**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

**ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи №2

# на тему: “Генерування динамічних двовимірних зображень за допомогою бібліотеки 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: Генерування динамічних двовимірних зображень за допомогою бібліотеки OpenGL.

**Мета:** Навчитися будувати двовимірні зображення з подальшим їхнім переміщенням, обертанням та масштабуванням на основі використання базових геометричних перетворень OpenGL.

**Теоретичні відомості**

Для створення динамічних зображень, які змінюють у часі своє положення, орієнтацію та розміри, в OpenGL використовують основні геометричні перетворення – перенесення, обертання і масштабування.

Більшість фігур наперед визначеної форми (особливо це стосується тривимірних об’єктів) можна намалювати тільки на початку координат, тобто в точці (0,0,0). Для того, щоб зобразити фігуру в точці (x0,y0,z0), треба перемістити початок координат у цю точку, тобто перейти до нових координат. Часто буває дуже зручно, крім того, повернути координати на необхідний кут. Для переходу до нових координат в OpenGL є дві функції:

**glTranslate(dx,dy,dz)**

**glRotate(ang,x0,y0,z0)**

Перша функція зміщує початок системи координат на (*dx,dy,dz*), друга – повертає на кут *ang* проти годинникової стрілки навколо вектора (*x0,y0,z0*), який виходить з початку координат (0,0,0). Ще дві дуже важливі функції –

**glPushMatrix()**

**glPopMatrix()**

призначені для збереження й відновлення поточних координат. Отож, для того, щоб намалювати фігуру не на початку координат, треба:

1. зберегти поточні координати (**glPushMatrix**);
2. змістити (**glTranslated**) та повернути (**glRotated**) систему координат;
3. намалювати те, що треба;
4. повернутися до старих координат (**glPopMatrix**).

Виклики **glPushMatrixglPopMatrix**можуть бути вкладеними, тобто:

**glPushMatrix();**

**...**

**glPushMatrix();**

**glPopMatrix();**

**...**

**glPopMatrix();**

Зрозуміло, що кількість викликів **glPopMatrix**повинне відповідати кількості викликів **glPushMatrix**, інакше вся сцена полетить у наперед невідомому напрямку.

Подібним чином функція **glScale(kx,ky,kz)** масштабує об’єкт (стискає чи розтягує його) на задані коефіцієнти по відповідних осях. Вона підходить також і для генерації дзеркального відображення – для цього коефіцієнтам надають від’ємних значень.

