Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: Programowania w OpenGL z użyciem shader'ów, SFML obsługa klawiatury i myszki

Przedmiot: Wizualizacja Danych

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Marynowski Przemysław

Data: 28 października 2024

Numer lekcji: 4

Grupa laboratoryjna: 4

Cel Ćwiczenia

Celem ćwiczenia było zapoznanie się z programowaniem grafiki przy użyciu shader'ów oraz obsługą zdarzeń klawiatury i myszy.

Przebieg Ćwiczenia

Praca odbywała się w wcześniej dostarczonym przez prowadzącego kodzie źródłowym, będącym prostym programem wyświetlającym trójkąt z wierzchołkami o różnych kolorach na ekranie. Pierwszym zadaniem było dodanie kodu sprawdzającego poprawność kompilacji shader'ów oraz wyświetlenie komunikatów z zaistniałymi błędami na ekranie.

Logika zajmująca się powyższym zadaniem została umieszczona w funkcji `shader_compiled`, której zwracanym typem jest wartość prawda/fałsz typu `bool`.

Przyjmowany argumentami są:

- `GLuint shader` - sprawdzany Shader.
- `bool console_dump = true` – flaga decydująca o wydruku w konsoli.
- `std::string name_identifier = ""` – identyfikator w wydruku.

Definicja funkcji jest następująca:

```
bool shader_compiled(GLuint shader, bool console_dump = true, std::string name_identifier = "")
{
    // Check for compilation error
    GLint success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retreive the error log
        glGetShaderInfoLog(shader, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Shader Compilation Failed!:\n\t"
            << error_msg << "\n";
    }

    return success;
}
```

W przypadku, w którym kompilacja shader'ów zakończy się niepowodzeniem zasoby zostaną zwrócone, okienko zamknięte a program zakończy się z kodem błędu -1.

Poniżej znajduje się użycie funkcji `shader_compiled` w funkcji `main`:

```
// Create and compile the vertex shader
GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, &vertex_source, NULL);
glCompileShader(vertex_shader);

// Create and compile the fragment shader
GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1, &fragment_source, NULL);
glCompileShader(fragment_shader);

// Check for shader compilation
if (!shader_compiled(vertex_shader, true, "Vertex") ||
      !shader_compiled(fragment_shader, true, "Fragment"))
{
      // Cleanup: delete shaders, buffers, and close the window
      glDeleteShader(fragment_shader);
      glDeleteShader(vertex_shader);
      glDeleteBuffers(1, &vbo);
      glDeleteVertexArrays(1, &vao);

      window.close();  // Close the rendering window
      return -1;
}
```

Poniżej znajduje się przykładowy wydruk w sytuacji niepowodzenia:

```
ERROR: Vertex Shader Compilation Failed!:
       ERROR: 0:3: 'position' : syntax error syntax error
```

Adekwatnie postąpiono z kwestią łącznia shader'ów w program, tworząc funkcje weryfikującą oraz zakańczając pracę w przypadku niepowodzenia.

Funkcja weryfikująca:

```
bool program_linked(GLuint program, bool console_dump = true, std::string name_identifier = "")
{
    GLint success;
    glGetProgramiv(program, GL_LINK_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retrieve the error log
        glGetProgramInfoLog(program, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Program Linking Failed!:\n\t" <<
             error_msg << "\n";
    }

    return success;
}
```

Użycie funkcji weryfikującej:

```
// Use the program if linking succeded
if (program_linked(shader_program, true, "Shader"))
       glUseProgram(shader_program);
else
{
       // Cleanup: delete shaders, buffers, and close the window
       glDeleteProgram(shader_program);
       glDeleteShader(fragment_shader);
       glDeleteShader(vertex_shader);
       glDeleteBuffers(1, &vbo);
       glDeleteVertexArrays(1, &vao);

       window.close();  // Close the rendering window
       return -2;
}
```

Następnym zadaniem była zmiana wyświetlanego trójkąta na dowolną figurę 3D z wykorzystaniem współrzędnych cylindrycznych.

W tym celu wprowadzone zostały następujące zmiany:

- Zmiana w kodzie shader'ów wektora pozycji z 2D na 3D.
- Deklaracja stałej opisującej ilość danych na wierzchołek (6).
- Deklaracja zmiennej śledzącej ilość wierzchołków.
- Zmiana tablicy wierzchołków ze statycznej na dynamiczną.
- Dodanie funkcji, której zadaniem jest utworzenie figury z użyciem współrzędnych cylindrycznych i zapisaniem jej wierzchołków do tablicy.
- Losowanie kolorów wierzchołków wewnątrz tej funkcji.

Poniżej znajduje się pełna definicja wspomnianej funkcji.

```
void find_polygon_verts(GLfloat* vertices, int vert_num, float radius)
{
    // Starting angle and change of angles between every vert
    float start_angle = 0.0f;
    float angle_step = 2.0f * PI / vert_num;

    for (int i = 0; i < vert_num; i++)
    {
        // Angle of the current vert
        float angle = start_angle + i * angle_step;

        // Vertice coordinates
        vertices[i * DATA_PER_VERT] = radius * cos(angle);  // X
        vertices[i * DATA_PER_VERT + 1] = radius * sin(angle);  // Y
        vertices[i * DATA_PER_VERT + 2] = (float)rand() / RAND_MAX;  // Z

        // Colors
        vertices[i * DATA_PER_VERT + 3] = (float)rand() / RAND_MAX; // R
        vertices[i * DATA_PER_VERT + 4] = (float)rand() / RAND_MAX; // G
        vertices[i * DATA_PER_VERT + 5] = (float)rand() / RAND_MAX; // B
    }
}
```

Kolejnym zadaniem było zmiana prymitywu wyświetlanej figury z użyciem klawiszy numerycznych 1 - 0. W tym celu wprowadzone zostały następujące zmiany:

- Deklaracja stałej opisującej ilość dostępnych prymitywów (10).
- Deklaracja stałych tablic zawierających kody prymitywów w OpenGL oraz ich nazwy w postaci ciągów znaków.
- Przeniesienie tak zwanej „głównej pętli" okienka do osobnej funkcji dla poprawy czytelności.
- Zmiana stałego prymitywu podawanego funkcji `glDrawArrays` na zmienną.
- Nasłuchiwanie na wciśnięcie klawiszy numerycznych od 1 – 0 z użyciem systemu wydarzeń biblioteki SFML i zmiana prymitywu w zmiennej na prymityw w tablicy o odpowiadającym indeksie.

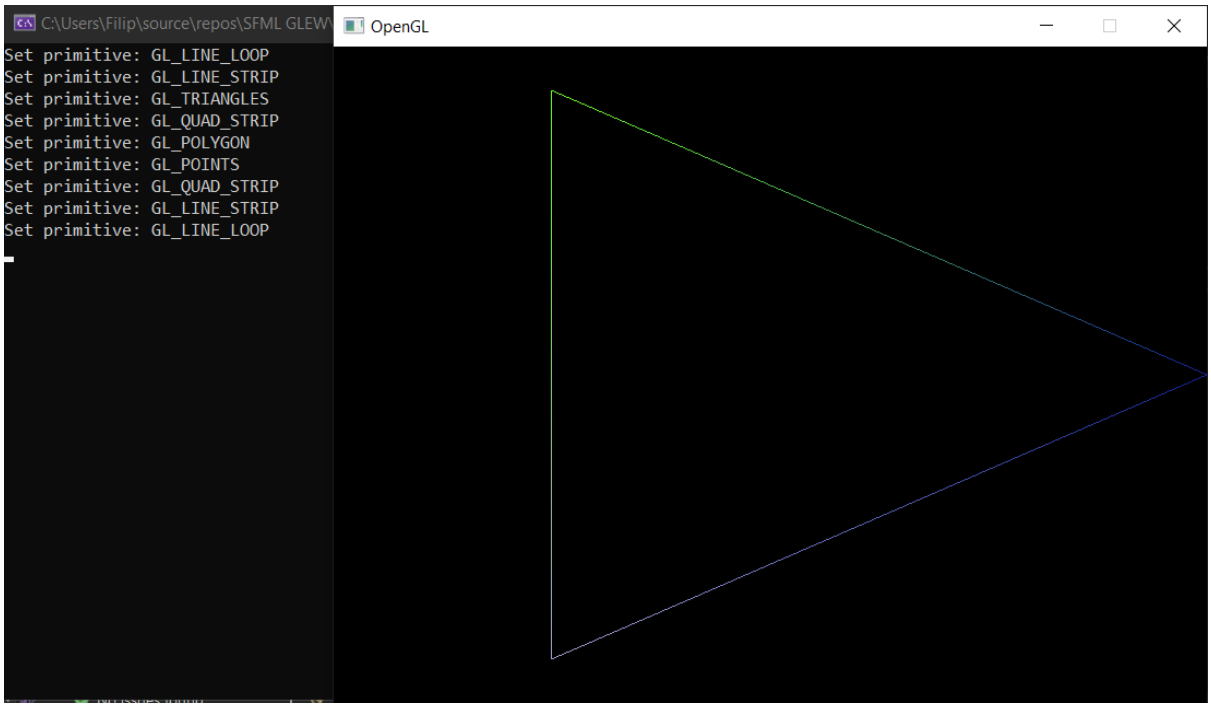
Poniżej znajdują się fragmenty funkcji **`main_loop`**, zajmujące się zmianą prymitywu i nasłuchiwaniem na kliknięcia.

```
GLenum used_primitive = GL_TRIANGLES;

// (...)

// Check if numerical key has been pressed
else if (window_event.key.code >= sf::Keyboard::Num0 && window_event.key.code <= sf::Keyboard::Num9)
{
	// Save numerical key as an integer
	int pressed_number = window_event.key.code - sf::Keyboard::Num0;
	used_primitive = primitives[pressed_number % primitives_num];
	std::cout << "Set primitive: " << primitives_names[used_primitive] << "\n";
}

// (...)

// Draw the shape
glDrawArrays(used_primitive, 0, vert_num);
```

