**Roll No: 1703105**

**Lab Performance Test [1]**

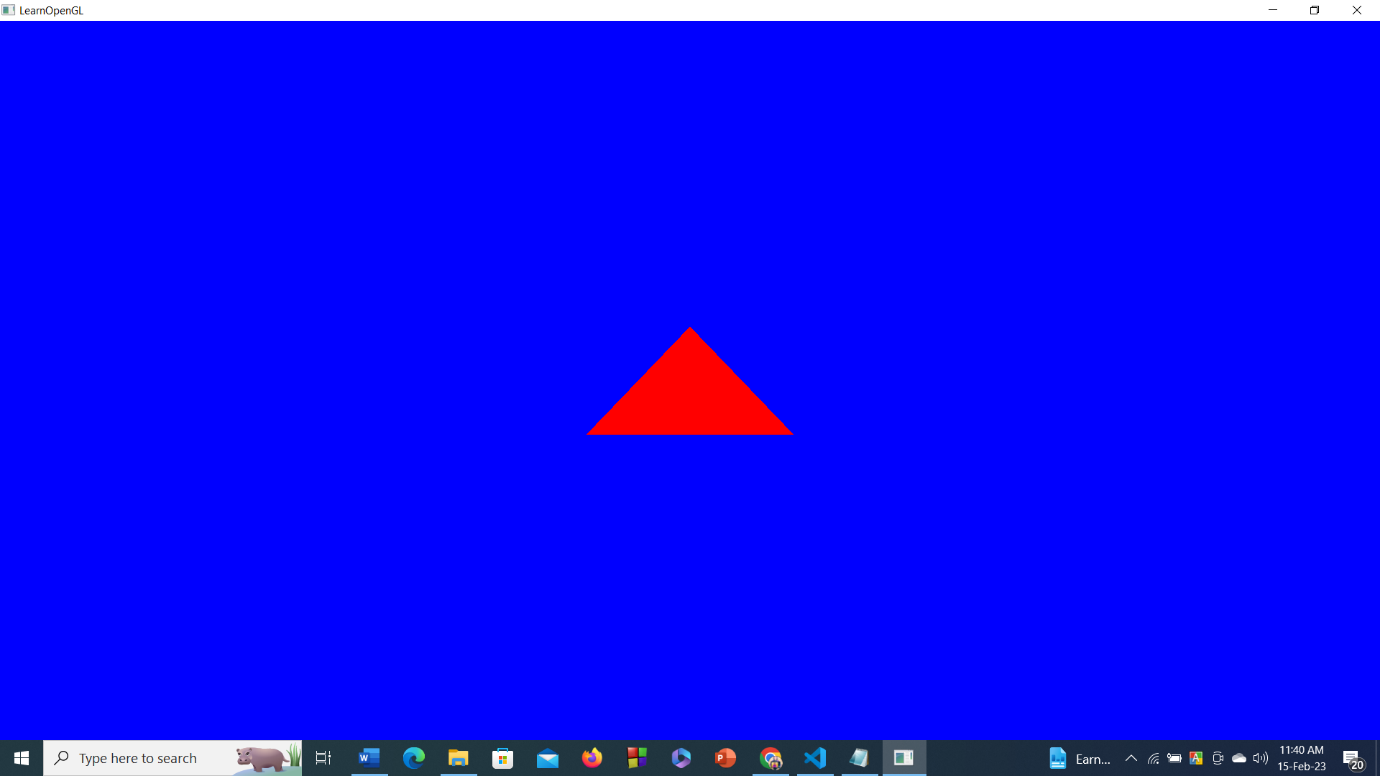
**Lab Task Q[1]**

**Question:** Show an OpenGL Program which will show a triangle with red color where each edge will be of length 3 with background color blue.

**Solution (Bold your own written code):**

|  |
| --- |
| #include <glad.h>  #include <glfw3.h>  #include <iostream>  //roll:1703105  //Show an OpenGL Program which will show a triangle with  //red color where each edge will be of length 3 with background color blue.  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 \*fragmentShaderSource = "#version 330 core\n"      "out vec4 FragColor;\n"      "void main()\n"      "{\n"      "   FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);\n"      //rgb color.color ke 255 dye vag dye      "}\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[] = {**  **-0.0f, 0.15f, 0.0f, // left**  **-0.15f, -0.15f, 0.0f, // right**  **0.15f,  -0.15f, 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);      //Bind function er kaj VAO ke modify krbe      glBindBuffer(GL\_ARRAY\_BUFFER, VBO);      //vbo vao join      glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);      //vbo verticess join      glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);//GL\_false mane +1 to -1 dhore coordinate dhora hoyeche      glEnableVertexAttribArray(0);//0 index theke suru hbe      // 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); //Vbo ke r modify korbo na..      // 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); //Vao ke r modify korbo na      // 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, 0.0f, 1.0f, 1.0f);          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 Performance Test [1]**

**Lab Task Q[2]**

**Question:** Show an OpenGL Program which will show one triangle with green color and length of 0.4 aligned in y axis which will be overlapped with another triangle with blue color and length of 0.2 aligned in z axis with white background using 2 VBO and 2 VAO.

**Solution (Bold your own written code):**

|  |
| --- |
| #include <glad.h>  #include <glfw3.h>  #include <iostream>  void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);  void processInput(GLFWwindow \*window);  //roll:1703105  //Show an OpenGL Program which will show one triangle with green color and length of 0.4  // aligned in y axis which will be overlapped with another triangle with blue  //color and length of 0.2 aligned in z axis with white background using 2 VBO and 2 VAO.  // 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(0.0f, 1.0f, 0.0f, 1.0f);\n"      "}\n\0";  const char \*fragmentShader2Source = "#version 330 core\n"      "out vec4 FragColor;\n"      "void main()\n"      "{\n"      "   FragColor = vec4(0.0f, 0.0f, 1.0f, 1.0f);\n"      "}\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 fragmentShaderOrange = glCreateShader(GL\_FRAGMENT\_SHADER); // the first fragment shader that outputs the color orange      unsigned int fragmentShaderYellow = glCreateShader(GL\_FRAGMENT\_SHADER); // the second fragment shader that outputs the color yellow      unsigned int shaderProgramOrange = glCreateProgram();      unsigned int shaderProgramYellow = glCreateProgram(); // the second shader program      glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);      glCompileShader(vertexShader);      glShaderSource(fragmentShaderOrange, 1, &fragmentShader1Source, NULL);      glCompileShader(fragmentShaderOrange);      glShaderSource(fragmentShaderYellow, 1, &fragmentShader2Source, NULL);      glCompileShader(fragmentShaderYellow);      // link the first program object      glAttachShader(shaderProgramOrange, vertexShader);      glAttachShader(shaderProgramOrange, fragmentShaderOrange);      glLinkProgram(shaderProgramOrange);      // 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(shaderProgramYellow, vertexShader);      glAttachShader(shaderProgramYellow, fragmentShaderYellow);      glLinkProgram(shaderProgramYellow);      // set up vertex data (and buffer(s)) and configure vertex attributes      // ------------------------------------------------------------------      float firstTriangle[] = {          0.0f, 0.5f, 0.0f,  // left          0.3f, 0.4f, 0.0f,  // right          0.0f, 0.1f, 0.0f,  // top      };      float secondTriangle[] = {           0.015f, 0.015f,0.0f,  // left           0.1f, 0.2f,0.0f,  // right           0.5f, 0.1f ,0.0f, // top      };      unsigned int VBOs[2], VAOs[2];      glGenVertexArrays(2, VAOs); // we can also generate multiple VAOs or buffers at the same time      glGenBuffers(2, 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)      // 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(1.0f, 1.0f, 1.0f, 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(shaderProgramOrange);          // draw the first triangle using the data from our first VAO          glBindVertexArray(VAOs[0]);          glDrawArrays(GL\_TRIANGLES, 0, 3);   // this call should output an orange 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(shaderProgramYellow);          glBindVertexArray(VAOs[1]);          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(shaderProgramOrange);      glDeleteProgram(shaderProgramYellow);      // 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:** 