Roll No: 1703016

Lab Performance Test 1

Lab Task Q1

Question: Show an OpenGL Program which will show a red isosceles triangle.

Solution (Bold your own written code):

```
// Q1. Show an OpenGL Program which will show a red
isosceles triangle.
// roll: 1703016

#include "glad.h"
#include "glfw3.h"

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int
width, int height);
void processInput(GLFWwindow *window);

// settings
// window height width change korbo
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z,
1.0);\n"
    "}\0";
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);\n" //
triangle color change korbo; format: red green blue
opacity
    "}\n\0";

int main()
{
    // glfw: initialize and configure
```

```
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if
(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" <<
std::endl;
        return -1;
    }


    // build and compile our shader program
    // ------------------------------------
    // vertex shader
    unsigned int vertexShader =
glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource,
NULL);
    glCompileShader(vertexShader);
    // check for shader compile errors
```

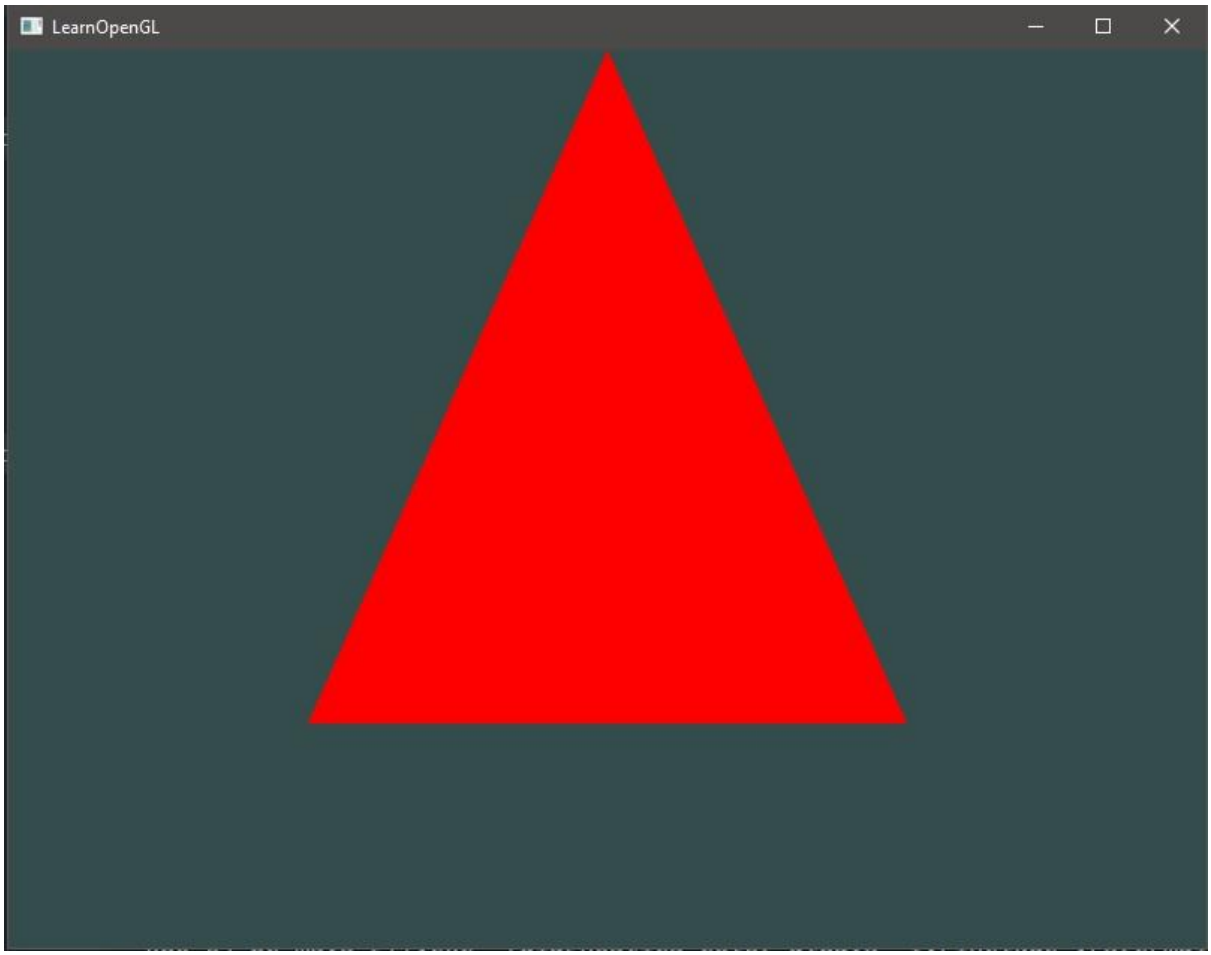
```
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS,
&success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL,
infoLog);
        std::cout <<
"ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog
<< std::endl;
    }
    // fragment shader
    unsigned int fragmentShader =
glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1,
&fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);
    // check for shader compile errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS,
&success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL,
infoLog);
        std::cout <<
"ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" <<
infoLog << std::endl;
    }
    // link shaders
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    // check for linking errors
    glGetProgramiv(shaderProgram, GL_LINK_STATUS,
&success);
    if (!success) {
        glGetProgramInfoLog(shaderProgram, 512, NULL,
infoLog);
        std::cout <<
"ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog <<
std::endl;
    }
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
```

```
    // set up vertex data (and buffer(s)) and configure
vertex attributes
    // ------------------------------------------------
----------------
    float vertices[] = {
        -0.5f, -0.5f, 0.0f, // left
         0.5f, -0.5f, 0.0f, // right
         0.0f,  1.0f, 0.0f,  // top
    };

    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    // bind the Vertex Array Object first, then bind and
set vertex buffer(s), and then configure vertex
attributes(s).
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // note that this is allowed, the call to
glVertexAttribPointer registered VBO as the vertex
attribute's bound vertex buffer object so afterwards we
can safely unbind
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // You can unbind the VAO afterwards so other VAO
calls won't accidentally modify this VAO, but this
rarely happens. Modifying other
    // VAOs requires a call to glBindVertexArray anyways
so we generally don't unbind VAOs (nor VBOs) when it's
not directly necessary.
    glBindVertexArray(0);


    // uncomment this call to draw in wireframe
polygons.
    //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

