**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

**ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи №3

# на тему: “Створення анімаційних зображень з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM”

# 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: cтворення анімаційних зображень з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.

**Мета:** навчитися будувати двовимірні анімаційні зображення з використанням використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.

**Теоретичні відомості**

Для створення анімаційних зображень в OpenGL використовують основні геометричні перетворення – перенесення, обертання і масштабування, реалізовані у бібліотеці GLM. Шейдер вершини – це один з шейдерів, який може написати будь-яким програміст. Сучасний OpenGL вимагає, щоб ми налаштували, щонайменше, шейдери вершини і фрагмента, якщо плануємо виконати якийсь рендеринг.

Для того, чтоб OpenGL міг використати шейдер, він має динамично скомпілювати його зі свого вихідного коду під час виконання програми. Спочатку нам треба створити об’єкт шейдера з ідентифікатором, який посилається на нього. Для цього тип змінної шейдера вершини оголосимо як unsigned int і створимо сам шейдер за допомогою функції glCreateShader().

Далі передаємо тип шейдера, який ми хочемо створити, як аргумент функції glCreateShader() – у нашому випадку GL\_VERTEX\_SHADER. Потім прикріплюємо вихідний код шейдера до об’єкту шейдера і компілюємо його: glShaderSource(vertexShader, 1, &vertexShaderSource, NULL); glCompileShader(vertexShader);

Шейдер фрагмента – другий і останній шейдер, який ми створимо для рендерингу фігур, він потрібний для обчислення вихідного кольору пікселів.

Процес компіляції шейдера фрагмента аналогічний компіляції шейдера вершини, але цього разу як тип шейдера ми використовуємо константу GL\_FRAGMENT\_SHADER: unsigned int fragmentShader; fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER); glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL); glCompileShader(fragmentShader);

Нам залишається зв’язати два шейдерні об’єкти в шейдерну програму, яку ми зможемо використати для рендерингу.

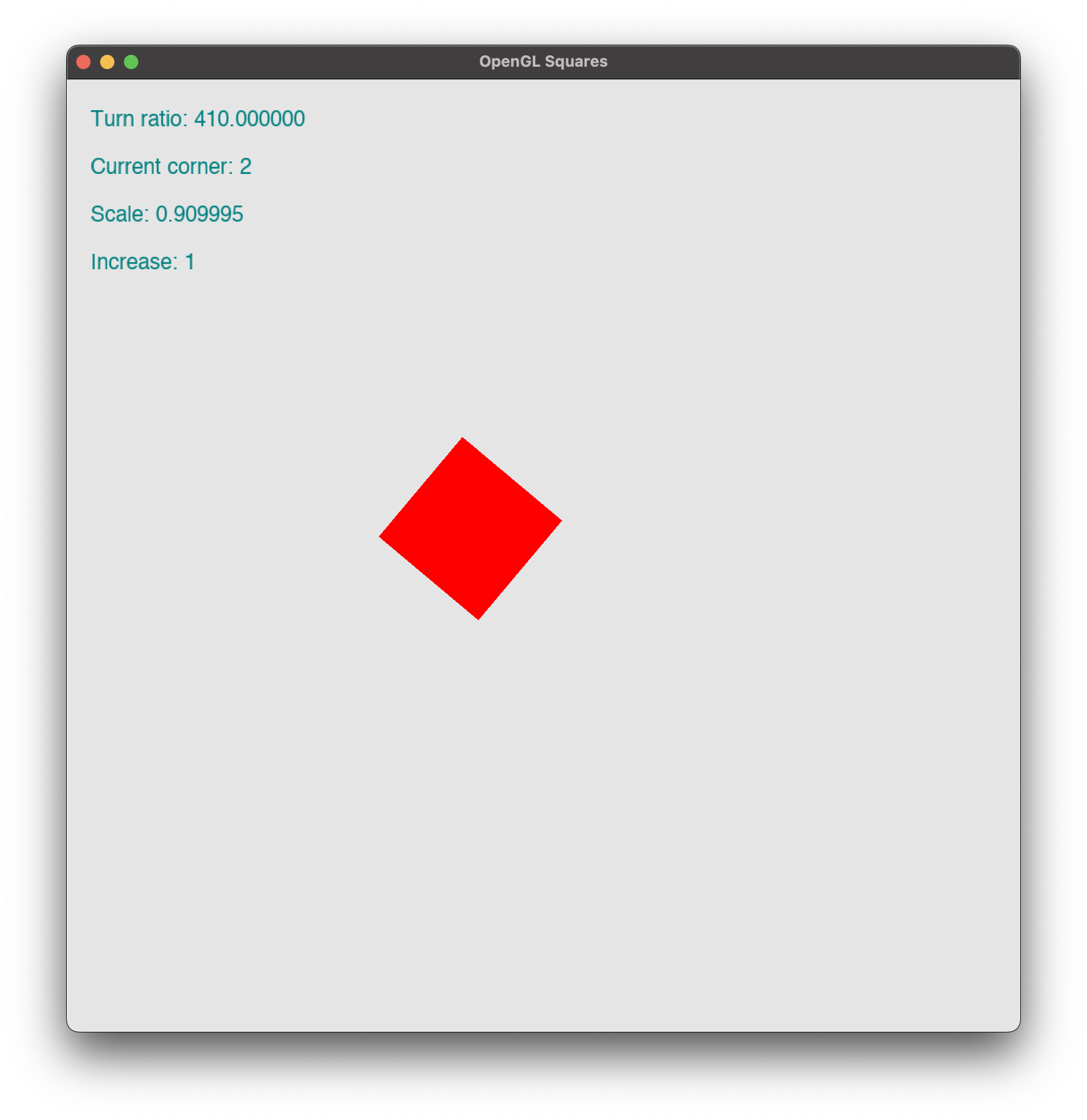
**Завдання**

14. Задано **чотирикутник**. Реалізувати послідовне обертання фігури за годинниковою стрілкою навколо кожної вершини з одночасним поступовим збільшенням на ***k*** відсотків і поверненням до попереднього розміру.

**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  
};  
  
enum ECorner {  
 TopLeft = 1,  
 BottomLeft,  
 BottomRight,  
 TopRight,  
};  
  
// Global variables  
GLuint shaderProgram, VAO, VBO;  
GLuint textShaderProgram, textVAO, textVBO;  
float scale = 1.0f;  
int windowWidth, windowHeight;  
float a = 1.0f;  
float k = 0.5f;  
bool increase = true;  
float turnRatio = 0.0f;  
float approxStep = 0.1f;  
bool playAnimation = false;  
ECorner currentCorner = static\_cast<ECorner>(0);  
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, "C:/Windows/Fonts/arial.ttf", 0, &face)) {  
 // std::cerr << "ERROR::FREETYPE: Failed to load font" << std::endl;  
 // return false;  
 // }  
  
 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 rotateModel(glm::mat4 &modelMatrix, float x, float y, glm::vec2 offset) {  
 // Step 1: Translate to the top-left corner (rotation point)  
 modelMatrix = glm::translate(modelMatrix, glm::vec3(offset.x + x, offset.y + y, 0.0f));  
  
 // Step 2: Apply rotation (around the Z-axis)  
 modelMatrix = glm::rotate(modelMatrix, glm::radians(turnRatio), glm::vec3(0.0f, 0.0f, 1.0f));  
  
 // Step 3: Translate back to the original position  
 modelMatrix = glm::translate(modelMatrix, glm::vec3(-x, -y, 0.0f));  
}  
  
void drawSquare(glm::vec2 offset, float squareSize, 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 square color  
 glUniform4f(glGetUniformLocation(shaderProgram, "squareColor"), color[0], color[1], color[2], color[3]);  
  
 float halfSize = squareSize \* 0.5f;  
  
 // Define the square's vertices (centered at origin for easier transformations)  
 std::vector<GLfloat> model = {  
 -halfSize, -halfSize, // Bottom-left  
 -halfSize, +halfSize, // Top-left  
 +halfSize, -halfSize, // Bottom-right  
 +halfSize, +halfSize // Top-right  
 };  
  
