**Roll No: 1703016**

**Lab Performance Test 2**

**Lab Task Q1**

**Question: Show an OpenGL program which will show a triangle whose triangle's color will alternate between a black color and blue color after some time.**

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

|  |
| --- |
| // Show an OpenGL program which will show a triangle whose triangle's color will alternate between a black color and blue color after some time.  // Roll: 1703016  #include "glad.h"  #include "glfw3.h"  #include "shader\_s.h"  #include<math.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 + 0.0, aPos.y, aPos.z, 1.0);\n"  "}\0";  const char \*fragmentShaderSource = "#version 330 core\n"  "out vec4 FragColor;\n"  "uniform vec4 ourColor;\n"  "void main()\n"  "{\n"  " FragColor = ourColor;\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  // ------------------------------------  // 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, // bottom right  -0.5f, -0.5f, 0.0f, // bottom left  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);  // 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);  // bind the VAO (it was already bound, but just to demonstrate): seeing as we only have a single VAO we can  // just bind it beforehand before rendering the respective triangle; this is another approach.  glBindVertexArray(VAO);  // render loop  // -----------  while (!glfwWindowShouldClose(window))  {  // input  // -----  processInput(window);  // render  // ------  glClearColor(0.2f, 0.3f, 0.3f, 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  // be sure to activate the shader before any calls to glUniform  glUseProgram(shaderProgram);  **// update shader uniform**  **double timeValue = glfwGetTime();**  **float blueValue = static\_cast<float>(sin(timeValue) / 2.0 + 0.5);**  **int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");**  **glUniform4f(vertexColorLocation, 0.0f, 0.0f, blueValue, 1.0f);**  // render the triangle  glDrawArrays(GL\_TRIANGLES, 0, 3);  // 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:**

****

****

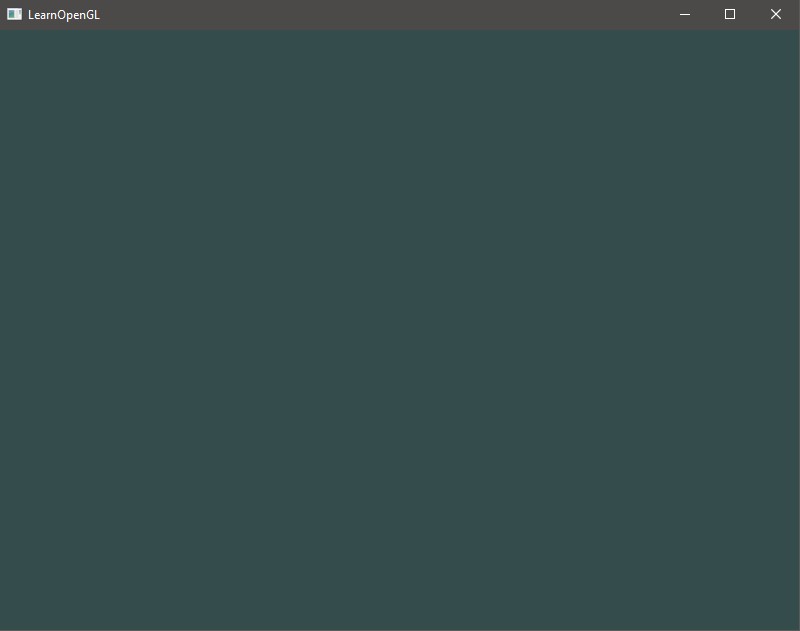
**Lab Task Q2**

**Question: Show an OpenGL program which will show a 3d pyramid at location (3,10,7) which scaled by 2 and is rotated by 90 degree counter-clockwise.**