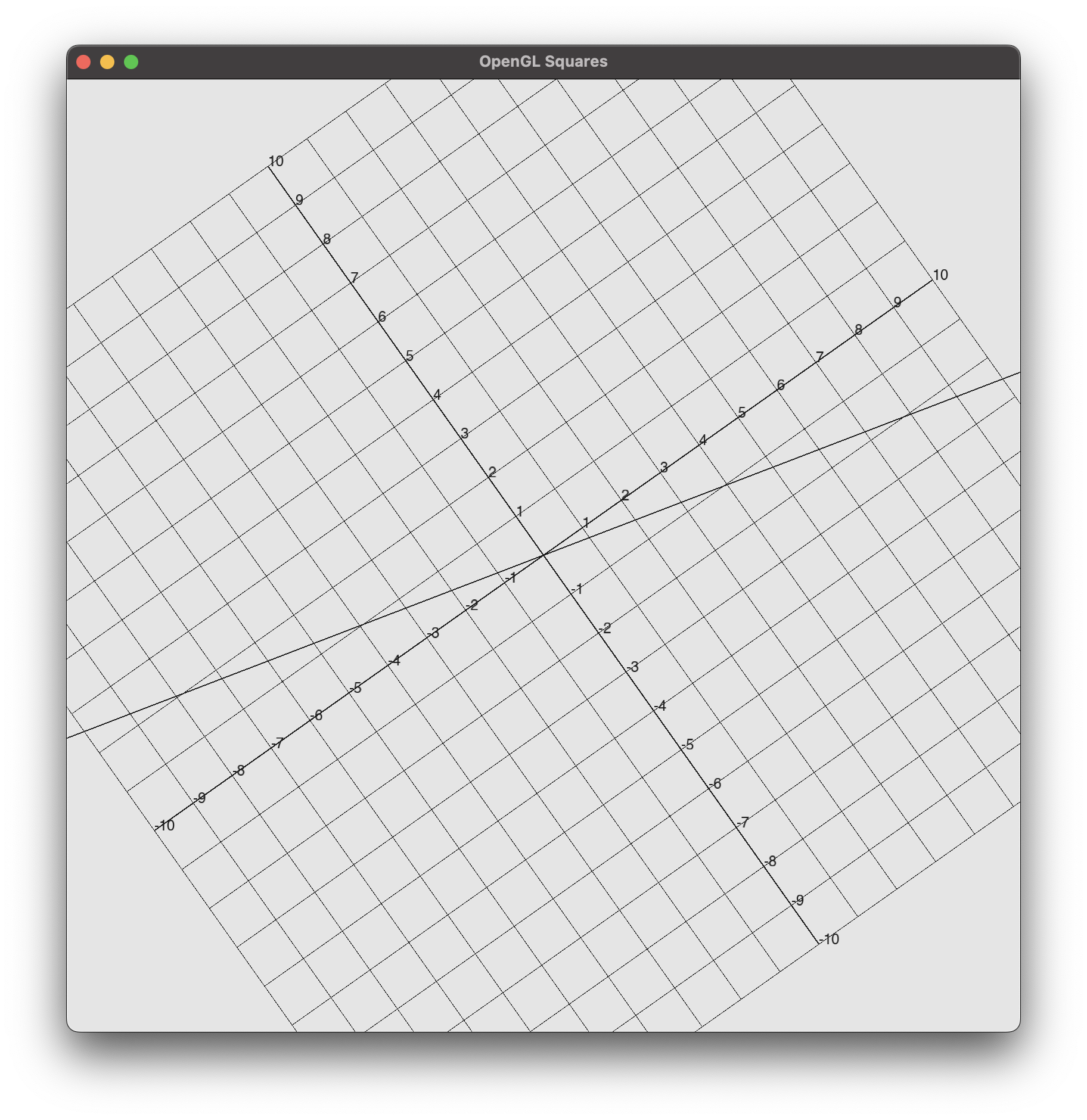
Отож, для створення засобами OpenGL сцени із динамічно змінюваними об’єктами треба організувати цикл, на кожній ітерації якого будуть обчислені значення зміщень та кутів повороту потрібних об’єктів (і, якщо необхідно, коефіцієнтів масштабування) та виконано пункти *a)–d)* процедури малювання об’єктів.

**Завдання**

10. Візуалізувати **графік функції** ***y=ax*** з осями координат, на які нанесено шкалу з відповідними підписами. Реалізувати одночасне обертання відносно точки **(0;0)**: лінії графіку – за годинниковою стрілкою, а осей – проти годинникової стрілки.

**Код програми**

#include <iostream>  
#include <GL/glew.h>  
#include <GLFW/glfw3.h>  
#include <vector>  
#include <cstdlib>  
#include <ctime>  
#include <sstream>  
#include <map>  
#include <ft2build.h>  
#include FT\_FREETYPE\_H  
#include <glm/glm.hpp>  
#include <glm/gtc/matrix\_transform.hpp>  
#include <glm/gtc/type\_ptr.hpp>  
  
// Shader sources  
const char\* vertexShaderSource = R"(  
 #version 410 core  
 layout (location = 0) in vec2 aPos;  
 void main() {  
 gl\_Position = vec4(aPos, 0.0, 1.0);  
 }  
)";  
  
const char\* fragmentShaderSource = R"(  
 #version 410 core  
 out vec4 FragColor;  
 uniform vec4 squareColor;  
 void main() {  
 FragColor = squareColor;  
 }  
)";  
  
// Text rendering shader sources  
const char\* textVertexShaderSource = R"(  
 #version 410 core  
 layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>  
 out vec2 TexCoords;  
  
 uniform mat4 projection;  
  
 void main() {  
 gl\_Position = projection \* vec4(vertex.xy, 0.0, 1.0);  
 TexCoords = vertex.zw;  
 }  
)";  
  
const char\* textFragmentShaderSource = R"(  
 #version 410 core  
 in vec2 TexCoords;  
 out vec4 color;  
  
 uniform sampler2D text;  
 uniform vec3 textColor;  
  
 void main() {  
 vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);  
 color = vec4(textColor, 1.0) \* sampled;  
 }  
)";  
  
