Roll No: 1703018

Lab Performance Test 1

Lab Task Q1

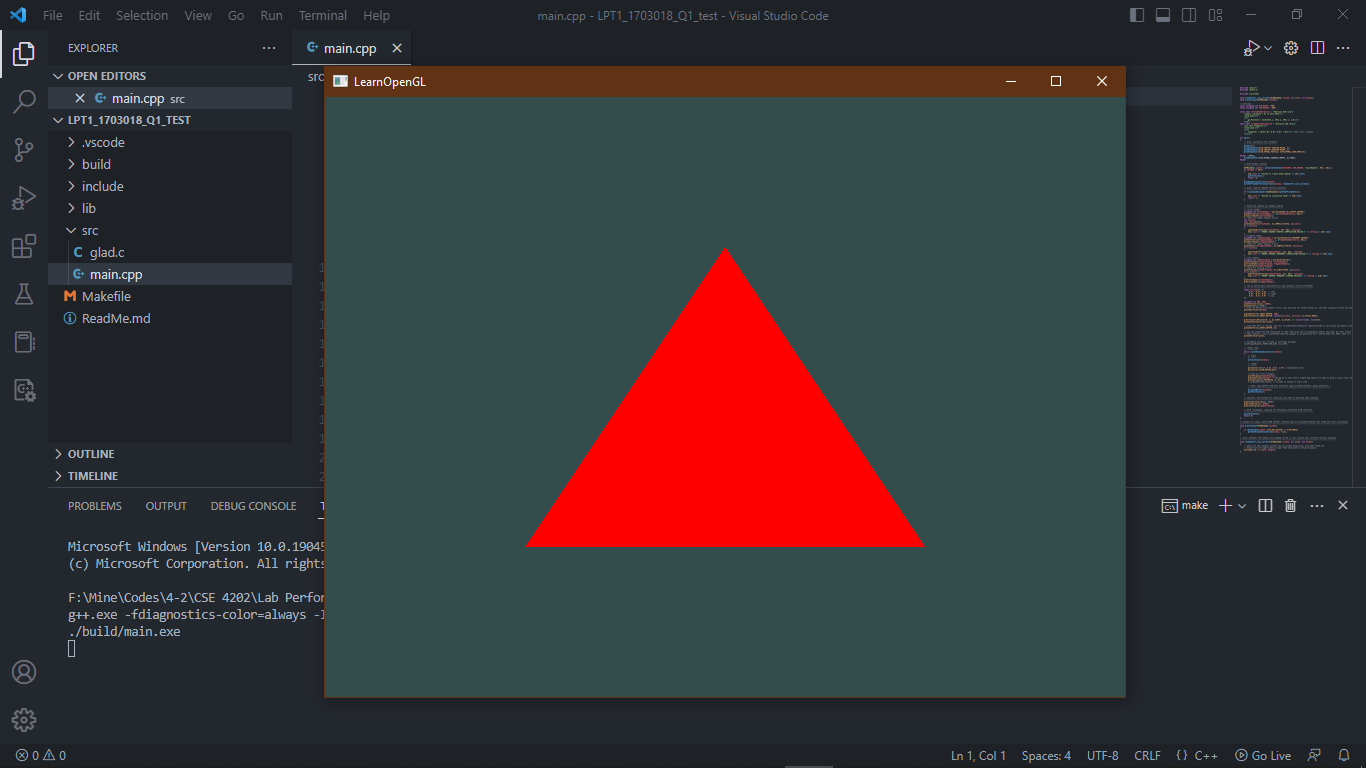
Question:

Show an OpenGL Program which will show a red triangle.

Solution:

|  |
| --- |
| **#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 \*fragmentShaderSource = "#version 330 core\n"  "out vec4 FragColor;\n"  "void main()\n"  "{\n"  " **FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);\n**" //Red color 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  // ------------------------------------  // 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, 0.5f, 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**  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 black, green and blue color in red background.

