**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

**ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи №1

**на тему:** “Візуалізація статичних 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:** Візуалізація статичних двовимірних зображень за допомогою бібліотеки OpenGL.

**Мета:** Вивчити методи роботи з найпростішими функціями бібліотеки OpenGL.

**Теоретичні відомості**

OpenGL (англ. Open Graphics Library – відкрита графічна бібліотека) – це потужний програмний інтерфейс, який використовують для одержання високоякісних, програмно генерованих зображень та інтерактивних застосунків, які описують дво- і тривимірні об’єкти, а також растрові зображення.

### **Складові OpenGL**

Імена функцій **основної бібліотеки**OpenGL починаються з префіксу gl, кожне слово, яке входить в ім’я функції, починається з великої букви, а самі функції зберігаються в бібліотеці, котру позначають GL. Крім основної, існує і декілька **додаткових бібліотек**.

**Бібліотека графічних утиліт** (GLU) включає стандартні функції, які дають змогу настроювати матриці проекції і візуалізації, описувати складні об’єкти наближено прямими і многокутниками, зображати квадратичні й бі-сплайни з допомогою лінійного наближення, зафарбовувати поверхні, тобто в її склад входять функції формування складних об’єктів типу сферичних поверхонь. Функції цієї бібліотеки звертаються тільки до функцій GL.

**OpenGL Utility Toolkit**(GLUT) містить функції для роботи з будь-якою системою вікон на екрані та методи, які дають змогу зафарбовувати криві і поверхні другого порядку. Вона забезпечує користувача основними можливостями, характерними для більшості сучасних багатовіконних систем.

### **Примітиви OpenGL**

За допомогою функцій примітивів, які є в основній бібліотеці OpenGL, можна будувати точки, прямолінійні відрізки, зафарбовані опуклі многокутники, а також піксельні чи растрові матричні структури. В бібліотеці GLUT можна знайти стандартні процедури для зображення рядків символів. Інші види примітивів, такі як кола, еліпси чи зафарбовані увігнуті багатокутні області, можна будувати за допомогою функцій основної бібліотеки або стандартними процедурами з бібліотек GLU і GLUT.

Базові примітиви OpenGL специфікують набором точок у просторі (вершин). Формат визначення об’єктів такий:

**glBegin (тип);**

**glVertex\* (…);**

**…**

**glEnd ();**

Значення «тип» визначає вигляд об’єкту і несе інформацію про те, яким чином треба інтерпретувати подальший список вершин. Функція **glVertex** повинна бути розміщена в програмі між функціями **glBegin** і **glEnd**. Аргумент функції **glBegin** визначає **тип графічного примітиву**:

1. **GL\_POINTS** – точка;
2. **GL\_LINES** – відрізок за 2-ма вершинами;
3. **GL\_LINE\_STRIP** – ламана лінія;
4. **GL\_POLYGON** – многокутник і т.д.

Коренева бібліотека OpenGL надає тільки випуклі многокутники, проте бібліотека GLU пропонує функції, що дають змогу працювати з увігнутими многокутниками та іншими невипуклими об’єктами з лінійними межами.

Під час візуалізації набору заданих у програмі об’єктів вони послідовно накладаються один на одний. Щоб видалити котрийсь із них, потрібно перемалювати весь набір примітивів, окрім зайвого.

### **Використання тексту**

У складі основної бібліотеки OpenGL нема спеціального примітиву для формування тексту, однак GLUT містить декілька визначених наборів символів та стандартні процедури для зображення шрифтів. **Символи растрового шрифту** визначаються на прямокутній області і є блоками бітів. Символи можна отримати за допомогою функції **glutBitmapCharacter(font, character)**; де **font** – значення символьної константи GLUT, яка вказує на визначений набір накреслень (шрифт з постійною шириною символів (моноширинний) – **GLUT\_BITMAP\_8\_BY\_13** тощо; шрифт з пропорційними проміжками (пропорційний) – **GLUT\_BITMAP\_TIMES\_ROMAN\_10**, **GLUT\_BITMAP\_ HELVETICA\_10** тощо); **character** – код ASCII або окремий символ, який потрібно зобразити (наприклад, код 65 або 'A'). Щоб задати координати поточного растрового положення, застосовують процедуру **glRasterPos\*(…)**; параметри й індекси аналогічні тим, що використовують у функції **glVertex**.

### **Налаштування кольорів**

Більшість налаштувань кольорів примітивів OpenGL виконується в режимі RGB. Поточні колірні компонент вибирають за допомогою функції **glColor\*(colorComponents)**; індекси аналогічні описаним для функції **glVertex**, причому перший може мати значення лише 3 або 4; колірні компоненти послідовно для червоного, зеленого та синього задають як числа від 0 до 1. Можливий четвертий параметр – коефіцієнт альфа – використовують для моделювання ефектів прозорості: коли він 0 – об’єкт повністю прозорий, 1 – непрозорий.

### **Загальна структура програми**

Мінімальний набір операцій, необхідний для створення зображення з використанням бібліотеки OpenGL, має такий вигляд.

1. Ініціалізація бібліотеки GLUT для створення сеансу зв’язку між підсистемою вікон і OpenGL командою **glutInit(&argc, argv)**.
2. Створення вікна зображення з відповідною назвою в рядку заголовка – **glutCreateWindow("<Ім’я вікна>")**. Для зміни параметрів вікна, встановлених системою за замовчуванням (розмір, розташування на екрані, колірний режим тощо), треба перед викликом **glutCreateWindow** звернутись до відповідних функцій бібліотеки GLUT (**glutInitDisplayMode**, **glutInitWindowPosition**, **glutInitWindowSize** і т.д.).
3. Передавання у вікно зображення графічних об’єктів, створених відповідною процедурою, за допомогою функції **glutDisplayFunc(<Ім’я процедури>)**.
4. Активізація всіх створених вікон зображень разом зі своїм графічним вмістом – **glutMainLoop()**. Ця функція, що має бути останнім оператором у програмі, відображає графічні об’єкти на екрані та запускає нескінченний цикл, в якому система приймає дані, що надходять від клавіатури чи миші. Це дає змогу організувати реакцію програми на дії користувача.

Процедури для створення рисунків, імена яких є аргументами для **glutDisplayFunc**, називають **функціями зворотного виклику зображень**. В кінці їхнього коду має стояти **glFlush()** – стандартна функція для прискорення виконання функцій OpenGL, записаних у буферах, що розташовані в різних місцях обчислювальної системи. В завантаженій мережі можуть виникати затримки обробки даних з деяких буферів, але виклик **glFlush** призводить до звільнення буферів та обробки функцій OpenGL.

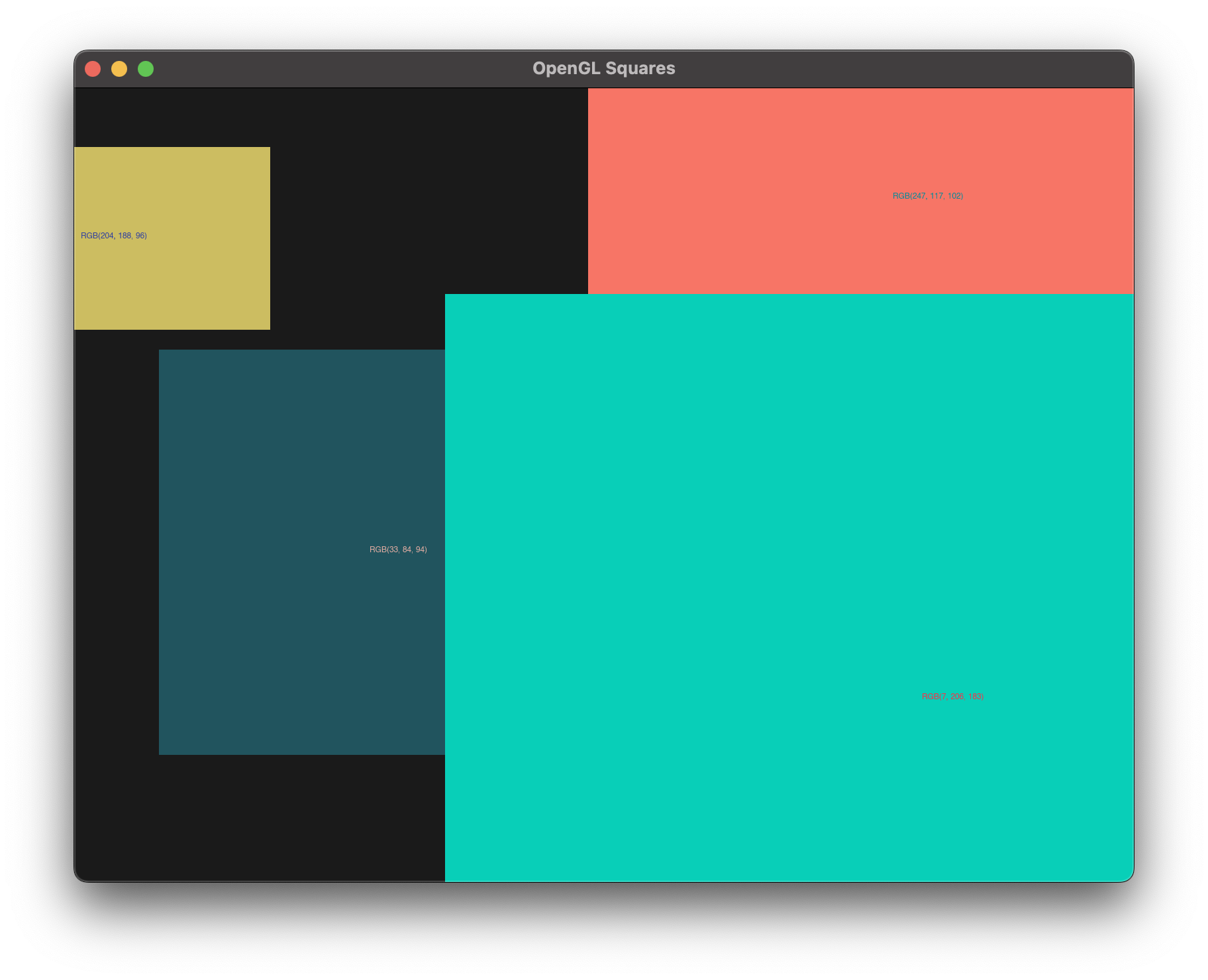
**Завдання**

13. Скласти програму побудови **чотирикутників** різних розмірів та кольорів у різних частинах вікна з використанням випадкового вибору. Підписати кожну фігуру кодом кольору. Послідовно витерти всі фігури та підписи.

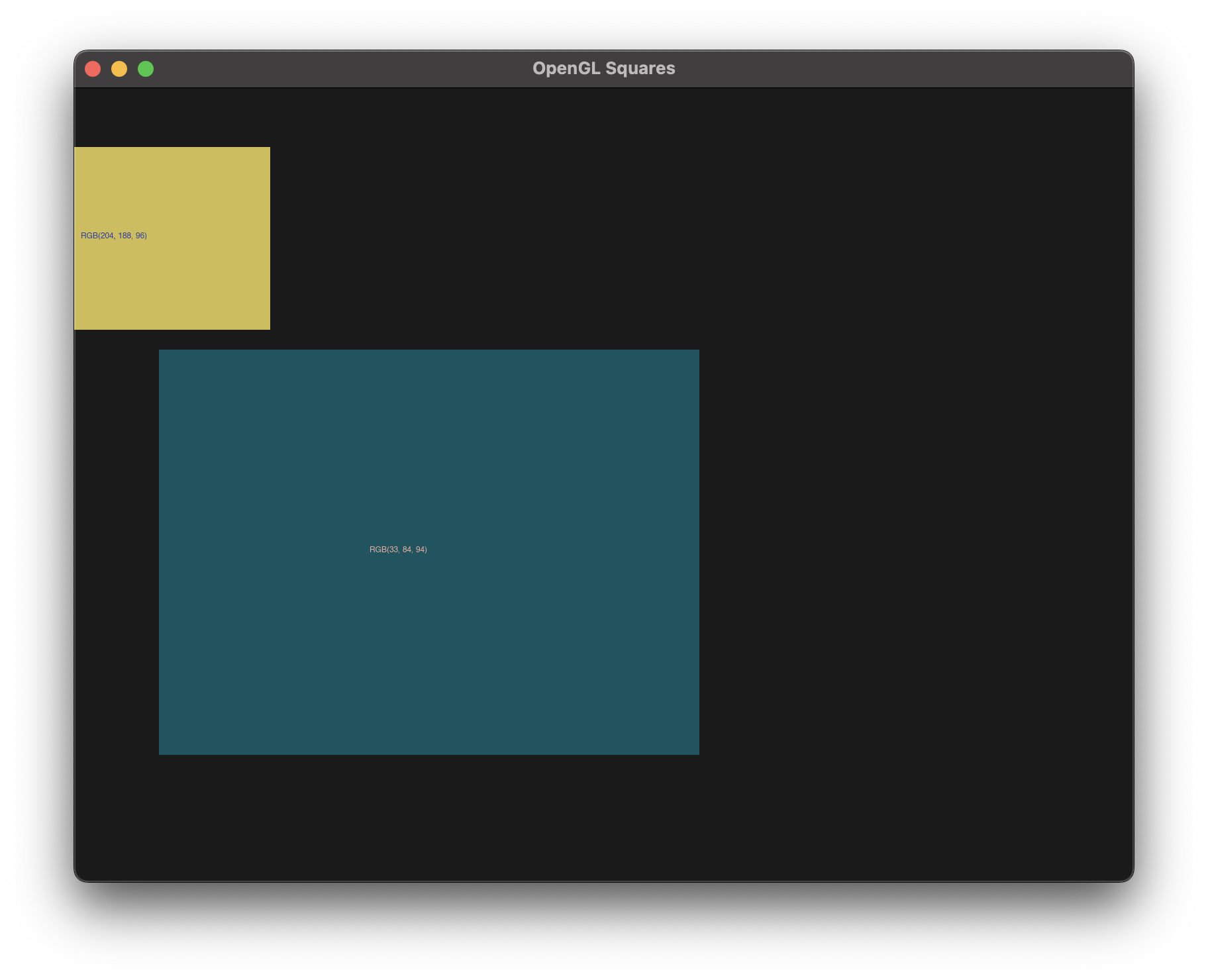
**Код програми**

#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;  
 }  
)";  
  
struct Square {  
 GLfloat x, y;  
 GLfloat size;  
 GLfloat color[3];  
};  
  
struct Character {  
 unsigned int TextureID;  
 glm::ivec2 Size;  
 glm::ivec2 Bearing;  
 unsigned int Advance;  
};  
  
GLuint shaderProgram, VAO, VBO;  
GLuint textShaderProgram, textVAO, textVBO;  
std::vector<Square> squares;  
std::map<char, Character> Characters;  
glm::mat4 projection;  
  
void renderText(const std::string& text, float x, float y, float scale, glm::vec3 color);  
  
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, 600, "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(0.1f, 0.1f, 0.1f, 1.0f); // Dark background  
 return true;  
}  
  
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 normalized device coordinates (NDC)  
 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) {  
 int width, height;  
 glfwGetFramebufferSize(glfwGetCurrentContext(), &width, &height);  
  
 // Convert from [-1, 1] to [0, width/height]  
 x = (x + 1.0f) \* 0.5f \* width;  
 y = (y + 1.0f) \* 0.5f \* height;  
}  
  
// Draw the squares with labels  
void drawSquares() {  
 // First draw all squares  
 glUseProgram(shaderProgram);  
 glBindVertexArray(VAO);  
  
 for (const auto& square : squares) {  
 // Set the square color  
 glUniform4f(glGetUniformLocation(shaderProgram, "squareColor"), square.color[0], square.color[1], square.color[2], 1.0f);  
  
 GLfloat model[] = {  
 square.x - square.size, square.y - square.size,  
 square.x + square.size, square.y - square.size,  
 square.x - square.size, square.y + square.size,  
 square.x + square.size, square.y + square.size  
 };  
  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, sizeof(model), model, GL\_STATIC\_DRAW);  
 glDrawArrays(GL\_TRIANGLE\_STRIP, 0, 4);  
 }  
  
 glBindVertexArray(0);  
  
 // Now render all text labels  
 glEnable(GL\_BLEND);  
 glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
  
 for (const auto& square : squares) {  
 GLfloat invertedColor[3];  
 invertColor(square.color, invertedColor); // Get the inverted color for better visibility  
  
 // Convert RGB values to text  
 std::stringstream ss;  
 ss << "RGB("  
 << static\_cast<int>(square.color[0] \* 255) << ", "  
 << static\_cast<int>(square.color[1] \* 255) << ", "  
 << static\_cast<int>(square.color[2] \* 255) << ")";  
  
 // Convert normalized coordinates to window coordinates for text rendering  
 float textX = square.x;  
 float textY = square.y;  
 normalizedToWindowCoords(textX, textY);  
  
 // Center the text on the square  
 textX -= ss.str().length() \* 6; // Adjust based on text length (approximation)  
  
 // Render the text with inverted color for better visibility  
 renderText(ss.str(), textX, textY, 0.5f, glm::vec3(invertedColor[0], invertedColor[1], invertedColor[2]));  
 }  
  
 glDisable(GL\_BLEND);  
}  
  
void handleKeyboardInput(GLFWwindow\* window) {  
 static bool zPressed = false; // Track if 'Z' was already pressed  
 static bool xPressed = false; // Track if 'X' was already pressed  
  
 // Check if the 'Z' key is pressed  
 if (glfwGetKey(window, GLFW\_KEY\_Z) == GLFW\_PRESS) {  
 if (!zPressed) { // Only trigger once on first press  
 // Create a new square with random properties  
 Square newSquare;  
 newSquare.x = static\_cast<float>(rand() % 200 - 100) / 100.0f; // Random position between -1.0 and 1.0  
 newSquare.y = static\_cast<float>(rand() % 200 - 100) / 100.0f;  
 newSquare.size = static\_cast<float>(rand() % 100 + 10) / 100.0f; // Random size between 0.1 and 1.0  
 newSquare.color[0] = static\_cast<float>(rand() % 100) / 100.0f;  
 newSquare.color[1] = static\_cast<float>(rand() % 100) / 100.0f;  
 newSquare.color[2] = static\_cast<float>(rand() % 100) / 100.0f;  
  
 squares.push\_back(newSquare); // Add the new square  
 zPressed = true; // Set the flag to indicate 'Z' has been pressed  
 }  
 } else {  
 zPressed = false; // Reset the flag when the key is released  
 }  
  
 // Check if the 'X' key is pressed  
 if (glfwGetKey(window, GLFW\_KEY\_X) == GLFW\_PRESS) {  
 if (!xPressed) { // Only trigger once on first press  
 // Delete the oldest square (if any)  
 if (!squares.empty()) {  
 squares.pop\_back();  
 }  
 xPressed = true; // Set the flag to indicate 'X' has been pressed  
 }  
 } else {  
 xPressed = false; // Reset the flag when the key is released  
 }  
}  
  
// Main loop for handling events and rendering  
void mainLoop(GLFWwindow\* window) {  
 while (!glfwWindowShouldClose(window)) {  
 glClear(GL\_COLOR\_BUFFER\_BIT);  
  
 handleKeyboardInput(window); // Check for keyboard input ('Z' and 'X')  
  
 drawSquares(); // Draw all squares with labels  
  
 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. Рендер чотирикутників з випадковими параметрами та відповідними підписами з кодом використаного кольору*

******

*Рис. 2. Стан програми після послідовно видалених двох чотирикутників*

**Висновки:** протягом виконання цієї лабораторної роботи я вивчив методи роботи з найпростішими функціями бібліотеки OpenGL. Оскільки на моїй системі примітиви не підтримуються я використав складнішу систему відмальовування графічних обʼєктів з використанням обʼєкта масиву вершинб обʼєкта буферу вершин, різних кастомних шейдерів та ортогональної матриці для рендеру тексту.