**Roll No: 1703105**

**Lab Performance Test [2]**

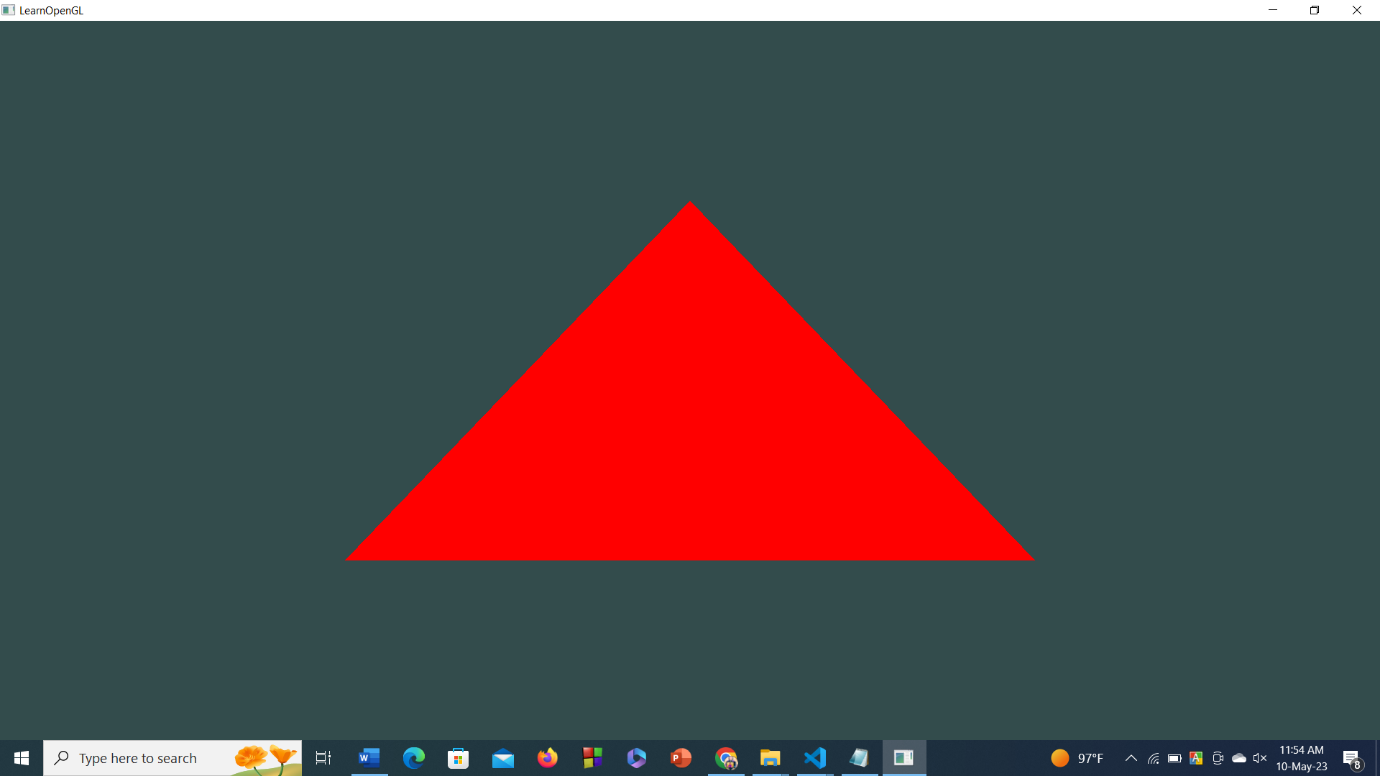
**Lab Task Q[1]**

**Question:** Show an OpenGL program with embedded vertex and fragment shader which will show one red colored 2D triangle where red color will be sent via uniform variable.