// Structure to store character data  
struct Character {  
 unsigned int TextureID; // ID handle of the glyph texture  
 glm::ivec2 Size; // Size of glyph  
 glm::ivec2 Bearing; // Offset from baseline to left/top of glyph  
 unsigned int Advance; // Offset to advance to next glyph  
};  
  
// Global variables  
GLuint shaderProgram, VAO, VBO;  
GLuint textShaderProgram, textVAO, textVBO;  
float scale = 1.0f;  
int windowWidth, windowHeight;  
float a = 1.0f;  
float coordinateRotationAngle = 0.0f;  
float approxStep = 0.1f;  
bool playAnimation = false;  
std::map<char, Character> Characters; // Map of characters for text rendering  
glm::mat4 projection; // Projection matrix for text rendering  
  
// Forward declaration  
void renderText(const std::string& text, float x, float y, float scale, glm::vec3 color);  
  
// Initialize GLFW, GLEW, and OpenGL settings  
bool initOpenGL(GLFWwindow\*& window) {  
 if (!glfwInit()) {  
 std::cerr << "Failed to initialize GLFW\n";  
 return false;  
 }  
  
 glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);  
 glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 1);  
 glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);  
 glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);  
  
 window = glfwCreateWindow(800, 800, "OpenGL Squares", nullptr, nullptr);  
 if (!window) {  
 std::cerr << "Failed to create GLFW window\n";  
 glfwTerminate();  
 return false;  
 }  
  
 glfwMakeContextCurrent(window);  
 glfwSetFramebufferSizeCallback(window, [](GLFWwindow\* window, int width, int height) {  
 glViewport(0, 0, width, height);  
 });  
  
 glewExperimental = GL\_TRUE;  
 if (glewInit() != GLEW\_OK) {  
 std::cerr << "Failed to initialize GLEW\n";  
 return false;  
 }  
  
 glClearColor(0.9f, 0.9f, 0.9f, 1.0f); // Dark background  
 return true;  
}  
  
// Initialize shaders and buffers  
void initShadersAndBuffers() {  
 // Compile vertex shader  
 GLuint vertexShader = glCreateShader(GL\_VERTEX\_SHADER);  
 glShaderSource(vertexShader, 1, &vertexShaderSource, nullptr);  
 glCompileShader(vertexShader);  
  
 // Compile fragment shader  
 GLuint fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);  
 glShaderSource(fragmentShader, 1, &fragmentShaderSource, nullptr);  
 glCompileShader(fragmentShader);  
  
 // Link shaders to a program  
 shaderProgram = glCreateProgram();  
 glAttachShader(shaderProgram, vertexShader);  
 glAttachShader(shaderProgram, fragmentShader);  
 glLinkProgram(shaderProgram);  
 glUseProgram(shaderProgram);  
  
 glDeleteShader(vertexShader);  
 glDeleteShader(fragmentShader);  
  
 // Set up vertex data for a square (4 vertices)  
 GLfloat vertices[] = {  
 -0.5f, -0.5f, // Bottom-left  
 0.5f, -0.5f, // Bottom-right  
 -0.5f, 0.5f, // Top-left  
 0.5f, 0.5f // Top-right  
 };  
  
 // Set up buffers  
 glGenVertexArrays(1, &VAO);  
 glGenBuffers(1, &VBO);  
  
 glBindVertexArray(VAO);  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  
  
 glVertexAttribPointer(0, 2, GL\_FLOAT, GL\_FALSE, 2 \* sizeof(GLfloat), (GLvoid\*)0);  
 glEnableVertexAttribArray(0);  
 glBindVertexArray(0); // Unbind VAO  
  