```
    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
        // input
        // -----
        processInput(window);

        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f); //
background color change korbo
        glClear(GL_COLOR_BUFFER_BIT);

        // draw our first triangle
        glUseProgram(shaderProgram);
        glBindVertexArray(VAO); // seeing as we only
have a single VAO there's no need to bind it every time,
but we'll do so to keep things a bit more organized
        glDrawArrays(GL_TRIANGLES, 0, 3);
        // glBindVertexArray(0); // no need to unbind it
every time

        // glfw: swap buffers and poll IO events (keys
pressed/released, mouse moved etc.)
        // -------------------------------------------
-------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de-allocate all resources once they've
outlived their purpose:
    // ------------------------------------------------
----------------------
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);

    // glfw: terminate, clearing all previously
allocated GLFW resources.
    // ------------------------------------------------
-----------------
    glfwTerminate();
    return 0;
}
```

```
// process all input: query GLFW whether relevant keys
are pressed/released this frame and react accordingly
// ------------------------------------------------------
------------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) ==
GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}

// glfw: whenever the window size changed (by OS or user
resize) this callback function executes
// ------------------------------------------------------
---------------------------------------
void framebuffer_size_callback(GLFWwindow* window, int
width, int height)
{
    // make sure the viewport matches the new window
dimensions; note that width and
    // height will be significantly larger than
specified on retina displays.
    glViewport(0, 0, width, height);
}
```

Output:



Lab Task Q2

Question: Show an OpenGL Program which will show three different triangles with red, black and blue color in green background.

Solution (Bold your own written code):

