Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: Programowania w OpenGL z użyciem shader'ów, Transformacje, Kamera

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, użycie transformacji układu współrzędnych oraz obsługa kamery.

Przebieg Ćwiczenia

Kontynuowana była edycja wcześniejszego kodu z poprzedniego laboratorium. Pierwszym krokiem było pobranie i umieszczenie w projekcie biblioteki pomocniczej GLM, której zadaniem jest ułatwienie pracy z macierzami. Biblioteka została pobrana z repozytorium GitHub, manualnie umieszczona w katalogu projektu oraz skonfigurowana w jego właściwościach. Jej funkcjonalność została zawarta w projekcie.

Fragment nagłówków zawartych w main.cpp

```
#include <glm.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>
```

W kodzie źródłowym shadera wierzchołków zostały wprowadzone następujące zmiany:

- Dodanie referencji do macierzy modelu, widoku oraz projekcji.
- Modyfikacja formuły obliczania pozycji **gl_Position**, wykorzystująca wcześniej dodane macierze.

Kod źródłowy shadera wierzchołków

```
// Vertex shader takes care of positioning on the screen
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

// Set outside the shader
uniform mat4 model_matrix;  // Model
uniform mat4 view_matrix;   // View (camera)
uniform mat4 proj_matrix;   // Projection

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

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

Zadeklarowanie zmienne są typami **uniform**, co oznacza, że ich wartości są takie same dla każdego wierzchołka. Dodatkowo muszą one zostać manualnie przekazane programowi shadera, co zostało zrealizowane w funkcji `main()`.

- Z pomocą biblioteki GLM zadeklarowane zostały wcześniej wspomniane macierze przekształceń i wypełnione danymi.
- Macierze te zostały przekazane programowi shadera, po tym jak został on wywołany.

Zmienione fragmenty kodu w funkcji ***`main()`***

```
// Declare shader uniform data
glm::mat4 model_matrix = glm::mat4(1.0f);
model_matrix = glm::rotate(model_matrix, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));

glm::mat4 view_matrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3(0.0f, 0.0f,
0.0f), glm::vec3(0.0f, 1.0f, 0.0f));

glm::mat4 proj_matrix = glm::perspective(glm::radians(45.0f), WINDOW_WIDTH / WINDOW_HEIGHT,
0.01f, 100.0f);

// (...)

// Use the program if linking succeeded
if (program_linked(shader_program, true, "Shader"))
{
	// Debug info
	std::cout << separator;
	std::cout << "Version:\t" << glGetString(GL_VERSION) << "\n";
	std::cout << "Running on:\t" << glGetString(GL_RENDERER) << "\n";
	std::cout << separator;

	// Use the program
	glUseProgram(shader_program);

	// Add uniform data
	GLint uni_trans = glGetUniformLocation(shader_program, "model_matrix");
	glUniformMatrix4fv(uni_trans, 1, GL_FALSE, glm::value_ptr(model_matrix));

	GLint uni_view = glGetUniformLocation(shader_program, "view_matrix");
	glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));

	GLint uni_proj = glGetUniformLocation(shader_program, "proj_matrix");
	glUniformMatrix4fv(uni_proj, 1, GL_FALSE, glm::value_ptr(proj_matrix));
}
```

Kolejną częścią zadania było utworzenie kamery pierwszoosobowej. Następujące zmiany zostały wprowadzone do kodu źródłowego:

- Deklaracja zmiennych opisujących kamerę (pozycja oraz prędkość).
- Wywołanie w głównej pętli logiki śledzącej operacje wejścia klawiatury:
  - W = ruch w przód (przybliżenie).
  - S = Ruch w tył (oddalenie).
  - A = Ruch w lewo.
  - D = Ruch w prawo.
  - Q = Rotacja wokół własnej osi w lewo.
  - E = Rotacja wokół własnej osi w prawo.
- Symulacja zmiany położenia kamery poprzez transformacje wyświetlanego obrazu z wykorzystaniem macierzy i zmodyfikowanego shadera.
- Pozycja kamery jest aktualizowana tylko w sytuacji, w której ulegnie ona zmianie co pozwala zaoszczędzić na zasobach jednak eliminuje możliwość zastosowania inercji kamery, co może zostać zmienione w przyszłości.

Zmienne opisujące kamerę