 // Initialize text shaders  
 vertexShader = glCreateShader(GL\_VERTEX\_SHADER);  
 glShaderSource(vertexShader, 1, &textVertexShaderSource, nullptr);  
 glCompileShader(vertexShader);  
  
 fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);  
 glShaderSource(fragmentShader, 1, &textFragmentShaderSource, nullptr);  
 glCompileShader(fragmentShader);  
  
 textShaderProgram = glCreateProgram();  
 glAttachShader(textShaderProgram, vertexShader);  
 glAttachShader(textShaderProgram, fragmentShader);  
 glLinkProgram(textShaderProgram);  
  
 glDeleteShader(vertexShader);  
 glDeleteShader(fragmentShader);  
  
 // Initialize text VAO and VBO  
 glGenVertexArrays(1, &textVAO);  
 glGenBuffers(1, &textVBO);  
 glBindVertexArray(textVAO);  
 glBindBuffer(GL\_ARRAY\_BUFFER, textVBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 6 \* 4, nullptr, GL\_DYNAMIC\_DRAW);  
 glVertexAttribPointer(0, 4, GL\_FLOAT, GL\_FALSE, 4 \* sizeof(float), 0);  
 glEnableVertexAttribArray(0);  
 glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
 glBindVertexArray(0);  
  
 // Set up orthographic projection for text rendering  
 int width, height;  
 glfwGetFramebufferSize(glfwGetCurrentContext(), &width, &height);  
 projection = glm::ortho(0.0f, static\_cast<float>(width), 0.0f, static\_cast<float>(height));  
}  
  
// Initialize FreeType and load a font  
bool initFont() {  
 FT\_Library ft;  
 if (FT\_Init\_FreeType(&ft)) {  
 std::cerr << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;  
 return false;  
 }  
  
 FT\_Face face;  
 // Load font (use a path to a TTF font file on your system)  
 if (FT\_New\_Face(ft, "/System/Library/Fonts/Helvetica.ttc", 0, &face)) {  
 std::cerr << "ERROR::FREETYPE: Failed to load font" << std::endl;  
 return false;  
 }  
  
 FT\_Set\_Pixel\_Sizes(face, 0, 24); // Set size to load glyphs as  
  
 glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1); // Disable byte-alignment restriction  
  
 // Load first 128 ASCII characters  
 for (unsigned char c = 0; c < 128; c++) {  
 // Load character glyph  
 if (FT\_Load\_Char(face, c, FT\_LOAD\_RENDER)) {  
 std::cerr << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;  
 continue;  
 }  
  
 // Generate texture  
 unsigned int texture;  
 glGenTextures(1, &texture);  
 glBindTexture(GL\_TEXTURE\_2D, texture);  
 glTexImage2D(  
 GL\_TEXTURE\_2D,  
 0,  
 GL\_RED,  
 face->glyph->bitmap.width,  
 face->glyph->bitmap.rows,  
 0,  
 GL\_RED,  
 GL\_UNSIGNED\_BYTE,  
 face->glyph->bitmap.buffer  
 );  
  
 // Set texture options  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  
  
 // Now store character for later use  
 Character character = {  
 texture,  
 glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),  
 glm::ivec2(face->glyph->bitmap\_left, face->glyph->bitmap\_top),  
 static\_cast<unsigned int>(face->glyph->advance.x)  
 };  
 Characters.insert(std::pair<char, Character>(c, character));  
 }  
  
 // Destroy FreeType once we're finished  
 FT\_Done\_Face(face);  
 FT\_Done\_FreeType(ft);  
 return true;  
}  
  
// Render a text string  
void renderText(const std::string& text, float x, float y, float scale, glm::vec3 color) {  
 // Activate corresponding render state  
 glUseProgram(textShaderProgram);  
 glUniform3f(glGetUniformLocation(textShaderProgram, "textColor"), color.x, color.y, color.z);  
 glUniformMatrix4fv(glGetUniformLocation(textShaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));  
 glActiveTexture(GL\_TEXTURE0);  
 glBindVertexArray(textVAO);  
  
 // Iterate through all characters  
 float startX = x;  
 for (char c : text) {  
 Character ch = Characters[c];  
  
 float xpos = startX + ch.Bearing.x \* scale;  
 float ypos = y - (ch.Size.y - ch.Bearing.y) \* scale;  
  
 float w = ch.Size.x \* scale;  
 float h = ch.Size.y \* scale;  
  