```
// Q2. Show an OpenGL Program which will show three
different triangles with red, black and blue color in
green background.
// roll: 1703016

#include "glad.h"
#include "glfw3.h"

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int
width, int height);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z,
1.0);\n"
    "}\0";
const char *fragmentShader1Source = "#version 330
core\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);\n" //
red triangle
    "}\n\0";
const char *fragmentShader2Source = "#version 330
core\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(0.0f, 0.0f, 0.0f, 1.0f);\n" //
```

```
black triangle
    "}\n\0";
const char *fragmentShader3Source = "#version 330
core\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(0.0f, 0.0f, 1.0f, 1.0f);\n" //
blue triangle
    "}\n\0";


int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if
(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
```

```
        std::cout << "Failed to initialize GLAD" <<
std::endl;
        return -1;
    }


    // build and compile our shader program
    // ------------------------------------
    // we skipped compile log checks this time for
readability (if you do encounter issues, add the
compile-checks! see previous code samples)
    unsigned int vertexShader =
glCreateShader(GL_VERTEX_SHADER);
    unsigned int fragmentShaderRed =
glCreateShader(GL_FRAGMENT_SHADER); // the first
fragment shader that outputs the color red
    unsigned int fragmentShaderBlack =
glCreateShader(GL_FRAGMENT_SHADER); // the second
fragment shader that outputs the color black
    unsigned int fragmentShaderBlue =
glCreateShader(GL_FRAGMENT_SHADER); // black

    unsigned int shaderProgramRed = glCreateProgram();
    unsigned int shaderProgramBlack = glCreateProgram();
// the second shader program
    unsigned int shaderProgramBlue = glCreateProgram();
// third

    glShaderSource(vertexShader, 1, &vertexShaderSource,
NULL);
    glCompileShader(vertexShader);
    glShaderSource(fragmentShaderRed, 1,
&fragmentShader1Source, NULL);
    glCompileShader(fragmentShaderRed);
    glShaderSource(fragmentShaderBlack, 1,
&fragmentShader2Source, NULL);
    glCompileShader(fragmentShaderBlack);
    glShaderSource(fragmentShaderBlue, 1,
&fragmentShader3Source, NULL);
    glCompileShader(fragmentShaderBlue);

    // link the first program object
    glAttachShader(shaderProgramRed, vertexShader);
    glAttachShader(shaderProgramRed, fragmentShaderRed);
    glLinkProgram(shaderProgramRed);
    // then link the second program object using a
```

```
different fragment shader (but same vertex shader)
    // this is perfectly allowed since the inputs and
outputs of both the vertex and fragment shaders are
equally matched.
    glAttachShader(shaderProgramBlack, vertexShader);
    glAttachShader(shaderProgramBlack,
fragmentShaderBlack);
    glLinkProgram(shaderProgramBlack);
    // link the third
    glAttachShader(shaderProgramBlue, vertexShader);
    glAttachShader(shaderProgramBlue,
fragmentShaderBlue);
    glLinkProgram(shaderProgramBlue);

    // set up vertex data (and buffer(s)) and configure
vertex attributes
    // ------------------------------------------------
----------------
    float firstTriangle[] = {
        -0.9f, -0.5f, 0.0f,  // left
        -0.0f, -0.5f, 0.0f,  // right
        -0.45f, 0.5f, 0.0f,  // top
    };
    float secondTriangle[] = {
        0.0f, -0.5f, 0.0f,  // left
        0.9f, -0.5f, 0.0f,  // right
        0.45f, 0.5f, 0.0f   // top
    };
    float thirdTriangle[] = {
        -0.75f, 0.8f, 0.0f,
        -0.5f, 0.8f, 0.0f,
        -0.6f, 1.0f, 0.0f
    };
    unsigned int VBOs[3], VAOs[3];
    glGenVertexArrays(3, VAOs); // we can also generate
multiple VAOs or buffers at the same time
    glGenBuffers(3, VBOs);
    // first triangle setup
    // --------------------
    glBindVertexArray(VAOs[0]);
    glBindBuffer(GL_ARRAY_BUFFER, VBOs[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(firstTriangle),
firstTriangle, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void*)0);   // Vertex attributes stay
the same
```