Tablice prymitywów i ich nazw:

```
const int primitives_num = 10;
const GLenum primitives[primitives_num] =
{
    GL_POINTS,         // 0
    GL_LINES,          // 1
    GL_LINE_LOOP,      // 2
    GL_LINE_STRIP,     // 3
    GL_TRIANGLES,      // 4
    GL_TRIANGLE_STRIP,// 5
    GL_TRIANGLE_FAN,   // 6
    GL_QUADS,          // 7
    GL_QUAD_STRIP,     // 8
    GL_POLYGON         // 9
};

const std::string primitives_names[primitives_num] =
{
    "GL_POINTS",         // 0
    "GL_LINES",          // 1
    "GL_LINE_LOOP",      // 2
    "GL_LINE_STRIP",     // 3
    "GL_TRIANGLES",      // 4
    "GL_TRIANGLE_STRIP",// 5
    "GL_TRIANGLE_FAN",   // 6
    "GL_QUADS",          // 7
    "GL_QUAD_STRIP",     // 8
    "GL_POLYGON",        // 9
};
```

Przykładowy wydruk i efekt działania:



Ostatnim zadaniem było dodanie logiki pozwalającej na zmianę ilości wierzchołków poprzez wertykalny ruch myszą. Dodatkowo dodano również możliwość zmiany ilości wierzchołków poprzez klikanie strzałek na klawiaturze. W tych celach wykonano następujące czynności:

- Utworzono stałe opisujące maksymalną i minimalną ilość wierzchołków (1 - 18).
- Wewnątrz pętli głównej dodano nasłuchiwanie na ruch myszy oraz kliknięcia strzałek w górę i w dół na klawiaturze.
- Utworzenie funkcji `mouse_to_verts`, której zadaniem jest konwersja pozycji myszki na ilość wierzchołków.
- Utworzenie funkcji `update_vertices`, która zarządza pamięcią tablicy wierzchołków, tworzy nową figurę dla nowej ilości wierzchołków za pośrednictwem wcześniej zadeklarowanej funkcji `find_polygon_verts` oraz przekazuje nowe wierzchołki do GPU.
- Uzależnienie funkcji `glDrawArrays` od zmiennej śledzącej ilość wierzchołków.

Poniżej znajdują się fragmenty funkcji **`main_loop`** zajmujące się nasłuchiwaniem na ruch myszki i klawisze strzałek oraz rysowaniem figury na ekranie.

```
// (...)

else if (window_event.key.code == sf::Keyboard::Up)
{
	int new_vert_num = vert_num + 1;
	if (new_vert_num > MAX_VERTS)
		new_vert_num = MAX_VERTS;

	// Avoid unneccessary updates
	if (new_vert_num == vert_num)
		break;

	// Update vert number
	vert_num = new_vert_num;
	std::cout << "Vertices: " << vert_num << "\n";

	// Update the display
	vertices = update_vertices(vertices, vert_num, vbo);
}

// (...)

case sf::Event::MouseMoved:

// Convert mouse pos to vertices
int new_vert_num = mouse_to_verts(window_event.mouseMove.y);
if (new_vert_num == vert_num)   // Avoid updates if unneccessary
	break;

// Update vert number
vert_num = new_vert_num;
std::cout << "Vertices: " << vert_num << "\n";

// Update the display
vertices = update_vertices(vertices, vert_num, vbo);

break;

// (...)

// Draw the shape
glDrawArrays(used_primitive, 0, vert_num);
```

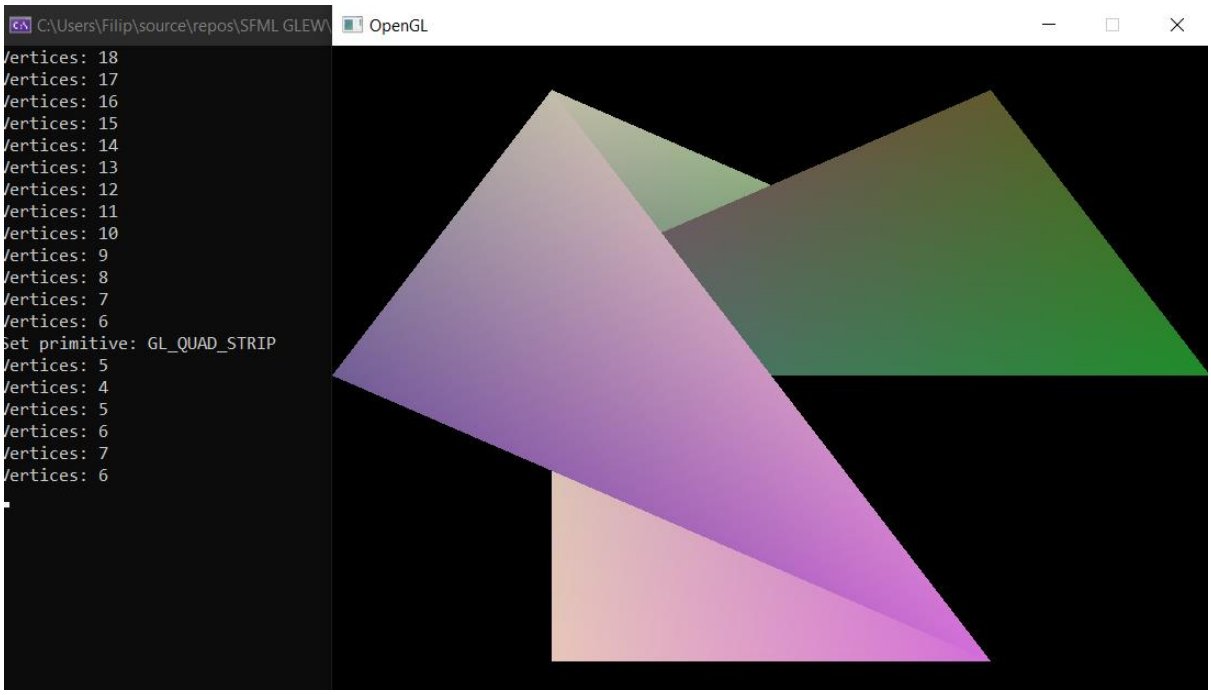
Definicja funkcji `mouse_to_verts`:

```
int mouse_to_verts(float mouse_pos_y)
{
    // Normalize the mouse Y position (0 at the top, 1 at the bottom)
    float normalized_mouse_y = mouse_pos_y / WINDOW_HEIGHT;

    // Invert the Y position so it progresses from bottom (0) to top (1)
    float top_down_mouse_y = 1.0f - normalized_mouse_y;

    // Calculate the number of vertices based on the mouse position within
the defined vertex range
    float vertex_range = MAX_VERTS - MIN_VERTS;
    float vertex_adj = vertex_range * top_down_mouse_y;

    // Set the vertex count by adjusting based on the mouse position
    int new_vert_num = (int)(MIN_VERTS + vertex_adj);

    return new_vert_num;
}
```

Definicja funkcji `update_vertices`:

```
GLfloat* update_vertices(GLfloat* vertices, int vert_num, GLuint vbo)
{
    // Reallocate memory for the new number of vertices
    delete[] vertices;
    vertices = new GLfloat[vert_num * DATA_PER_VERT];

    // Update vertices based on the new vertex count
    find_polygon_verts(vertices, vert_num, 1.0f);

    // Upload the updated vertex data to the GPU
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vert_num * DATA_PER_VERT *
sizeof(GLfloat), vertices, GL_DYNAMIC_DRAW);

    return vertices;
}
```

Przykładowy wydruk i efekt działania:



Pełny kod źródłowy

```
// Headers for OpenGL and SFML
// #include "stdafx.h"  // This line might be needed in some IDEs

#pragma once
#include <GL/glew.h>
#include <SFML/Window.hpp>
#include <iostream>
#include <time.h>
#include <cmath>

// Constants
// --------------------
const int primitives_num = 10;
const GLenum primitives[primitives_num] =
{
    GL_POINTS,        // 0
    GL_LINES,         // 1
    GL_LINE_LOOP,     // 2
    GL_LINE_STRIP,    // 3
    GL_TRIANGLES,     // 4
    GL_TRIANGLE_STRIP,// 5
    GL_TRIANGLE_FAN,  // 6
    GL_QUADS,         // 7
    GL_QUAD_STRIP,    // 8
    GL_POLYGON        // 9
};

const std::string primitives_names[primitives_num] =
{
    "GL_POINTS",        // 0
    "GL_LINES",         // 1
    "GL_LINE_LOOP",     // 2
    "GL_LINE_STRIP",    // 3
    "GL_TRIANGLES",     // 4
    "GL_TRIANGLE_STRIP",// 5
    "GL_TRIANGLE_FAN",  // 6
    "GL_QUADS",         // 7
    "GL_QUAD_STRIP",    // 8
    "GL_POLYGON",       // 9
};

const int DATA_PER_VERT = 6;
const double PI = 3.14159265358979323846;
const float WINDOW_WIDTH = 800.0;
const float WINDOW_HEIGHT = 600.0;
const int MIN_VERTS = 1;
const int MAX_VERTS = 18;

// Shaders
// --------------------

// Vertex shader source code
const GLchar* vertex_source = R"glsl(
#version 150 core
in vec3 position; // Input vertex position
in vec3 color;     // Input vertex color
out vec3 Color;    // Output color passed to the fragment shader

void main() {
    Color = color;  // Pass the color to the fragment shader
```

```
    gl_Position = vec4(position, 1.0);  // Set the position of the vertex
}
)glsl";

// Fragment shader source code
const GLchar* fragment_source = R"glsl(
#version 150 core
in vec3 Color;      // Color received from the vertex shader
out vec4 outColor;   // Output color to the framebuffer

void main() {
    outColor = vec4(Color, 1.0);  // Set the fragment color with full opacity
}
)glsl";

// Main loop functions
// --------------------
void find_polygon_verts(GLfloat* vertices, int vert_num, float radius)
{
    // Starting angle and change of angles between every vert
    float start_angle = 0.0f;
    float angle_step = 2.0f * PI / vert_num;

    for (int i = 0; i < vert_num; i++)
    {
        // Angle of the current vert
        float angle = start_angle + i * angle_step;

        // Vertice coordinates
        vertices[i * DATA_PER_VERT] = radius * cos(angle);  // X
        vertices[i * DATA_PER_VERT + 1] = radius * sin(angle);  // Y
        vertices[i * DATA_PER_VERT + 2] = (float)rand() / RAND_MAX;  // Z

        // Colors
        vertices[i * DATA_PER_VERT + 3] = (float)rand() / RAND_MAX; // R
        vertices[i * DATA_PER_VERT + 4] = (float)rand() / RAND_MAX; // G
        vertices[i * DATA_PER_VERT + 5] = (float)rand() / RAND_MAX; // B
    }
}

int mouse_to_verts(float mouse_pos_y)
{
    // Normalize the mouse Y position (0 at the top, 1 at the bottom)
    float normalized_mouse_y = mouse_pos_y / WINDOW_HEIGHT;

    // Invert the Y position so it progresses from bottom (0) to top (1)
    float top_down_mouse_y = 1.0f - normalized_mouse_y;

    // Calculate the number of vertices based on the mouse position within the defined vertex range
    float vertex_range = MAX_VERTS - MIN_VERTS;
    float vertex_adj = vertex_range * top_down_mouse_y;

    // Set the vertex count by adjusting based on the mouse position
    int new_vert_num = (int)(MIN_VERTS + vertex_adj);

    return new_vert_num;
}

GLfloat* update_vertices(GLfloat* vertices, int vert_num, GLuint vbo)
{
    // Reallocate memory for the new number of vertices
    delete[] vertices;
    vertices = new GLfloat[vert_num * DATA_PER_VERT];
```

```
    // Update vertices based on the new vertex count
    find_polygon_verts(vertices, vert_num, 1.0f);

    // Upload the updated vertex data to the GPU
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vert_num * DATA_PER_VERT * sizeof(GLfloat), vertices,
GL_DYNAMIC_DRAW);

    return vertices;
}

void main_loop(sf::Window& window, GLuint shader_program, GLuint vao, GLuint vbo, int vert_num,
GLfloat* vertices)
{
    bool running = true;
    GLenum used_primitive = GL_TRIANGLES;

    while (running)
    {
        sf::Event window_event;
        while (window.pollEvent(window_event))
        {
            switch (window_event.type)
            {
            case sf::Event::Closed:
                running = false;
                break;

            case sf::Event::KeyPressed:
                if (window_event.key.code == sf::Keyboard::Escape)
                {
                    running = false;
                }
                else if (window_event.key.code == sf::Keyboard::Up)
                {
                    int new_vert_num = vert_num + 1;
                    if (new_vert_num > MAX_VERTS)
                        new_vert_num = MAX_VERTS;

                    // Avoid unneccessary updates
                    if (new_vert_num == vert_num)
                        break;

                    // Update vert number
                    vert_num = new_vert_num;
                    std::cout << "Vertices: " << vert_num << "\n";

                    // Update the display
                    vertices = update_vertices(vertices, vert_num, vbo);
                }
                else if (window_event.key.code == sf::Keyboard::Down)
                {
                    int new_vert_num = vert_num - 1;
                    if (new_vert_num < MIN_VERTS)
                        new_vert_num = MIN_VERTS;

                    // Avoid unneccessary updates
                    if (new_vert_num == vert_num)
                        break;

                    // Update vert number
                    vert_num = new_vert_num;
```

```
                    std::cout << "Vertices: " << vert_num << "\n";

                    // Update the display
                    vertices = update_vertices(vertices, vert_num, vbo);
                }
                // Check if numerical key has been pressed
                else if (window_event.key.code >= sf::Keyboard::Num0 && window_event.key.code <=
sf::Keyboard::Num9)
                {
                    // Save numerical key as an integer
                    int pressed_number = window_event.key.code - sf::Keyboard::Num0;
                    used_primitive = primitives[pressed_number % primitives_num];
                    std::cout << "Set primitive: " << primitives_names[used_primitive] << "\n";
                }

                break;
            case sf::Event::MouseMoved:

                // Convert mouse pos to vertices
                int new_vert_num = mouse_to_verts(window_event.mouseMove.y);
                if (new_vert_num == vert_num)   // Avoid updates if unneccessary
                    break;

                // Update vert number
                vert_num = new_vert_num;
                std::cout << "Vertices: " << vert_num << "\n";

                // Update the display
                vertices = update_vertices(vertices, vert_num, vbo);

                break;
            }
        }

        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Draw the shape
        glDrawArrays(used_primitive, 0, vert_num);

        // Swap the front and back buffers
        window.display();
    }
}
// Validation functions
// --------------------
bool shader_compiled(GLuint shader, bool console_dump = true, std::string name_identifier = "")
{
    // Check for compilation error
    GLint success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retreive the error log
```

```
        glGetShaderInfoLog(shader, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Shader Compilation Failed!:\n\t" << error_msg
