float diff = max(dot(norm, lightDir), 0.0);  
 vec3 diffuse = diff \* lightColor;  
  
 // Specular lighting  
 float specularStrength = 0.5;  
 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);  
 }  
)";  
  
struct Camera {  
 glm::vec3 position;  
 glm::vec3 front;  
 glm::vec3 up;  
 float speed;  
 float yaw;  
 float pitch;  
 bool firstMouse;  
 float lastX, lastY;  
 float sensitivity;  
};  
  
Camera camera = {  
 glm::vec3(0.0f, 0.0f, 3.0f),  
 glm::vec3(0.0f, 0.0f, -1.0f),  
 glm::vec3(0.0f, 1.0f, 0.0f),  
 0.05f,  
 -90.0f,  
 0.0f,  
 true,  
 SCR\_WIDTH / 2.0f,  
 SCR\_HEIGHT / 2.0f,  
 0.1f // Setting sensitivity  
};  
  
bool usePerspective = true;  
float orthogonalSize = 5.0f;  
  
int binomialCoef(int n, int k);  
  
void mouse\_callback(GLFWwindow\* window, double xpos, double ypos) {  
 if (camera.firstMouse) {  
 camera.lastX = xpos;  
 camera.lastY = ypos;  
 camera.firstMouse = false;  
 }  
  
 float xoffset = xpos - camera.lastX;  
 float yoffset = camera.lastY - ypos;  
 camera.lastX = xpos;  
 camera.lastY = ypos;  
  
 xoffset \*= camera.sensitivity;  
 yoffset \*= camera.sensitivity;  
  
 camera.yaw += xoffset;  
 camera.pitch += yoffset;  
  
 if (camera.pitch > 89.0f)  
 camera.pitch = 89.0f;  
 if (camera.pitch < -89.0f)  
 camera.pitch = -89.0f;  
  
 glm::vec3 front;  
 front.x = cos(glm::radians(camera.yaw)) \* cos(glm::radians(camera.pitch));  
 front.y = sin(glm::radians(camera.pitch));  
 front.z = sin(glm::radians(camera.yaw)) \* cos(glm::radians(camera.pitch));  
 camera.front = glm::normalize(front);  
}  
  
// Bezier curve function: B(t) = (1-t)³P₀ + 3(1-t)²tP₁ + 3(1-t)t²P₂ + t³P₃  
glm::vec3 bezierCurve(float t, const glm::vec3& p0, const glm::vec3& p1, const glm::vec3& p2, const glm::vec3& p3) {  
 float u = 1.0f - t;  
 float tt = t \* t;  
 float uu = u \* u;  
 float uuu = uu \* u;  
 float ttt = tt \* t;  
  
 glm::vec3 point = uuu \* p0; // (1-t)³ \* P₀  
 point += 3.0f \* uu \* t \* p1; // 3(1-t)² \* t \* P₁  
 point += 3.0f \* u \* tt \* p2; // 3(1-t) \* t² \* P₂  
 point += ttt \* p3; // t³ \* P₃  
  
 return point;  
}  
  
void generateBezierPlane(std::vector<float>& vertices, std::vector<unsigned int>& indices,  
 int resolution = 20, float size = 2.0f) {  
 float step = 1.0f / (resolution - 1);  
 float holeSize = size \* 0.4f; // Diamond hole size  
  
 // Control points for the Bezier surface  
 glm::vec3 controlPoints[4][4] = {  
 { glm::vec3(-size, 0.0f, -size), glm::vec3(-size/3, 0.5f, -size), glm::vec3(size/3, 0.5f, -size), glm::vec3(size, 0.0f, -size) },  
 { glm::vec3(-size, 0.5f, -size/3), glm::vec3(-size/3, 1.0f, -size/3), glm::vec3(size/3, 1.0f, -size/3), glm::vec3(size, 0.5f, -size/3) },  
 { glm::vec3(-size, 0.5f, size/3), glm::vec3(-size/3, 1.0f, size/3), glm::vec3(size/3, 1.0f, size/3), glm::vec3(size, 0.5f, size/3) },  
 { glm::vec3(-size, 0.0f, size), glm::vec3(-size/3, 0.5f, size), glm::vec3(size/3, 0.5f, size), glm::vec3(size, 0.0f, size) }  
 };  
  
 // Specialized Bernstein polynomial calculator for cubic bezier (n=3)  
 auto bernstein3 = [](int i, float t) {  
 float u = 1.0f - t;  
  
 switch (i) {  
 case 0: return u \* u \* u; // (1-t)³  
 case 1: return 3.0f \* u \* u \* t; // 3(1-t)²t  
 case 2: return 3.0f \* u \* t \* t; // 3(1-t)t²  
 case 3: return t \* t \* t; // t³  
 default: return 0.0f;  
 }  
 };  
  
 // First pass: compute all surface points  
 std::vector<glm::vec3> surfacePoints(resolution \* resolution);  
 std::vector<bool> insideHole(resolution \* resolution, false);  
  
 for (int i = 0; i < resolution; i++) {  
 float u = i \* step;  
 for (int j = 0; j < resolution; j++) {  
 float v = j \* step;  
 int idx = i \* resolution + j;  
 glm::vec3 point(0.0f);  
  
 // Compute Bezier surface point using specialized formula  
 for (int ki = 0; ki <= 3; ki++) {  
 float bu = bernstein3(ki, u);  
 for (int kj = 0; kj <= 3; kj++) {  
 float bv = bernstein3(kj, v);  
 point += controlPoints[ki][kj] \* (bu \* bv);  
 }  
 }  
 surfacePoints[idx] = point;  
  
 // Strict diamond hole boundary using Manhattan distance  
 float manhattanDist = fabs(point.x) + fabs(point.z);  
 insideHole[idx] = (manhattanDist < holeSize);  
 }  
 }  
  
 // Second pass: identify boundary points  
 std::vector<bool> isBoundaryPoint(resolution \* resolution, false);  
  
 for (int i = 0; i < resolution; i++) {  
 for (int j = 0; j < resolution; j++) {  
 int idx = i \* resolution + j;  
 if (insideHole[idx]) continue;  
  
 // Check if this point is adjacent to a hole point  
 for (int di = -1; di <= 1; di++) {  
 for (int dj = -1; dj <= 1; dj++) {  
 if (di == 0 && dj == 0) continue;  
  
 int ni = i + di;  
 int nj = j + dj;  
  
 if (ni >= 0 && ni < resolution && nj >= 0 && nj < resolution) {  
 int neighborIdx = ni \* resolution + nj;  
 if (insideHole[neighborIdx]) {  
 isBoundaryPoint[idx] = true;  
 break;  
 }  
 }  
 }  
 if (isBoundaryPoint[idx]) break;  
 }  
 }  
 }  
  
 // Third pass: Calculate normals with special handling for boundary points  
 std::vector<glm::vec3> surfaceNormals(resolution \* resolution);  
  
 for (int i = 0; i < resolution; i++) {  
 for (int j = 0; j < resolution; j++) {  
 int idx = i \* resolution + j;  
 if (insideHole[idx]) continue;  
  
 // Calculate the tangent vectors  
 glm::vec3 tangentU, tangentV;  
  
 if (i == 0)  
 tangentU = surfacePoints[(i + 1) \* resolution + j] - surfacePoints[i \* resolution + j];  
 else if (i == resolution - 1)  
 tangentU = surfacePoints[i \* resolution + j] - surfacePoints[(i - 1) \* resolution + j];  
 else  
 tangentU = surfacePoints[(i + 1) \* resolution + j] - surfacePoints[(i - 1) \* resolution + j];  
  
 if (j == 0)  
 tangentV = surfacePoints[i \* resolution + j + 1] - surfacePoints[i \* resolution + j];  
 else if (j == resolution - 1)  
 tangentV = surfacePoints[i \* resolution + j] - surfacePoints[i \* resolution + j - 1];  
 else  
 tangentV = surfacePoints[i \* resolution + j + 1] - surfacePoints[i \* resolution + j - 1];  
  
 // Compute normal from tangents  
 if (glm::length(tangentU) < 0.0001f || glm::length(tangentV) < 0.0001f) {  
 surfaceNormals[idx] = glm::vec3(0.0f, 1.0f, 0.0f);  
 } else {  
 surfaceNormals[idx] = glm::normalize(glm::cross(tangentU, tangentV));  
  
 // Special handling for boundary points - direct normals toward hole center  
 if (isBoundaryPoint[idx]) {  
 glm::vec3 point = surfacePoints[idx];  
 // Vector from origin to point projected on XZ plane  
 glm::vec3 fromCenter = glm::normalize(glm::vec3(point.x, 0.0f, point.z));  
  
 // Blend normal with outward direction for sharp edge  
 glm::vec3 upVector = glm::vec3(0.0f, 1.0f, 0.0f);  
 glm::vec3 outwardNormal = glm::normalize(fromCenter + 0.5f \* upVector);  
  
 // Strong blend toward the outward direction for boundary points  
 surfaceNormals[idx] = glm::normalize(surfaceNormals[idx] \* 0.2f + outwardNormal \* 0.8f);  
 }  
 }  
 }  
 }  
  
 // Fourth pass: Create exact boundary vertices along the diamond edge  
 std::map<std::pair<int, int>, int> edgeVertexMap; // Maps edge indices to vertex indices  
 int vertexCount = 0;  
  
 // Add all non-hole vertices first  
 for (int i = 0; i < resolution; i++) {  
 for (int j = 0; j < resolution; j++) {  
 int idx = i \* resolution + j;  
 if (insideHole[idx]) continue;  
  
 // Add vertex  
 vertices.push\_back(surfacePoints[idx].x);  
 vertices.push\_back(surfacePoints[idx].y);  
 vertices.push\_back(surfacePoints[idx].z);  
 vertices.push\_back(surfaceNormals[idx].x);  
 vertices.push\_back(surfaceNormals[idx].y);  
 vertices.push\_back(surfaceNormals[idx].z);  
  
 edgeVertexMap[{i, j}] = vertexCount++;  
 }  
 }  
  
 // Generate triangles while properly handling the hole  
 for (int i = 0; i < resolution - 1; i++) {  
 for (int j = 0; j < resolution - 1; j++) {  
 // Get quad corner indices  
 std::pair<int, int> p00 = {i, j};  
 std::pair<int, int> p10 = {i+1, j};  
 std::pair<int, int> p01 = {i, j+1};  
 std::pair<int, int> p11 = {i+1, j+1};  
  
 // Get vertex indices, -1 if inside hole  
 int v00 = insideHole[i \* resolution + j] ? -1 : edgeVertexMap[p00];  
 int v10 = insideHole[(i+1) \* resolution + j] ? -1 : edgeVertexMap[p10];  
 int v01 = insideHole[i \* resolution + j+1] ? -1 : edgeVertexMap[p01];  
 int v11 = insideHole[(i+1) \* resolution + j+1] ? -1 : edgeVertexMap[p11];  
  
 // Count how many vertices are valid  
 int validCount = (v00 != -1 ? 1 : 0) + (v10 != -1 ? 1 : 0) +  
 (v01 != -1 ? 1 : 0) + (v11 != -1 ? 1 : 0);  
  
 // Only create triangles if we have 3 or 4 valid vertices  
 if (validCount == 4) {  
 // Standard quad triangulation  
 indices.push\_back(v00);  
 indices.push\_back(v10);  
 indices.push\_back(v01);  
  
 indices.push\_back(v01);  
 indices.push\_back(v10);  
 indices.push\_back(v11);  
 }  
 else if (validCount == 3) {  
 // Single triangle for a partial quad  
 if (v00 != -1 && v10 != -1 && v01 != -1) {  
 indices.push\_back(v00);  
 indices.push\_back(v10);  
 indices.push\_back(v01);  
 }  
 else if (v10 != -1 && v01 != -1 && v11 != -1) {  
 indices.push\_back(v10);  
 indices.push\_back(v11);  
 indices.push\_back(v01);  
 }  
 else if (v00 != -1 && v01 != -1 && v11 != -1) {  
 indices.push\_back(v00);  
 indices.push\_back(v11);  
 indices.push\_back(v01);  
 }  
 else if (v00 != -1 && v10 != -1 && v11 != -1) {  
 indices.push\_back(v00);  
 indices.push\_back(v10);  
 indices.push\_back(v11);  
 }  
 }  
 }  
 }  
}  
  
  
// Improved binomial coefficient calculation using an iterative approach  
int binomialCoef(int n, int k) {  
 // C(n,k) = C(n,n-k)  
 if (k > n - k)  
 k = n - k;  
  
 if (k == 0)  
 return 1;  
  