 // Update VBO for each character  
 float vertices[6][4] = {  
 { xpos, ypos + h, 0.0f, 0.0f },  
 { xpos, ypos, 0.0f, 1.0f },  
 { xpos + w, ypos, 1.0f, 1.0f },  
  
 { xpos, ypos + h, 0.0f, 0.0f },  
 { xpos + w, ypos, 1.0f, 1.0f },  
 { xpos + w, ypos + h, 1.0f, 0.0f }  
 };  
  
 // Render glyph texture over quad  
 glBindTexture(GL\_TEXTURE\_2D, ch.TextureID);  
  
 // Update content of VBO memory  
 glBindBuffer(GL\_ARRAY\_BUFFER, textVBO);  
 glBufferSubData(GL\_ARRAY\_BUFFER, 0, sizeof(vertices), vertices);  
 glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
  
 // Render quad  
 glDrawArrays(GL\_TRIANGLES, 0, 6);  
  
 // Now advance cursors for next glyph (note that advance is number of 1/64 pixels)  
 startX += (ch.Advance >> 6) \* scale; // Bitshift by 6 to get value in pixels (2^6 = 64)  
 }  
 glBindVertexArray(0);  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
}  
  
// Function to invert color  
void invertColor(const GLfloat color[3], GLfloat invertedColor[3]) {  
 for (int i = 0; i < 3; i++) {  
 invertedColor[i] = 1.0f - color[i]; // Invert the color component  
 }  
}  
  
// Convert normalized OpenGL coordinates to window coordinates  
void normalizedToWindowCoords(float& x, float& y) {  
 // Convert from [-1, 1] to [0, width/height]  
 x = (x + 1.0f) \* 0.5f \* windowWidth;  
 // Convert from [-1, 1] to [0, height] for y (flip y-axis)  
 y = (1.0f - (y + 1.0f) \* 0.5f) \* windowHeight;  
}  
  
// Convert window coordinates to normalized OpenGL coordinates  
void windowToNormalizedCoords(float& x, float& y) {  
 // Convert from [0, width] to [-1, 1] for x  
 x = (x / windowWidth) \* 2.0f - 1.0f;  
  
 // Convert from [0, height] to [-1, 1] for y (flip y-axis)  
 y = 1.0f - (y / windowHeight) \* 2.0f;  
}  
  
float pixelToNDC(int pixelCount, bool isRelativeToWidth) {  
 if (isRelativeToWidth) {  
 // Convert pixel count to NDC relative to width  
 return static\_cast<float>(pixelCount) / windowWidth \* 2.0f;  
 } else {  
 // Convert pixel count to NDC relative to height  
 return static\_cast<float>(pixelCount) / windowHeight \* 2.0f;  
 }  
}  
  
float NDCToPixel(float ndcValue, bool isRelativeToWidth) {  
 if (isRelativeToWidth) {  
 // Convert pixel count to NDC relative to width  
 return ndcValue \* windowWidth / 2.0f;  
 } else {  
 // Convert pixel count to NDC relative to height  
 return ndcValue \* windowHeight / 2.0f;  
 }  
}  
  
void drawVerticalLine(float offset, float lineWidth, std::vector<GLfloat> color) {  
 if (color.size() != 4) {  
 throw std::runtime\_error("draw line has to have 4 elements");  
 }  
  
 glUseProgram(shaderProgram);  
 glBindVertexArray(VAO);  
 // Set the line color  
 glUniform4f(glGetUniformLocation(shaderProgram, "squareColor"), color[0], color[1], color[2], color[3]);  
  
 // Apply rotation transformation  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
 GLfloat model[] = {  
 offset + lineWidth / 2, 1.0f,  
 offset - lineWidth / 2, 1.0f,  
 offset + lineWidth / 2, -1.0f,  
 offset - lineWidth / 2, -1.0f,  
 };  
  
 // Rotate each vertex  
 for (int i = 0; i < 4; i++) {  
 float x = model[i \* 2];  
 float y = model[i \* 2 + 1];  
 model[i \* 2] = x \* cosTheta - y \* sinTheta; // Rotated x  
 model[i \* 2 + 1] = x \* sinTheta + y \* cosTheta; // Rotated y  
 }  
  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(model), model, GL\_DYNAMIC\_DRAW);  
 glDrawArrays(GL\_TRIANGLE\_STRIP, 0, 4);  
  
 glBindVertexArray(0);  
}  
  
void drawHorizontalLine(float offset, float lineWidth, std::vector<GLfloat> color) {  
 if (color.size() != 4) {  
 throw std::runtime\_error("draw line has to have 4 elements");  
 }  
  
 glUseProgram(shaderProgram);  
 glBindVertexArray(VAO);  
 // Set the line color  
 glUniform4f(glGetUniformLocation(shaderProgram, "squareColor"), color[0], color[1], color[2], color[3]);  
  
 // Apply rotation transformation  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
 GLfloat model[] = {  
 1.0f, offset + lineWidth / 2,  
 1.0f, offset - lineWidth / 2,  
 -1.0f, offset + lineWidth / 2,  
 -1.0f, offset - lineWidth / 2,  
 };  
  
 // Rotate each vertex  
 for (int i = 0; i < 4; i++) {  
 float x = model[i \* 2];  
 float y = model[i \* 2 + 1];  
 model[i \* 2] = x \* cosTheta - y \* sinTheta; // Rotated x  
 model[i \* 2 + 1] = x \* sinTheta + y \* cosTheta; // Rotated y  
 }  
  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(model), model, GL\_DYNAMIC\_DRAW);  
 glDrawArrays(GL\_TRIANGLE\_STRIP, 0, 4);  
  
 glBindVertexArray(0);  
}  
  
void drawCoordinates() {  
 std::vector<GLfloat> lineColor = {0.1f, 0.1f, 0.1f, 1.0f};  
 float boldLineVert = pixelToNDC(2, true);  
 float normalLineVert = pixelToNDC(1, true);  
  
 float boldLineHoriz = pixelToNDC(2, false);  
 float normalLineHoriz = pixelToNDC(1, false);  
  
 int maxDimension = std::max(windowWidth, windowHeight);  
 bool isRelativeToWidth = maxDimension == windowWidth;  
  
 float pInitialPosition = static\_cast<float>(windowWidth) / 2.0f;  
 const float pOffset = NDCToPixel(scale / 10.0f, isRelativeToWidth);  
 float pPosition = pInitialPosition;  
  
 drawVerticalLine(0, boldLineVert, lineColor); // Central vertical coord line  
  
 int coordsLabel = 0;  
 // Draw vertical lines  
 while (pPosition + pOffset <= static\_cast<float>(windowWidth)) {  
 pPosition += pOffset;  
 float position = pixelToNDC(pPosition, isRelativeToWidth) - 1.0f;  
 drawVerticalLine(position, normalLineVert, lineColor);  
  
 // Calculate rotated text position  
 float textX = pPosition;  
 float textY = windowHeight / 2;  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
 float centerX = windowWidth / 2.0f;  
 float centerY = windowHeight / 2.0f;  
  
 // Translate to origin  
 float translatedX = textX - centerX;  
 float translatedY = textY - centerY;  
  
 // Rotate  
 float rotatedX = translatedX \* cosTheta - translatedY \* sinTheta;  
 float rotatedY = translatedX \* sinTheta + translatedY \* cosTheta;  
  
 // Translate back  
 float finalX = rotatedX + centerX;  
 float finalY = rotatedY + centerY;  
  
  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
 renderText(std::to\_string(++coordsLabel), finalX, finalY, 1.0f, glm::vec3(0.1f, 0.1f, 0.1f));  
 glDisable(GL\_BLEND);  
 }  
 pPosition = pInitialPosition;  
 coordsLabel = 0;  
 while (pPosition - pOffset >= 0) {  
 pPosition -= pOffset;  
 float position = pixelToNDC(pPosition, isRelativeToWidth) - 1.0f;  
 drawVerticalLine(position, normalLineVert, lineColor);  
  
 // Calculate rotated text position  
 float textX = pPosition;  
 float textY = windowHeight / 2;  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
 float centerX = windowWidth / 2.0f;  
 float centerY = windowHeight / 2.0f;  
  
 // Translate to origin  
 float translatedX = textX - centerX;  
 float translatedY = textY - centerY;  
  