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

|  |
| --- |
| // Show an OpenGL program which will show a 3d pyramid at location (3,10,7) which scaled by 2 and is rotated by 90 degree counter-clockwise.  // Roll: 1703016  #include "glad.h"  #include "glfw3.h"  #include "glm/glm.hpp"  #include "glm/gtc/matrix\_transform.hpp"  #include "glm/gtc/type\_ptr.hpp"  #define STB\_IMAGE\_IMPLEMENTATION  #include "stb\_image.h"  #include "shader\_m.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;  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;  }  // configure global opengl state  // -----------------------------  glEnable(GL\_DEPTH\_TEST);  // build and compile our shader zprogram  // ------------------------------------  Shader ourShader("src\\6.2.coordinate\_systems.vs", "src\\6.2.coordinate\_systems.fs");  // set up vertex data (and buffer(s)) and configure vertex attributes  // ------------------------------------------------------------------  float vertices[] = {  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  0.5f, -0.5f, -0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f  };  unsigned int VBO, VAO;  glGenVertexArrays(1, &VAO);  glGenBuffers(1, &VBO);  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  // position attribute  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0);  // texture coord attribute  glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void\*)(3 \* sizeof(float)));  glEnableVertexAttribArray(1);  // load and create a texture  // -------------------------  unsigned int texture1, texture2;  // texture 1  // ---------  glGenTextures(1, &texture1);  glBindTexture(GL\_TEXTURE\_2D, texture1);  // set the texture wrapping parameters  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  // set texture filtering parameters  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  // load image, create texture and generate mipmaps  int width, height, nrChannels;  stbi\_set\_flip\_vertically\_on\_load(true); // tell stb\_image.h to flip loaded texture's on the y-axis.  unsigned char \*data = stbi\_load("src\\container.jpg", &width, &height, &nrChannels, 0);  if (data)  {  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);  glGenerateMipmap(GL\_TEXTURE\_2D);  }  else  {  std::cout << "Failed to load texture" << std::endl;  }  stbi\_image\_free(data);  // texture 2  // ---------  glGenTextures(1, &texture2);  glBindTexture(GL\_TEXTURE\_2D, texture2);  // set the texture wrapping parameters  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  // set texture filtering parameters  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  // load image, create texture and generate mipmaps  data = stbi\_load("src\\resources\\textures\\awesomeface.png", &width, &height, &nrChannels, 0);  if (data)  {  // note that the awesomeface.png has transparency and thus an alpha channel, so make sure to tell OpenGL the data type is of GL\_RGBA  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, data);  glGenerateMipmap(GL\_TEXTURE\_2D);  }  else  {  std::cout << "Failed to load texture awesomeface" << std::endl;  }  stbi\_image\_free(data);  // tell opengl for each sampler to which texture unit it belongs to (only has to be done once)  // -------------------------------------------------------------------------------------------  ourShader.use();  ourShader.setInt("texture1", 0);  ourShader.setInt("texture2", 1);  // render loop  // -----------  while (!glfwWindowShouldClose(window))  {  // input  // -----  processInput(window);  // render  // ------  glClearColor(0.2f, 0.3f, 0.3f, 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // also clear the depth buffer now!  // bind textures on corresponding texture units  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D, texture1);  glActiveTexture(GL\_TEXTURE1);  glBindTexture(GL\_TEXTURE\_2D, texture2);  // activate shader  ourShader.use();  // create transformations  glm::mat4 model = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first  glm::mat4 view = glm::mat4(1.0f);  glm::mat4 projection = glm::mat4(1.0f);  **model = glm::translate(model, glm::vec3(3.0f,10.0f,7.0f)); // here point update**  **model = glm::rotate(model, (float)glm::radians(-90.0f), glm::vec3(0.0f, 0.0f, 1.0f)); // here angle update**  **model = glm::scale(model, glm::vec3(2.0, 2.0, 2.0)); // here scaling update**  projection = glm::perspective(glm::radians(45.0f), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);  // retrieve the matrix uniform locations  unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");  unsigned int viewLoc = glGetUniformLocation(ourShader.ID, "view");  // pass them to the shaders (3 different ways)  glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));  glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, &view[0][0]);  // note: currently we set the projection matrix each frame, but since the projection matrix rarely changes it's often best practice to set it outside the main loop only once.  ourShader.setMat4("projection", projection);  // render box  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  // 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);  // 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:**

****