 // Calculate using multiplicative formula  
 int result = 1;  
 for (int i = 0; i < k; ++i) {  
 result \*= (n - i);  
 result /= (i + 1);  
 }  
  
 return result;  
}  
  
// Updated sphere generation to include normals for lighting  
void generateSphereVertices(std::vector<float>& vertices, float radius, unsigned int sectorCount, unsigned int stackCount) {  
 float x, y, z, xy;  
 float nx, ny, nz;  
 float sectorStep = 2 \* glm::pi<float>() / sectorCount;  
 float stackStep = glm::pi<float>() / stackCount;  
 float sectorAngle, stackAngle;  
  
 for (unsigned int i = 0; i <= stackCount; ++i) {  
 stackAngle = glm::pi<float>() / 2 - i \* stackStep;  
 xy = radius \* cosf(stackAngle);  
 z = radius \* sinf(stackAngle);  
  
 // Vertices of the current stack  
 for (unsigned int j = 0; j <= sectorCount; ++j) {  
 sectorAngle = j \* sectorStep;  
  
 x = xy \* cosf(sectorAngle);  
 y = xy \* sinf(sectorAngle);  
  
 // Normalized vertex normal (pointing outward from the sphere center)  
 nx = x / radius;  
 ny = y / radius;  
 nz = z / radius;  
  
 // Position  
 vertices.push\_back(x);  
 vertices.push\_back(y);  
 vertices.push\_back(z);  
  
 // Normal  
 vertices.push\_back(nx);  
 vertices.push\_back(ny);  
 vertices.push\_back(nz);  
 }  
 }  
}  
  
void generateSphereIndices(std::vector<unsigned int>& indices, unsigned int sectorCount, unsigned int stackCount) {  
 unsigned int k1, k2;  
 for (unsigned int i = 0; i < stackCount; ++i) {  
 k1 = i \* (sectorCount + 1);  
 k2 = k1 + sectorCount + 1;  
  
 for (unsigned int j = 0; j < sectorCount; ++j, ++k1, ++k2) {  
 if (i != 0) {  
 indices.push\_back(k1);  
 indices.push\_back(k2);  
 indices.push\_back(k1 + 1);  
 }  
  
 if (i != (stackCount - 1)) {  
 indices.push\_back(k1 + 1);  
 indices.push\_back(k2);  
 indices.push\_back(k2 + 1);  
 }  
 }  
 }  
}  
  
GLuint compileShader(GLenum type, const char\* source) {  
 GLuint shader = glCreateShader(type);  
 glShaderSource(shader, 1, &source, nullptr);  
 glCompileShader(shader);  
  
 GLint success;  
 glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);  
 if (!success) {  
 char infoLog[512];  
 glGetShaderInfoLog(shader, 512, nullptr, infoLog);  
 std::cerr << "ERROR: Shader compilation failed\n" << infoLog << std::endl;  
 }  
 return shader;  
}  
  
GLuint createShaderProgram() {  
 GLuint vertexShader = compileShader(GL\_VERTEX\_SHADER, vertexShaderSource);  
 GLuint fragmentShader = compileShader(GL\_FRAGMENT\_SHADER, fragmentShaderSource);  
  
 GLuint shaderProgram = glCreateProgram();  
 glAttachShader(shaderProgram, vertexShader);  
 glAttachShader(shaderProgram, fragmentShader);  
 glLinkProgram(shaderProgram);  
  
 GLint success;  
 glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);  
 if (!success) {  
 char infoLog[512];  
 glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);  
 std::cerr << "ERROR: Shader program linking failed\n" << infoLog << std::endl;  
 }  
  
 glDeleteShader(vertexShader);  
 glDeleteShader(fragmentShader);  
  
 return shaderProgram;  
}  
  
void processInput(GLFWwindow \*window, Camera &camera) {  
 static bool fPressed = false;  
  
 if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)  
 glfwSetWindowShouldClose(window, true);  
  
 float speed = camera.speed;  
 if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)  
 camera.position += speed \* camera.front;  
 if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)  
 camera.position -= speed \* camera.front;  
  
 glm::vec3 right = glm::normalize(glm::cross(camera.front, camera.up));  
 if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)  
 camera.position -= right \* speed;  
 if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)  
 camera.position += right \* speed;  
  
 if (glfwGetKey(window, GLFW\_KEY\_SPACE) == GLFW\_PRESS)  
 camera.position += camera.up \* speed;  
 if (glfwGetKey(window, GLFW\_KEY\_LEFT\_SHIFT) == GLFW\_PRESS)  
 camera.position -= camera.up \* speed;  
 if (glfwGetKey(window, GLFW\_KEY\_F) == GLFW\_PRESS) {  
 if (!fPressed) {  
 usePerspective = !usePerspective;  
 fPressed = true;  
 }  
 } else {  
 fPressed = false;  
 }  
}  
  
glm::mat4 getViewMatrix(const Camera &camera) {  
 return glm::lookAt(camera.position, camera.position + camera.front, camera.up);  
}  
  
int main() {  
 if (!glfwInit()) {  
 std::cerr << "Failed to initialize GLFW" << std::endl;  
 return -1;  
 }  
  
 glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);  
 glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);  
 glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);  
  
 GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "OpenGL Sphere & Bezier Plane with Diamond Hole", nullptr, nullptr);  
 if (!window) {  
 std::cerr << "Failed to create GLFW window" << std::endl;  
 glfwTerminate();  
 return -1;  
 }  
  
 glfwMakeContextCurrent(window);  
 glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);  
 glfwSetCursorPosCallback(window, mouse\_callback);  
  
 if (glewInit() != GLEW\_OK) {  
 std::cerr << "Failed to initialize GLEW" << std::endl;  
 return -1;  
 }  
  
 glEnable(GL\_DEPTH\_TEST);  
  
 GLuint shaderProgram = createShaderProgram();  
  
 // Create sphere  
 std::vector<float> sphereVertices;  
 std::vector<unsigned int> sphereIndices;  
 generateSphereVertices(sphereVertices, 1.0f, 36, 18);  
 generateSphereIndices(sphereIndices, 36, 18);  
  
 GLuint sphereVAO, sphereVBO, sphereEBO;  
 glGenVertexArrays(1, &sphereVAO);  
 glGenBuffers(1, &sphereVBO);  
 glGenBuffers(1, &sphereEBO);  
  
 glBindVertexArray(sphereVAO);  
 glBindBuffer(GL\_ARRAY\_BUFFER, sphereVBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sphereVertices.size() \* sizeof(float), sphereVertices.data(), GL\_STATIC\_DRAW);  
  
 glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, sphereEBO);  
 glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sphereIndices.size() \* sizeof(unsigned int), sphereIndices.data(), GL\_STATIC\_DRAW);  
  
 // Position attribute for sphere  
 glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);  
 glEnableVertexAttribArray(0);  
  
 // Normal attribute for sphere  
 glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)(3 \* sizeof(float)));  
 glEnableVertexAttribArray(1);  
  
 glBindVertexArray(0);  
  
 // Create Bezier plane with diamond hole  
 std::vector<float> planeVertices;  
 std::vector<unsigned int> planeIndices;  
 generateBezierPlane(planeVertices, planeIndices, 40, 2.0f); // Higher resolution for smoother curves  
  
 GLuint planeVAO, planeVBO, planeEBO;  
 glGenVertexArrays(1, &planeVAO);  
 glGenBuffers(1, &planeVBO);  
 glGenBuffers(1, &planeEBO);  
  
 glBindVertexArray(planeVAO);  
 glBindBuffer(GL\_ARRAY\_BUFFER, planeVBO);  
 glBufferData(GL\_ARRAY\_BUFFER, planeVertices.size() \* sizeof(float), planeVertices.data(), GL\_STATIC\_DRAW);  
  
 glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, planeEBO);  
 glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, planeIndices.size() \* sizeof(unsigned int), planeIndices.data(), GL\_STATIC\_DRAW);  
  
 // Position attribute for plane  
 glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);  
 glEnableVertexAttribArray(0);  
  
 // Normal attribute for plane  
 glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)(3 \* sizeof(float)));  
 glEnableVertexAttribArray(1);  
  
 glBindVertexArray(0);  
  
 // Light properties and object colors  
 glm::vec3 lightPos(1.2f, 1.0f, 2.0f);  
 glm::vec3 lightColor(1.0f, 1.0f, 1.0f);  
 glm::vec3 sphereColor(0.5f, 0.2f, 0.7f); // Purple-ish color for sphere  
 glm::vec3 planeColor(0.2f, 0.6f, 0.5f); // Teal-ish color for plane  
  
 while (!glfwWindowShouldClose(window)) {  
 processInput(window, camera);  
  
 glClearColor(0.1f, 0.1f, 0.1f, 1.0f); // Dark background  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
  
 glUseProgram(shaderProgram);  
  
 // Set lighting uniforms  
 glUniform3fv(glGetUniformLocation(shaderProgram, "lightPos"), 1, glm::value\_ptr(lightPos));  
 glUniform3fv(glGetUniformLocation(shaderProgram, "viewPos"), 1, glm::value\_ptr(camera.position));  
 glUniform3fv(glGetUniformLocation(shaderProgram, "lightColor"), 1, glm::value\_ptr(lightColor));  
  
 glm::mat4 view = getViewMatrix(camera);  
 glm::mat4 projection;  
 if (usePerspective) {  
 projection = glm::perspective(glm::radians(45.0f), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);  
 }  
 else {  
 float aspectRatio = (float)SCR\_WIDTH / (float)SCR\_HEIGHT;  
 float halfSize = orthogonalSize / 2.0f;  
 projection = glm::ortho(-halfSize \* aspectRatio, halfSize \* aspectRatio, -halfSize, halfSize, 0.1f, 100.0f);  
 }  
  
 GLuint viewLoc = glGetUniformLocation(shaderProgram, "view");  
 GLuint projectionLoc = glGetUniformLocation(shaderProgram, "projection");  
 GLuint modelLoc = glGetUniformLocation(shaderProgram, "model");  
  
 glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));  
 glUniformMatrix4fv(projectionLoc, 1, GL\_FALSE, glm::value\_ptr(projection));  
  
 // Move the light position over time for dynamic lighting effect  
 float timeValue = glfwGetTime();  
 // lightPos.x = 2.4f + sin(timeValue) \* 2.0f;  
 // lightPos.y = 2.0f + sin(timeValue / 2.0f) \* 1.0f;  
  
 lightPos.x = 4.0f;  
 lightPos.y = 3.0f;  
 lightPos.z = -0.5f;  
  
  
 // Draw sphere  
 glm::mat4 sphereModel = glm::mat4(1.0f);  
 sphereModel = glm::translate(sphereModel, glm::vec3(0.0f, 0.0f, 0.0f));  
 glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(sphereModel));  
 glUniform3fv(glGetUniformLocation(shaderProgram, "objectColor"), 1, glm::value\_ptr(sphereColor));  
  
 glBindVertexArray(sphereVAO);  
 glDrawElements(GL\_TRIANGLES, sphereIndices.size(), GL\_UNSIGNED\_INT, 0);  
  
 // Draw Bezier plane with hole  
 glm::mat4 planeModel = glm::mat4(1.0f);  
 planeModel = glm::translate(planeModel, glm::vec3(4.0f, -1.0f, 0.0f));  
 glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(planeModel));  
 glUniform3fv(glGetUniformLocation(shaderProgram, "objectColor"), 1, glm::value\_ptr(planeColor));  
  
 glBindVertexArray(planeVAO);  
 glDrawElements(GL\_TRIANGLES, planeIndices.size(), GL\_UNSIGNED\_INT, 0);  
  
 glfwSwapBuffers(window);  
 glfwPollEvents();  
 }  
  
 // Clean up  
 glDeleteVertexArrays(1, &sphereVAO);  
 glDeleteBuffers(1, &sphereVBO);  
 glDeleteBuffers(1, &sphereEBO);  
  
 glDeleteVertexArrays(1, &planeVAO);  
 glDeleteBuffers(1, &planeVBO);  
 glDeleteBuffers(1, &planeEBO);  
  
 glDeleteProgram(shaderProgram);  
  
 glfwTerminate();  
 return 0;  
}

**Результати виконання роботи**

****

*Рис. 1. Сфера та поверхня безʼє в перспективній проєʼкції*

**

*Рис. 2. Сфера та поверхня безʼє в ортогональній проєʼкції*

**Висновки:** протягом виконання цієї лабораторної роботи я навчився будувати тривимірні об’єкти та проектувати їх на екран монітора, застосовуючи функції OpenGL.