 // Rotate  
 float rotatedX = translatedX \* cosTheta - translatedY \* sinTheta;  
 float rotatedY = translatedX \* sinTheta + translatedY \* cosTheta;  
  
 // Translate back  
 float finalX = rotatedX + centerX;  
 float finalY = rotatedY + centerY;  
  
  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
 renderText(std::to\_string(--coordsLabel), finalX, finalY, 1.0f, glm::vec3(0.1f, 0.1f, 0.1f));  
 glDisable(GL\_BLEND);  
 }  
  
 pInitialPosition = static\_cast<float>(windowHeight) / 2.0f;  
 pPosition = pInitialPosition;  
 coordsLabel = 0;  
  
 drawHorizontalLine(0, boldLineHoriz, lineColor); // Central horizontal coord line  
  
 // Draw horizontal lines  
 while (pPosition + pOffset <= windowHeight) {  
 pPosition += pOffset;  
 float position = pixelToNDC(pPosition, !isRelativeToWidth) - 1.0f;  
 drawHorizontalLine(position, normalLineHoriz, lineColor);  
  
 // Calculate rotated text position  
 float textX = windowWidth / 2;  
 float textY = pPosition;  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
  
 float centerX = windowWidth / 2.0f;  
 float centerY = windowHeight / 2.0f;  
  
 // Translate to origin  
 float translatedX = textX - centerX;  
 float translatedY = textY - centerY;  
  
 // Rotate  
 float rotatedX = translatedX \* cosTheta - translatedY \* sinTheta;  
 float rotatedY = translatedX \* sinTheta + translatedY \* cosTheta;  
  
 // Translate back  
 float finalX = rotatedX + centerX;  
 float finalY = rotatedY + centerY;  
  
  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
 renderText(std::to\_string(++coordsLabel), finalX, finalY, 1.0f, glm::vec3(0.1f, 0.1f, 0.1f));  
 glDisable(GL\_BLEND);  
 }  
 pPosition = pInitialPosition;  
 coordsLabel = 0;  
 while (pPosition - pOffset >= 0) {  
 pPosition -= pOffset;  
 float position = pixelToNDC(pPosition, !isRelativeToWidth) - 1.0f;  
 drawHorizontalLine(position, normalLineHoriz, lineColor);  
  
 // Calculate rotated text position  
 float textX = windowWidth / 2;  
 float textY = pPosition;  
 float cosTheta = cos(coordinateRotationAngle);  
 float sinTheta = sin(coordinateRotationAngle);  
  
 float centerX = windowWidth / 2.0f;  
 float centerY = windowHeight / 2.0f;  
  
 // Translate to origin  
 float translatedX = textX - centerX;  
 float translatedY = textY - centerY;  
  
 // Rotate  
 float rotatedX = translatedX \* cosTheta - translatedY \* sinTheta;  
 float rotatedY = translatedX \* sinTheta + translatedY \* cosTheta;  
  
 // Translate back  
 float finalX = rotatedX + centerX;  
 float finalY = rotatedY + centerY;  
  
  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
 renderText(std::to\_string(--coordsLabel), finalX, finalY, 1.0f, glm::vec3(0.1f, 0.1f, 0.1f));  
 glDisable(GL\_BLEND);  
 }  
}  
  
void drawFunction(float lineWidth, std::vector<GLfloat> color) {  
 if (color.size() != 4) {  
 throw std::runtime\_error("draw line has to have 4 elements");  
 }  
  
 // Set the line width  
 glLineWidth(lineWidth);  
  
 // Enable line antialiasing (optional)  
 glEnable(GL\_LINE\_SMOOTH);  
 glHint(GL\_LINE\_SMOOTH\_HINT, GL\_NICEST);  
  
 glUseProgram(shaderProgram);  
 glBindVertexArray(VAO);  
 // Set the line color  
 glUniform4f(glGetUniformLocation(shaderProgram, "functionLineColor"), color[0], color[1], color[2], color[3]);  
  
 std::vector<GLfloat> model;  
  
 float x = 0.0f;  
 float y = 0.0f;  
 for (float i = -1.0f; i < 1.000009f; i += approxStep) {  
 x = i;  
 y = a \* x;  
  
 model.push\_back(x);  
 model.push\_back(y);  
 }  
  
 GLfloat modelArray[model.size()];  
 std::copy(model.begin(), model.end(), modelArray);  
  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(modelArray), modelArray, GL\_DYNAMIC\_DRAW);  
  