Solution:

|  |
| --- |
| **#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(0.0f, 0.0f, 1.0f, 1.0f);\n" //blue color**  **"}\n\0";**  **const char \*fragmentShader2Source = "#version 330 core\n"**  **"out vec4 FragColor;\n"**  **"void main()\n"**  **"{\n"**  **" FragColor = vec4(0.0f, 1.0f, 0.0f, 1.0f);\n" //green color**  **"}\n\0";**  **const char \*fragmentShader3Source = "#version 330 core\n"**  **"out vec4 FragColor;\n"**  **"void main()\n"**  **"{\n"**  **" FragColor = vec4(0.0f, 0.0f, 0.0f, 1.0f);\n" //black color**  **"}\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 fragmentShaderBlue = glCreateShader(GL\_FRAGMENT\_SHADER); // the first fragment shader that outputs the color Blue**  **unsigned int fragmentShaderGreen = glCreateShader(GL\_FRAGMENT\_SHADER); // the second fragment shader that outputs the color Green**  **unsigned int fragmentShaderBlack = glCreateShader(GL\_FRAGMENT\_SHADER); // the third fragment shader that outputs the color Black**  **unsigned int shaderProgramBlue = glCreateProgram();**  **unsigned int shaderProgramGreen = glCreateProgram(); // the second shader program**  **unsigned int shaderProgramBlack = glCreateProgram(); // the third shader program**  **glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);**  **glCompileShader(vertexShader);**  **glShaderSource(fragmentShaderBlue, 1, &fragmentShader1Source, NULL);**  **glCompileShader(fragmentShaderBlue);**  **glShaderSource(fragmentShaderGreen, 1, &fragmentShader2Source, NULL);**  **glCompileShader(fragmentShaderGreen);**  **glShaderSource(fragmentShaderBlack, 1, &fragmentShader3Source, NULL);**  **glCompileShader(fragmentShaderBlack);**  // link the first program object  **glAttachShader(shaderProgramBlue, vertexShader);**  **glAttachShader(shaderProgramBlue, fragmentShaderBlue);**  **glLinkProgram(shaderProgramBlue);**  // 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(shaderProgramGreen, vertexShader);**  **glAttachShader(shaderProgramGreen, fragmentShaderGreen);**  **glLinkProgram(shaderProgramGreen);**  // link the third program object  **glAttachShader(shaderProgramBlack, vertexShader);**  **glAttachShader(shaderProgramBlack, fragmentShaderBlack);**  **glLinkProgram(shaderProgramBlack);**  // set up vertex data (and buffer(s)) and configure vertex attributes  // ------------------------------------------------------------------  **float firstTriangle[] = {**  **-0.9f, -0.5f, 0.0f, // left**  **-0.5f, -0.5f, 0.0f, // right**  **-0.7f, 0.5f, 0.0f, // top**  **};**  **float secondTriangle[] = {**  **-0.4f, -0.5f, 0.0f, // left**  **0.2f, -0.5f, 0.0f, // right**  **-0.1f, 0.5f, 0.0f // top**  **};**  **float thirdTriangle[] = {**  **0.5f, -0.5f, 0.0f, // left**  **0.9f, -0.5f, 0.0f, // right**  **0.7f, 0.5f, 0.0f // top**  **};**  **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)**  **// third triangle setup**  **// ---------------------**  **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(1.0f, 0.0f, 0.0f, 1.0f); //red background**  glClear(GL\_COLOR\_BUFFER\_BIT);  // now when we draw the triangle we first use the vertex and Blue fragment shader from the first program  **glUseProgram(shaderProgramBlue);**  **// draw the first triangle using the data from our first VAO**  **glBindVertexArray(VAOs[0]);**  **glDrawArrays(GL\_TRIANGLES, 0, 3); // this call should output an Blue 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 Green fragment shader.**  **glUseProgram(shaderProgramGreen);**  **glBindVertexArray(VAOs[1]);**  **glDrawArrays(GL\_TRIANGLES, 0, 3); // this call should output a Green triangle**  **glUseProgram(shaderProgramBlack);**  **glBindVertexArray(VAOs[2]);**  **glDrawArrays(GL\_TRIANGLES, 0, 3); // this call should output a Black 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(3, VAOs);**  **glDeleteBuffers(3, VBOs);**  **glDeleteProgram(shaderProgramBlue);**  **glDeleteProgram(shaderProgramGreen);**  **glDeleteProgram(shaderProgramBlack);**  // 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:

