Project 3 Report

D11315807
Ardiawan Bagus Harisa
Department of CSIE

Tasks:

1. Define a triangle using 3 vertices (position only).

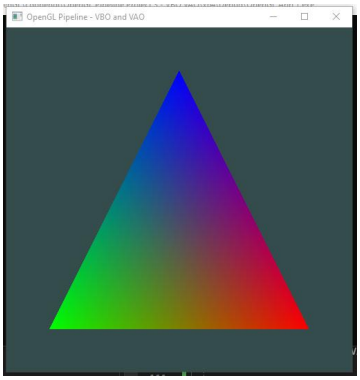
2. Create and bind a VAO and VBO.

3 Set up the vertex attribute pointer.

4. Use glDrawArrays() to draw.

How to use my program:

1. First, you must have the freeglut and glew library installed.
2. For my convenience, I use VS Studio for debugging.
3. Just run the debug by pressing **F5**. You will get the following result:

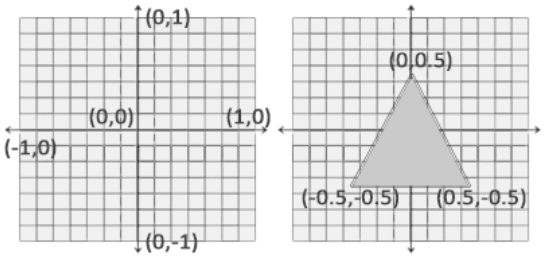


Program:

1. **Define a triangle**

First, we need to provide the points/ vertices as input to define the triangle. Following this tutorial: https://learnopengl.com/Getting-started/Hello-Triangle, we now have three vertices. From all points, I set the z to be 0 still. Because I only do 2D projection.

```
95  void initBuffers() {
96      float vertices[] = {
97          0.0f,  0.5f, 0.0f,  // Top
98         -0.5f, -0.5f, 0.0f,  // Bottom Left
99          0.5f, -0.5f, 0.0f   // Bottom Right
100     };
101     // ...
```



2. Create VAO and VBO

The next step is to create the VAO and VBO. First, I must define the variable for VAO and VBO as a global variable.

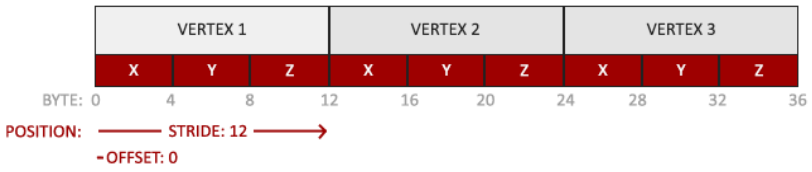
```
25  GLuint shaderProgram;
26  GLuint VAO;
27  GLuint VBO;
```

Later on, I must initialize the VAO and VBO after I define the triangle vertices. Because I will push the triangle vertices to the GPU using VBO and VAO (which store vertex data). First, I create VAO that later will be used to store the settings of vertex attributes and related buffer. Then, also create VBO to hold the actual vertex data (pos, color, normal). And bind it.

```
81
82      glGenVertexArrays(1, &VAO);
83      glGenBuffers(1, &VBO);
84
85      // Binding of VAO and VBO
86      glBindVertexArray(VAO);
87      glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

The vertex buffer data is formatted as follows:

- The position data is stored as 32-bit (4 byte) floating point values.
- Each position is composed of 3 of those values.
- There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
- The first value in the data is at the beginning of the buffer.



3. Setup vertex attribute

The line glBufferData() is used to push the vertex data to the GPU with the array buffer as the target type, with the following details: the size matches the vertices, point to the vertices data, then using static drawing to optimize storage because it will not change frequently. Then setup the configurations for the vertex: 3 components per vertex, float type. Finally, I need to unbind VBO and then VAO to avoid modification.

```
88
89      // Upload vertex data
90      glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
91
92      // Set vertex attribute pointers
93      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
94      glEnableVertexAttribArray(0);
95
96      // Unbind
97      glBindBuffer(GL_ARRAY_BUFFER, 0);
98      glBindVertexArray(0);
99  }
100
```

Put all those line in a initialization function *initBuffers()*.

4. Draw

To draw to the screen, I use *display()* function, similar to the previous projects. Typically, we must clear the buffer using *glClear()*. The *glUseProgram()* is use to activates the shader program from *shaderProgram*. It contains compiled and linked vertex and fragment shaders. Meaning it defines the process of vertices and pixels.

The *glBindVertexArray(VAO)* is used to binds the VAO again because it contains previous configuration and the detail of associated VBO from the init function. Then, use *glDrawArrays()* to draw the triangle with 0 as the starting index in the vertex array, and number of vertices = 3. Then call glFlush to execute the OpenGL commands.

```
101 void display() {
102     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);   // Background
103     glClear(GL_COLOR_BUFFER_BIT);
104
105     // Use our shader and draw
106     glUseProgram(shaderProgram);
107     glBindVertexArray(VAO);
108     glDrawArrays(GL_TRIANGLES, 0, 3);
109
110     glFlush();
111 }
```

And here is the detail of the *shaderProgram*.

Define the variable of shaderProgram to hold the shader program. First, try to compile the vertex shader, later followed by fragment and link shader compilations.

```
GLuint shaderProgram;
GLuint VAO;
GLuint VBO;

void initShaders() {
    // Compile Vertex Shader
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);

    GLint success;
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cerr << "Vertex Shader Compilation Failed\n" << infoLog << std::endl;
    }

    // Compile Fragment Shader
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);

    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cerr << "Fragment Shader Compilation Failed\n" << infoLog << std::endl;
    }
```

Vertex shader: create new shader object, specifically vertex shader. It will process each vertex from the triangle and define its position in the screen space. Here, we can say that *vertexShader* contains the program (source code) for the shader. Then, try to compile the *vertexShader* program. We can retrieve the error status from *glGetShaderiv()*, and pass the status to the success pointer. In case of error, print on the console window. The same goes with the fragment shader.

```
    // Link Shaders
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);

    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cerr << "Shader Program Linking Failed\n" << infoLog << std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
}
```

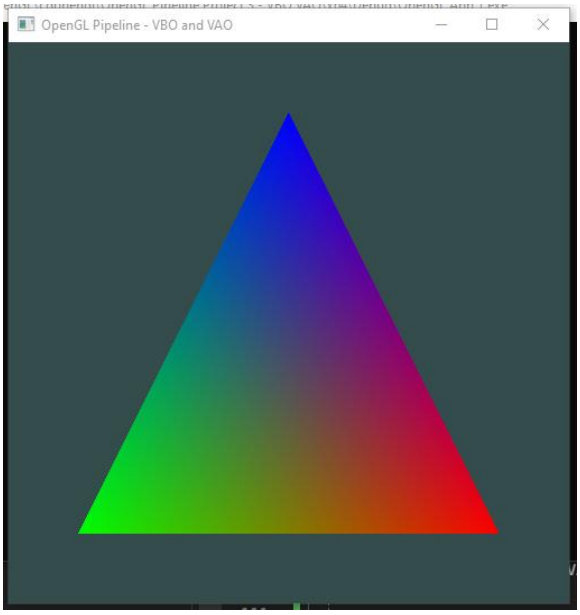
After vertex and fragment shader compilation, we must link those shaders to the shader program. First, I create a shader program using *glCreateProgram(),* then attach those shaders to the program. In case of error, just debug on the console. Then, delete the artifacts (vertexShader and fragmentShader) once the shaders are linked, to free up some GPU space.

Then, just call the display function to the main function.

```
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitContextVersion(3, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("OpenGL Pipeline - VBO and VAO");

    glewExperimental = GL_TRUE;
    GLenum err = glewInit();
    if (err != GLEW_OK) { ... }

    initShaders();
    initBuffers();
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

It will create this triangle:

Now, let’s edit some things: color and size.



Source code:

https://github.com/ardiawanbagusharisa/cgopengl/tree/main/OpenGL%20Pipeline%20Project%203%20-%20VBO%20VAO

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <iostream>

// Vertex Shader Source
//const char* vertexShaderSource = R"(
//#version 330 core
//layout(location = 0) in vec3 aPos;
//
//void main() {
//    gl_Position = vec4(aPos, 1.0);
//}
//)";
const char* vertexShaderSource = R"(
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aColor;

out vec3 vertexColor;

void main() {
    gl_Position = vec4(aPos, 1.0);
    vertexColor = aColor;
}
)";


// Fragment Shader Source
//const char* fragmentShaderSource = R"(
//#version 330 core
//out vec4 FragColor;
//
//void main() {
//    FragColor = vec4(1.0, 0.5, 0.2, 1.0); // Orange color
//}
//)";
const char* fragmentShaderSource = R"(
#version 330 core
in vec3 vertexColor;
out vec4 FragColor;
```

```
void main() {
    FragColor = vec4(vertexColor, 1.0);
}
)";


GLuint shaderProgram;
GLuint VAO;
GLuint VBO;

void initShaders() {
    // Compile Vertex Shader
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);

    GLint success;
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cerr << "Vertex Shader Compilation Failed\n" << infoLog <<
std::endl;
    }

    // Compile Fragment Shader
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);

    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cerr << "Fragment Shader Compilation Failed\n" << infoLog <<
std::endl;
    }

    // Link Shaders
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);

    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
```

```
        std::cerr << "Shader Program Linking Failed\n" << infoLog << std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
}

void initBuffers() {
    //float vertices[] = {
    //     0.0f,  0.5f, 0.0f,  // Top
    //    -0.5f, -0.5f, 0.0f,  // Bottom Left
    //     0.5f, -0.5f, 0.0f   // Bottom Right
    //};
    // Now add RGB color to each vertex (x, y, z, r, g, b)
    float vertices[] = {
        // positions         // colors
         0.0f,  0.75f, 0.0f,   0.0f, 0.0f, 1.0f, // Top - Blue
        -0.75f, -0.75f, 0.0f,   0.0f, 1.0f, 0.0f, // Bottom Left - Green
         0.75f, -0.75f, 0.0f,   1.0f, 0.0f, 0.0f  // Bottom Right - Red
    };

    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    // Binding of VAO and VBO
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    // Upload vertex data
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    //// Set vertex attribute pointers
    //glVertexAttribPointer(0,  3,  GL_FLOAT,  GL_FALSE,  3  *  sizeof(float),
(void*)0);
    //glEnableVertexAttribArray(0);
    // Position attribute (location = 0)
    glVertexAttribPointer(0,  3,  GL_FLOAT,  GL_FALSE,  6  *  sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    // Color attribute (location = 1)
    glVertexAttribPointer(1,  3,  GL_FLOAT,  GL_FALSE,  6  *  sizeof(float),
(void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);


    // Unbind
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
```

```
}

void display() {
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);   // Background
    glClear(GL_COLOR_BUFFER_BIT);

    // Use our shader and draw
    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitContextVersion(3, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("OpenGL Pipeline - VBO and VAO");

    glewExperimental = GL_TRUE;
    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "GLEW Error: " << glewGetErrorString(err) << std::endl;
        return -1;
    }

    initShaders();
    initBuffers();
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```