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

|  |
| --- |
| #include <glad.h>  #include <glfw3.h>  #include "shader\_s.h"  #include <iostream>  void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);  void processInput(GLFWwindow \*window);  // settings  // 1703105  const unsigned int SCR\_WIDTH = 800;  const unsigned int SCR\_HEIGHT = 600;  //float offset = 0.5f;  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  // ------------------------------------  Shader ourShader("src/shader.vs", "src/shader.fs"); // you can name your shader files however you like  // set up vertex data (and buffer(s)) and configure vertex attributes  // ------------------------------------------------------------------  float vertices[] = {  // positions // colors  0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom right  -0.5f, -0.5f, 0.0f, 1.0f, .0f, 0.0f, // bottom left  0.0f, 0.5f, 0.0f, 1.0f, 0.0f, 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);  // position attribute  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0);  // color attribute  glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)(3 \* sizeof(float)));  glEnableVertexAttribArray(1);  // 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);  // render loop  // -----------  while (!glfwWindowShouldClose(window))  {  // input  // -----  processInput(window);  // render  // ------  glClearColor(0.2f, 0.3f, 0.3f, 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  // render the triangle  //ourShader.setFloat("xOffset", offset);  ourShader.use();  glBindVertexArray(VAO);  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);  // 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);  }  Fragment shader:  #version 330 core  out vec4 FragColor;  in vec3 ourColor;  void main()  {      FragColor = vec4(ourColor, 1.0f);  }  Vertex shader  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aColor;  out vec3 ourColor;  uniform float xOffset;  void main()  {  gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0); // add the xOffset to the x position of the vertex position  ourColor = aColor;  } |

**Output:**



**Roll No: 1703105**

**Lab Performance Test [2]**

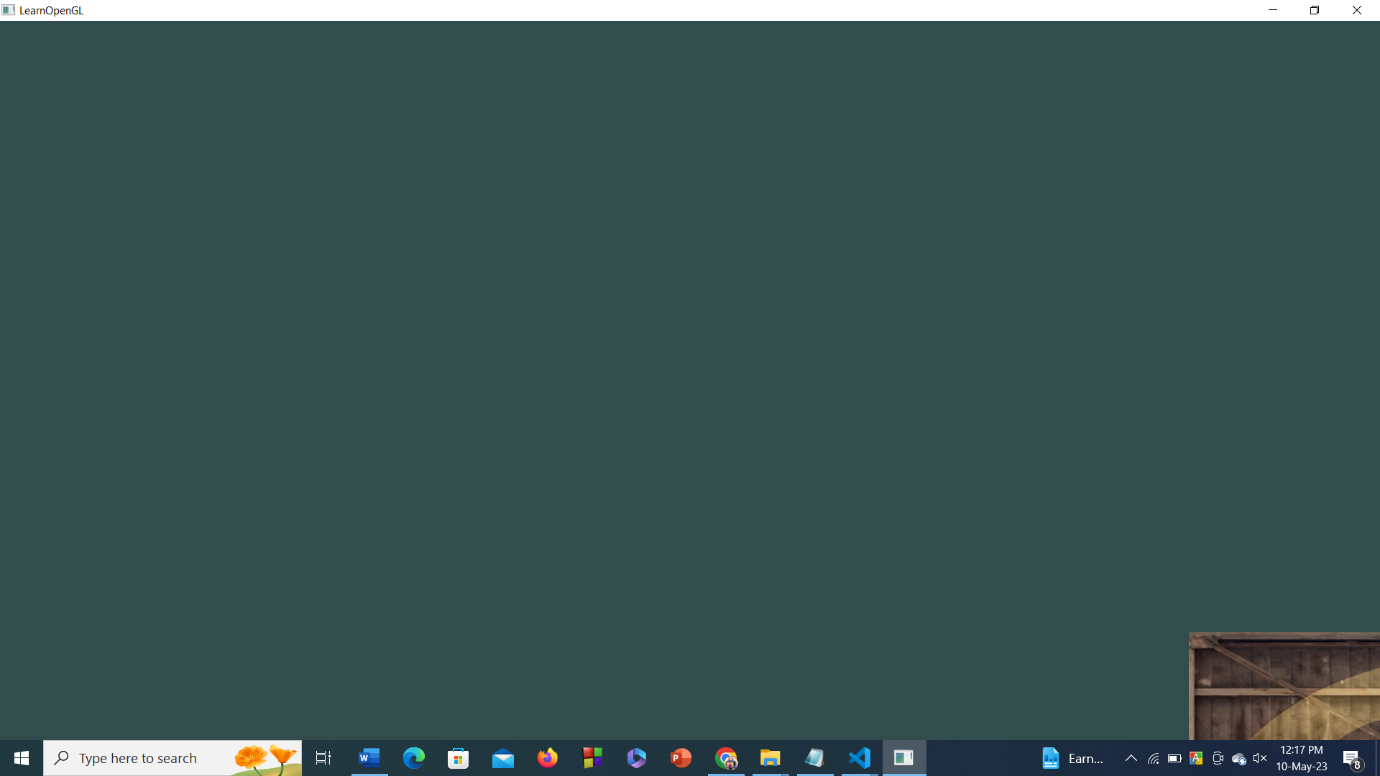
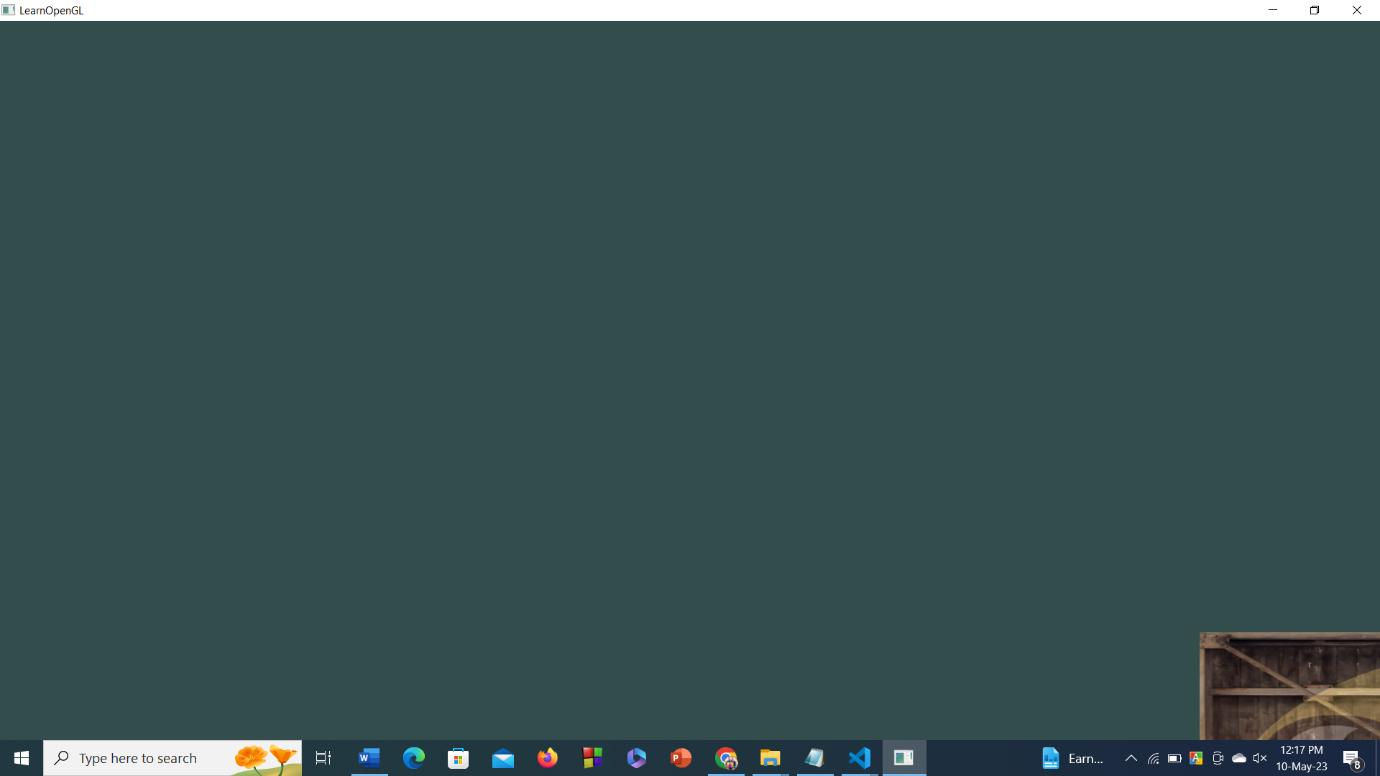
**Lab Task Q[2]**