 // Create a transformation matrix  
 glm::mat4 modelMatrix = glm::mat4(1.0f); // Start with identity matrix  
  
 modelMatrix = glm::scale(modelMatrix, glm::vec3(k, k, 1.0f));  
  
 if (currentCorner == TopLeft) {  
 rotateModel(modelMatrix, -halfSize, halfSize, offset);  
 }  
 else if (currentCorner == BottomLeft) {  
 rotateModel(modelMatrix, -halfSize, -halfSize, offset);  
 }  
 else if (currentCorner == BottomRight) {  
 rotateModel(modelMatrix, halfSize, -halfSize, offset);  
 }  
 else if (currentCorner == TopRight) {  
 rotateModel(modelMatrix, halfSize, halfSize, offset);  
 }  
  
  
  
 // Apply the transformation to the vertices  
 for (size\_t i = 0; i < model.size(); i += 2) {  
 glm::vec4 vertex(model[i], model[i + 1], 0.0f, 1.0f); // Homogeneous coordinates  
 vertex = modelMatrix \* vertex; // Transform the vertex  
 model[i] = vertex.x; // Update the vertex position  
 model[i + 1] = vertex.y;  
 }  
  
 // Copy the transformed vertices to a std::vector<GLfloat>  
 std::vector<GLfloat> modelArray(model.begin(), model.end());  
  
 // Upload the vertex data to the GPU  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, modelArray.size() \* sizeof(GLfloat), modelArray.data(), GL\_DYNAMIC\_DRAW);  
  
 // Draw the square  
 glDrawArrays(GL\_TRIANGLE\_STRIP, 0, model.size() / 2);  
  
 glBindVertexArray(0);  
}  
  
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;  
 turnRatio -= 0.1f;  
 qPressed = true;  
 }  
 } else {  
 qPressed = false;  
 }  
  
 if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {  
 if (!wPressed) {  
 a -= 0.1f;  
 turnRatio += 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);  
  
 drawSquare({0.0f, 0.0f}, 0.3f, {1.0f, 0.0f, 0.0f, 1.0f});  
  
 if (playAnimation) {  
 turnRatio += 1.0f;  
  
 if (k < 0.5f) {  
 k = 0.5f;  
 increase = true;  
 }  
 if (k > 2.0f) {  
 k = 2.0f;  
 increase = false;  
 }  
  
 if (increase) {  
 k += 0.001f;  
 } else if (!increase) {  
 k -= 0.001f;  
 }  
  
 }  
  
 if (turnRatio >= 360 \* 4) {  
 turnRatio = 0.0f;  
 currentCorner = static\_cast<ECorner>(1);  
 }  
  
 if (turnRatio / 360 > static\_cast<float>(currentCorner)) {  
 currentCorner = static\_cast<ECorner>(currentCorner + 1);  
 }  
  
 std::string ratioString = "Turn ratio: " + std::to\_string(turnRatio);  
 std::string cornerString = "Current corner: " + std::to\_string(currentCorner);  
 std::string scaleString = "Scale: " + std::to\_string(k);  
 std::string increaseString = "Increase: " + std::to\_string(increase);  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
 renderText(ratioString, NDCToPixel(0.05f, true), NDCToPixel(1.9f, false), 1.5f, {0.0f, 0.5f, 0.5f});  
 renderText(cornerString, NDCToPixel(0.05f, true), NDCToPixel(1.8f, false), 1.5f, {0.0f, 0.5f, 0.5f});  
 renderText(scaleString, NDCToPixel(0.05f, true), NDCToPixel(1.7f, false), 1.5f, {0.0f, 0.5f, 0.5f});  
 renderText(increaseString, NDCToPixel(0.05f, true), NDCToPixel(1.6f, false), 1.5f, {0.0f, 0.5f, 0.5f});  
 glDisable(GL\_BLEND);  
  
 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 прямокутника навколо кожного з його кутів поступово та одночасне його збільшення*

**Висновки:** протягом виконання цієї лабораторної роботи я навчився будувати двовимірні анімаційні зображення з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.