```
// Camera
glm::vec3 camera_pos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 camera_front = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 camera_up = glm::vec3(0.0f, 1.0f, 0.f);
float camera_yaw = 0;
float camera_speed = 0.001f;
bool camera_pos_changed = false;    // Remove for damping implementation
```

Symulacja kamery poprzez transformacje obrazu

```
if (camera_pos_changed) // Remove check for damping implementation
{
	glm::mat4 view_matrix = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
	camera_front.x = sin(camera_yaw);
	camera_front.z = -cos(camera_yaw);

	GLint uniView = glGetUniformLocation(shader_program, "view_matrix");
		glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view_matrix));
	camera_pos_changed = false;
}
```

Fragment kodu odczytujący wejście z klawiatury

```
// Check camera movement keys in real-time
if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))   // Forward
{
      camera_pos += camera_speed * camera_front;
      camera_pos_changed = true;
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))   // Backwards
{
      camera_pos -= camera_speed * camera_front;
      camera_pos_changed = true;
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))   // Move left
{
      camera_pos -= glm::normalize(glm::cross(camera_front, camera_up)) *  camera_speed;
      camera_pos_changed = true;
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))   // Move right
{
      camera_pos += glm::normalize(glm::cross(camera_front, camera_up)) * camera_speed;
      camera_pos_changed = true;
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Q))   // Rotation left
{
      camera_yaw -= camera_speed;
      camera_pos_changed = true;
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::E))   // Rotation right
{
      camera_yaw += camera_speed;
      camera_pos_changed = true;
}
```

Ostatnią częścią zadania było narysowanie sześcianu na ekranie oraz włączenie Z-bufora.

- Dodanie stałych flag globalnych pozwalających na wyłączenie możliwości manipulacji kamery, ilości wierzchołków widocznych na ekranie oraz typu prymitywu.
- Dodanie globalnej tablicy ze wszystkimi wierzchołkami sześcianu utworzonego z trójkątów.
- Przypisanie tablicy sześcianu do tablicy wierzchołków przed przejściem do głównej pętli.

Flagi globalne kontrolujące funkcjonalność programu

```
// Flags
const bool enable_camera_manip = true;
const bool enable_vert_manip = false;
const bool enable_primitve_manip = false;
```

Przykład wykorzystania flagi

```
if (enable_vert_manip)
{
	// 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 unnecessary
		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);
}
```

Utworzenie sześcianu i przekazanie go karcie graficznej

```
// Generate a cube
vert_num = 36;
vertices = cube_vertices;

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

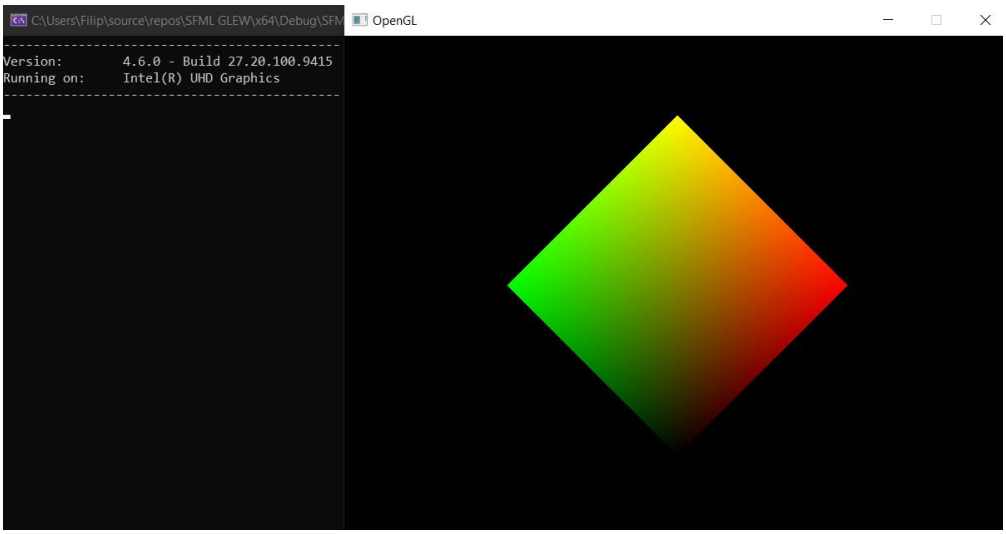
Włączenie Z-bufora

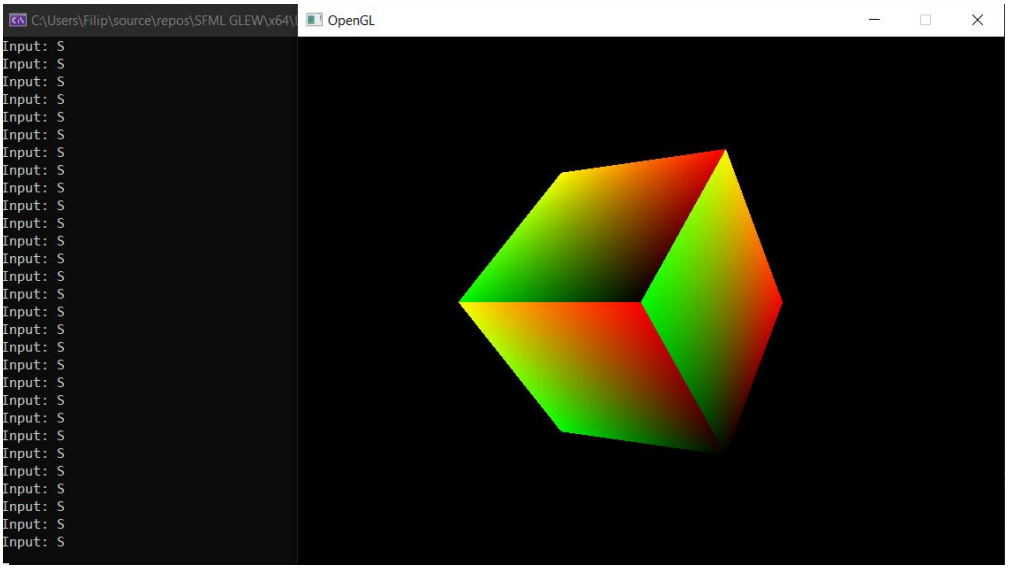
```
// Enabling Z-buffer
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

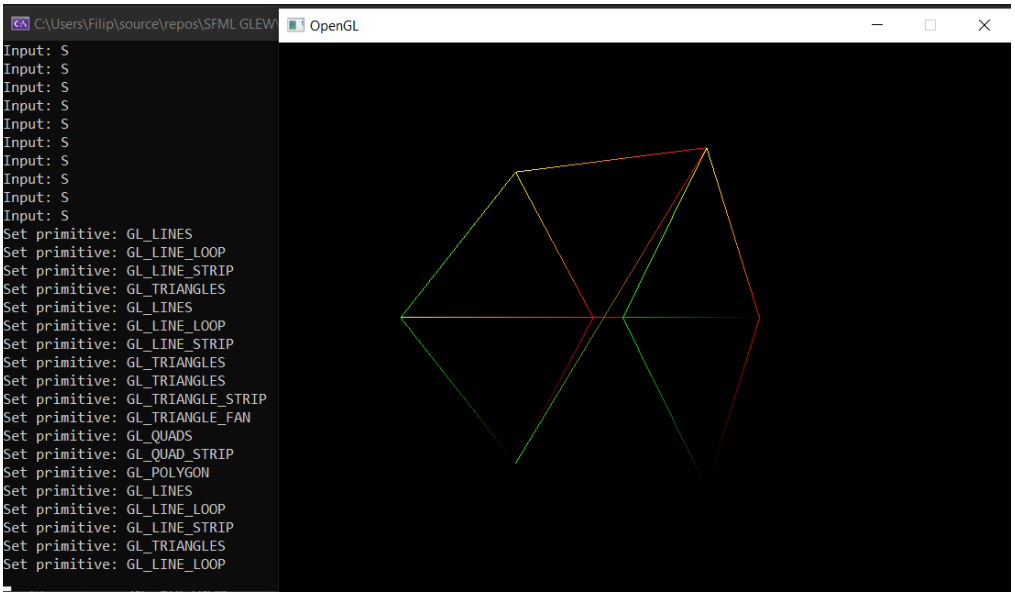
Modyfikacja czyszczenia ekranu

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

Przykład efektu pracy programu







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>
#include <glm.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>

// Constants
// --------------------

// Flags
const bool enable_camera_manip = true;
const bool enable_vert_manip = false;
const bool enable_primitve_manip = true;

// Primitives
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 = 36;
const std::string separator = std::string(45, '-') + "\n";
```

```
// Shaders
// --------------------

// Vertex shader takes care of positioning on the screen
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

// Set outside the shader
uniform mat4 model_matrix;  // Model
uniform mat4 view_matrix;   // View (camera)
uniform mat4 proj_matrix;   // Projection

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

    // Set the position of the vertex
    gl_Position = proj_matrix * view_matrix * model_matrix * vec4(position, 1.0);
}
)glsl";

// Fragment shader's job is to figure out area between surfaces
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";


// Shapes
GLfloat cube_vertices[] =
{
    // Front
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f,
    0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, 0.0f,

    // Rear
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,

    // Left
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
```

```
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,

    // Right
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,

    // Bottom
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,

    // Top
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f
};

// 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;

    // Camera
    glm::vec3 camera_pos = glm::vec3(0.0f, 0.0f, 3.0f);
    glm::vec3 camera_front = glm::vec3(0.0f, 0.0f, -1.0f);
    glm::vec3 camera_up = glm::vec3(0.0f, 1.0f, 0.f);
    float camera_yaw = 0;
    float camera_speed = 0.001f;
    bool camera_pos_changed = false;    // Remove for damping implementation

    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:
                // Exit condition
                if (window_event.key.code == sf::Keyboard::Escape)
                {
                    running = false;
                }

                // Vertice number manipulation
                if (enable_vert_manip)
                {
                    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 unnecessary 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);
                }

                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 unnecessary 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);
                }
            }

            if (enable_primitve_manip)
            {
                // Primitive manipulation
                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 %
primitives_num] << "\n";
                }
            }

            break;
        case sf::Event::MouseMoved:

            if (enable_vert_manip)
            {
                // 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 unnecessary
                    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;
    }
}

if (enable_camera_manip)
{
    // Check camera movement keys in real-time
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))   // Forward
    {
        camera_pos += camera_speed * camera_front;
        camera_pos_changed = true;
        std::cout << "Input: W\n";
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))   // Backwards
    {
        camera_pos -= camera_speed * camera_front;
        camera_pos_changed = true;
        std::cout << "Input: S\n";
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))   // Move left
    {
        camera_pos -= glm::normalize(glm::cross(camera_front, camera_up)) * camera_speed;
        camera_pos_changed = true;
        std::cout << "Input: A\n";
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))   // Move right
    {
        camera_pos += glm::normalize(glm::cross(camera_front, camera_up)) * camera_speed;
        camera_pos_changed = true;
        std::cout << "Input: D\n";
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Q))   // Rotation left
    {
        camera_yaw -= camera_speed;
        camera_pos_changed = true;
        std::cout << "Input: Q\n";
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::E))   // Rotation right
    {
        camera_yaw += camera_speed;
        camera_pos_changed = true;
        std::cout << "Input: E\n";
    }

    if (camera_pos_changed) // Remove check for damping implementation
    {
        glm::mat4 view_matrix = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
        camera_front.x = sin(camera_yaw);
        camera_front.z = -cos(camera_yaw);

        GLint uniView = glGetUniformLocation(shader_program, "view_matrix");
        glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view_matrix));
        camera_pos_changed = false;
    }
}
```

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

    // Enabling Z-buffer
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    // 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);

    // Generate a cube
    vert_num = 36;
    vertices = cube_vertices;

    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;
    }

    // Declare shader uniform data
    glm::mat4 model_matrix = glm::mat4(1.0f);
    model_matrix = glm::rotate(model_matrix, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
    glm::mat4 view_matrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3(0.0f, 0.0f, 0.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 proj_matrix = glm::perspective(glm::radians(45.0f), WINDOW_WIDTH / WINDOW_HEIGHT, 0.01f,
100.0f);

    // 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 succeeded
    if (program_linked(shader_program, true, "Shader"))
    {
        // Debug info
        std::cout << separator;
        std::cout << "Version:\t" << glGetString(GL_VERSION) << "\n";
        std::cout << "Running on:\t" << glGetString(GL_RENDERER) << "\n";
        std::cout << separator;

        // Use the program
        glUseProgram(shader_program);

        // Add uniform data
        GLint uni_trans = glGetUniformLocation(shader_program, "model_matrix");
        glUniformMatrix4fv(uni_trans, 1, GL_FALSE, glm::value_ptr(model_matrix));

        GLint uni_view = glGetUniformLocation(shader_program, "view_matrix");
        glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));

        GLint uni_proj = glGetUniformLocation(shader_program, "proj_matrix");
        glUniformMatrix4fv(uni_proj, 1, GL_FALSE, glm::value_ptr(proj_matrix));
    }
    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;
}
```