**Question:** Show an OpenGL program which will show a 3d cube which will change its size 1.2 times and change its location +1 in one axis from time to time.

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

|  |
| --- |
| #include <glad.h>  #include <glfw3.h>  #define STB\_IMAGE\_IMPLEMENTATION  #include <stb\_image.h>  #include <glm/glm.hpp>  #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtc/type\_ptr.hpp>  //#include <learnopengl/filesystem.h>  #include <shader\_s.h>  //1703105  #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;      }      // build and compile our shader zprogram      // ------------------------------------      Shader ourShader("src/transform.vs", "src/transform.fs");      // set up vertex data (and buffer(s)) and configure vertex attributes      // ------------------------------------------------------------------      float vertices[] = {          // positions          // texture coords           0.5f,  0.5f, 0.0f,   1.0f, 1.0f, // top right           0.5f, -0.5f, 0.0f,   1.0f, 0.0f, // bottom right          -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, // bottom left          -0.5f,  0.5f, 0.0f,   0.0f, 1.0f  // top left      };      unsigned int indices[] = {          0, 1, 3, // first triangle          1, 2, 3  // second triangle      };      unsigned int VBO, VAO, EBO;      glGenVertexArrays(1, &VAO);      glGenBuffers(1, &VBO);      glGenBuffers(1, &EBO);      glBindVertexArray(VAO);      glBindBuffer(GL\_ARRAY\_BUFFER, VBO);      glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);      glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);      glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, 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("C:/Users/User/Downloads/labtest2/transfromation/trans-5.1/texture/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("C:/Users/User/Downloads/labtest2/transfromation/trans-5.1/texture/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" << 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);          // bind textures on corresponding texture units          glActiveTexture(GL\_TEXTURE0);          glBindTexture(GL\_TEXTURE\_2D, texture1);          glActiveTexture(GL\_TEXTURE1);          glBindTexture(GL\_TEXTURE\_2D, texture2);          // create transformations          glm::mat4 transform = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first          transform = glm::translate(transform, glm::vec3(1.2f, -1.2f, 0.0f));          transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));          // get matrix's uniform location and set matrix          ourShader.use();          unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");          glUniformMatrix4fv(transformLoc, 1, GL\_FALSE, glm::value\_ptr(transform));          // render container          glBindVertexArray(VAO);          glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);          // 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);      glDeleteBuffers(1, &EBO);      // 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);  }  Fragshader:  #version 330 core  out vec4 FragColor;  in vec2 TexCoord;  // texture samplers  uniform sampler2D texture1;  uniform sampler2D texture2;  void main()  {      // linearly interpolate between both textures (80% container, 20% awesomeface)      FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);  }  Vertexshader:  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec2 aTexCoord;  out vec2 TexCoord;  uniform mat4 transform;  void main()  {  gl\_Position = transform \* vec4(aPos, 1.0);  TexCoord = vec2(aTexCoord.x, aTexCoord.y);  } |

**Output:**



**Roll No: 1703105**

**Lab Performance Test [2]**

**Lab Task Q[3]**

**Question:** Show an OpenGL Program which will show four different overlapping 2d squares with four different colors and are always rotating and changing locations at different rates.

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

|  |
| --- |
| #include <glad.h>  #include <glfw3.h>  #define STB\_IMAGE\_IMPLEMENTATION  #include <stb\_image.h>  //1703015  #include <glm/glm.hpp>  #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtc/type\_ptr.hpp>  #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/texture7.vs", "src/transform6.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      };      // world space positions of our cubes      glm::vec3 cubePositions[] = {          glm::vec3( 0.0f,  0.0f,  0.0f),          glm::vec3( 2.0f,  5.0f, -15.0f),          glm::vec3(-1.5f, -2.2f, -2.5f),          glm::vec3(-3.8f, -2.0f, -12.3f),          glm::vec3( 2.4f, -0.4f, -3.5f),          glm::vec3(-1.7f,  3.0f, -7.5f),          glm::vec3( 1.3f, -2.0f, -2.5f),          glm::vec3( 1.5f,  2.0f, -2.5f),          glm::vec3( 1.5f,  0.2f, -1.5f),          glm::vec3(-1.3f,  1.0f, -1.5f)      };      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("C:/Users/User/Downloads/labtest2/transfromation/trans-5.1/texture/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("C:/Users/User/Downloads/labtest2/transfromation/trans-5.1/texture/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" << 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 view          = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first          glm::mat4 projection    = glm::mat4(1.0f);          projection = glm::perspective(glm::radians(45.0f), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);          view       = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));          // pass transformation matrices to the shader          ourShader.setMat4("projection", projection); // 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("view", view);          // render boxes          glBindVertexArray(VAO);          for (unsigned int i = 0; i < 4; i++)          {              // calculate the model matrix for each object and pass it to shader before drawing              glm::mat4 model = glm::mat4(1.0f);              model = glm::translate(model, cubePositions[i]);              float angle = 20.0f \* i;                  // every 3rd iteration (including the first) we set the angle using GLFW's time function.              angle = glfwGetTime() \* 25.0f;              model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));              ourShader.setMat4("model", model);              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);  }  vertex:  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec2 aTexCoord;  out vec2 TexCoord;  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main()  {  gl\_Position = projection \* view \* model \* vec4(aPos, 1.0f);  TexCoord = vec2(aTexCoord.x, 1.0 - aTexCoord.y);  }  Fragment:  #version 330 core  out vec4 FragColor;  in vec2 TexCoord;  // texture samplers  uniform sampler2D texture1;  uniform sampler2D texture2;  void main()  {      // linearly interpolate between both textures (80% container, 20% awesomeface)      FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);  } |

**Output:**