 // Use GL\_LINE\_STRIP to draw a continuous line  
 glDrawArrays(GL\_LINE\_STRIP, 0, model.size() / 2);  
  
 glBindVertexArray(0);  
  
 // Disable line antialiasing (optional)  
 glDisable(GL\_LINE\_SMOOTH);  
}  
  
void handleKeyboardInput(GLFWwindow\* window) {  
 static bool zPressed = false;  
 static bool xPressed = false;  
 static bool qPressed = false;  
 static bool wPressed = false;  
 static bool spacePressed = false;  
  
 if (glfwGetKey(window, GLFW\_KEY\_Z) == GLFW\_PRESS) {  
 if (!zPressed) {  
 scale += 0.1f;  
 zPressed = true;  
 }  
 } else {  
 zPressed = false;  
 }  
  
  
 if (glfwGetKey(window, GLFW\_KEY\_X) == GLFW\_PRESS) {  
 if (!xPressed) {  
 scale -= 0.1f;  
 xPressed = true;  
 }  
 } else {  
 xPressed = false;  
 }  
  
 if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS) {  
 if (!qPressed) {  
 a += 0.1f;  
 coordinateRotationAngle -= 0.1f;  
 qPressed = true;  
 }  
 } else {  
 qPressed = false;  
 }  
  
 if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {  
 if (!wPressed) {  
 a -= 0.1f;  
 coordinateRotationAngle += 0.1f;  
 wPressed = true;  
 }  
 } else {  
 wPressed = false;  
 }  
  
 if (glfwGetKey(window, GLFW\_KEY\_SPACE) == GLFW\_PRESS) {  
 if (!spacePressed) {  
 playAnimation = !playAnimation;  
 spacePressed = true;  
 }  
 } else {  
 spacePressed = false;  
 }  
}  
  
// Main loop for handling events and rendering  
void mainLoop(GLFWwindow\* window) {  
 while (!glfwWindowShouldClose(window)) {  
 glClear(GL\_COLOR\_BUFFER\_BIT);  
  
 glfwGetFramebufferSize(glfwGetCurrentContext(), &windowWidth, &windowHeight);  
  
 handleKeyboardInput(window);  
  
 drawCoordinates();  
 drawFunction(50.0f, {1.0f, 0.0f, 0.0f, 1.0f});  
  
 if (playAnimation) {  
 a -= 0.001f;  
 coordinateRotationAngle += 0.001f;  
 }  
  
 glfwSwapBuffers(window);  
 glfwPollEvents();  
 }  
}  
  
int main() {  
 srand(time(0)); // Seed for random number generation  
  
 GLFWwindow\* window;  
  
 // Initialize OpenGL  
 if (!initOpenGL(window)) {  
 return -1;  
 }  
  
 // Initialize shaders and buffers  
 initShadersAndBuffers();  
  
 // Initialize font for text rendering  
 if (!initFont()) {  
 return -1;  
 }  
  
 // Start the main loop  
 mainLoop(window);  
  
 // Clean up and terminate  
 glDeleteVertexArrays(1, &VAO);  
 glDeleteBuffers(1, &VBO);  
 glDeleteVertexArrays(1, &textVAO);  
 glDeleteBuffers(1, &textVBO);  
 glDeleteProgram(shaderProgram);  
 glDeleteProgram(textShaderProgram);  
  
 // Clean up character textures  
 for (auto& c : Characters) {  
 glDeleteTextures(1, &c.second.TextureID);  
 }  
  
 glfwDestroyWindow(window);  
 glfwTerminate();  
 return 0;  
}

**Результати виконання роботи**

*Рис. 1. Обертання графіка функії y = ax за годинниковою стрілкоб та координатних осей проти годинникової стрілки*

**Висновки:** протягом виконання цієї лабораторної роботи я навчився будувати двовимірні зображення з подальшим їхнім переміщенням, обертанням та масштабуванням на основі використання базових геометричних перетворень OpenGL.