<< "\n";
    }

    return success;
}

bool program_linked(GLuint program, bool console_dump = true, std::string name_identifier = "")
{
    GLint success;
    glGetProgramiv(program, GL_LINK_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retrieve the error log
        glGetProgramInfoLog(program, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Program Linking Failed!:\n\t" << error_msg <<
"\n";
    }

    return success;
}

// Main function
// -------------------
int main()
{
    // Init for radom number generation
    srand(time(NULL));

    // Setup OpenGL context settings
    sf::ContextSettings settings;
    settings.depthBits = 24;     // Bits for depth buffer
    settings.stencilBits = 8;    // Bits for stencil buffer

    // Create a rendering window with OpenGL context
    sf::Window window(sf::VideoMode(WINDOW_WIDTH, WINDOW_HEIGHT, 32), "OpenGL", sf::Style::Titlebar |
sf::Style::Close, settings);

    // Initialize GLEW (must be done after creating the window and OpenGL context)
    glewExperimental = GL_TRUE;
    glewInit();

    // Create and bind a Vertex Array Object (VAO) to store vertex state
    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // Create a Vertex Buffer Object (VBO) and upload vertex data to it
    GLuint vbo;
    glGenBuffers(1, &vbo);
```

```
    // Vertex data: positions (x, y) and colors (r, g, b) for each vertex
    int vert_num = 3;
    GLfloat* vertices = new GLfloat[vert_num * DATA_PER_VERT];

    // Generate a polygon
    find_polygon_verts(vertices, vert_num, 1.0f);

    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vert_num * DATA_PER_VERT * sizeof(GLfloat), vertices,
GL_STATIC_DRAW);

    // Create and compile the vertex shader
    GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, &vertex_source, NULL);
    glCompileShader(vertex_shader);

    // Create and compile the fragment shader
    GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, &fragment_source, NULL);
    glCompileShader(fragment_shader);

    // Check for shader compilation
    if (!shader_compiled(vertex_shader, true, "Vertex") || !shader_compiled(fragment_shader, true,
"Fragment"))
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);
        glDeleteBuffers(1, &vbo);
        glDeleteVertexArrays(1, &vao);

        window.close();  // Close the rendering window
        return -1;
    }


    // Link both shaders into a single shader program
    GLuint shader_program = glCreateProgram();
    glAttachShader(shader_program, vertex_shader);
    glAttachShader(shader_program, fragment_shader);
    glBindFragDataLocation(shader_program, 0, "outColor");  // Bind fragment output
    glLinkProgram(shader_program);

    // Use the program if linking succeded
    if (program_linked(shader_program, true, "Shader"))
        glUseProgram(shader_program);
    else
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteProgram(shader_program);
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);
        glDeleteBuffers(1, &vbo);
        glDeleteVertexArrays(1, &vao);

        window.close();  // Close the rendering window
        return -2;
    }


    // Specify the layout of the vertex data
    GLint pos_attrib = glGetAttribLocation(shader_program, "position");
```

```
    glEnableVertexAttribArray(pos_attrib);
    glVertexAttribPointer(pos_attrib, 3, GL_FLOAT, GL_FALSE, DATA_PER_VERT * sizeof(GLfloat), 0);

    GLint col_attrib = glGetAttribLocation(shader_program, "color");
    glEnableVertexAttribArray(col_attrib);
    glVertexAttribPointer(col_attrib, 3, GL_FLOAT, GL_FALSE, DATA_PER_VERT * sizeof(GLfloat),
(void*)(3 * sizeof(GLfloat)));

    // Main event loop
    main_loop(window, shader_program, vao, vbo, vert_num, vertices);

    // Cleanup: delete shaders, buffers, and close the window
    glDeleteProgram(shader_program);
    glDeleteShader(fragment_shader);
    glDeleteShader(vertex_shader);
    glDeleteBuffers(1, &vbo);
    glDeleteVertexArrays(1, &vao);

    window.close();  // Close the rendering window
    return 0;
}
```