```
    glEnableVertexAttribArray(0);
    // glBindVertexArray(0); // no need to unbind at all
as we directly bind a different VAO the next few lines
    // second triangle setup
    // --------------------
    glBindVertexArray(VAOs[1]); // note that we bind to
a different VAO now
    glBindBuffer(GL_ARRAY_BUFFER, VBOs[1]);   // and a
different VBO
    glBufferData(GL_ARRAY_BUFFER,
sizeof(secondTriangle), secondTriangle, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
(void*)0); // because the vertex data is tightly packed
we can also specify 0 as the vertex attribute's stride
to let OpenGL figure it out
    glEnableVertexAttribArray(0);
    // glBindVertexArray(0); // not really necessary as
well, but beware of calls that could affect VAOs while
this one is bound (like binding element buffer objects,
or enabling/disabling vertex attributes)

    glBindVertexArray(VAOs[2]); // note that we bind to
a different VAO now
    glBindBuffer(GL_ARRAY_BUFFER, VBOs[2]);   // and a
different VBO
    glBufferData(GL_ARRAY_BUFFER, sizeof(thirdTriangle),
thirdTriangle, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
(void*)0); // because the vertex data is tightly packed
we can also specify 0 as the vertex attribute's stride
to let OpenGL figure it out
    glEnableVertexAttribArray(0);
    // glBindVertexArray(0); // not really necessary as
well, but beware of calls that could affect VAOs while
this one is bound (like binding element buffer objects,
or enabling/disabling vertex attributes)


    // uncomment this call to draw in wireframe
polygons.
    //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
```

```
        // input
        // -----
        processInput(window);

        // render
        // ------
        glClearColor(0.0f, 1.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // now when we draw the triangle we first use
the vertex and orange fragment shader from the first
program
        glUseProgram(shaderProgramRed);
        // draw the first triangle using the data from
our first VAO
        glBindVertexArray(VAOs[0]);
        glDrawArrays(GL_TRIANGLES, 0, 3); // this call
should output an red triangle
        // then we draw the second triangle using the
data from the second VAO
        // when we draw the second triangle we want to
use a different shader program so we switch to the
shader program with our yellow fragment shader.
        glUseProgram(shaderProgramBlack);
        glBindVertexArray(VAOs[1]);
        glDrawArrays(GL_TRIANGLES, 0, 3); // this call
should output a black triangle

        glUseProgram(shaderProgramBlue); // blue
        glBindVertexArray(VAOs[2]);
        glDrawArrays(GL_TRIANGLES, 0, 3); // this call
should output a yellow triangle
        // glfw: swap buffers and poll IO events (keys
pressed/released, mouse moved etc.)
        // -------------------------------------------
-------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de-allocate all resources once they've
outlived their purpose:
    // ------------------------------------------------
----------------------
    glDeleteVertexArrays(2, VAOs);
    glDeleteBuffers(2, VBOs);
```

```
    glDeleteProgram(shaderProgramRed);
    glDeleteProgram(shaderProgramBlack);
    glDeleteProgram(shaderProgramBlue);

    // glfw: terminate, clearing all previously
allocated GLFW resources.
    // ------------------------------------------------
----------------
    glfwTerminate();
    return 0;
}

// process all input: query GLFW whether relevant keys
are pressed/released this frame and react accordingly
// ------------------------------------------------------
-----------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) ==
GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}

// glfw: whenever the window size changed (by OS or user
resize) this callback function executes
// ------------------------------------------------------
---------------------------------------
void framebuffer_size_callback(GLFWwindow* window, int
width, int height)
{
    // make sure the viewport matches the new window
dimensions; note that width and
    // height will be significantly larger than
specified on retina displays.
    glViewport(0, 0, width, height);
}
```

Output: