**Roll No: 1703105**

**Lab Final**

**Lab Task Q[1]**

**Question: Q1.**  Show an OpenGL program which will show:

**a) Hello Triangle/Shapes:** Two 2D Triangle

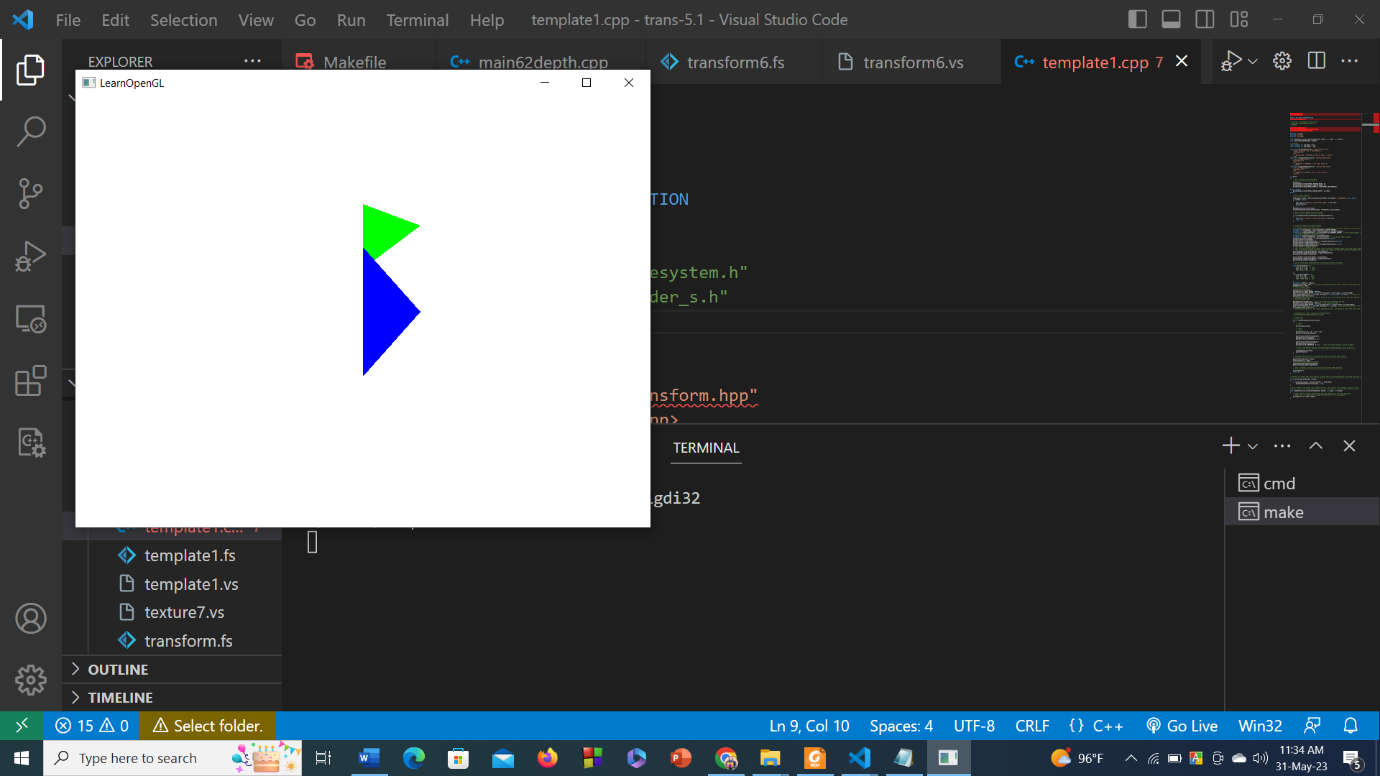
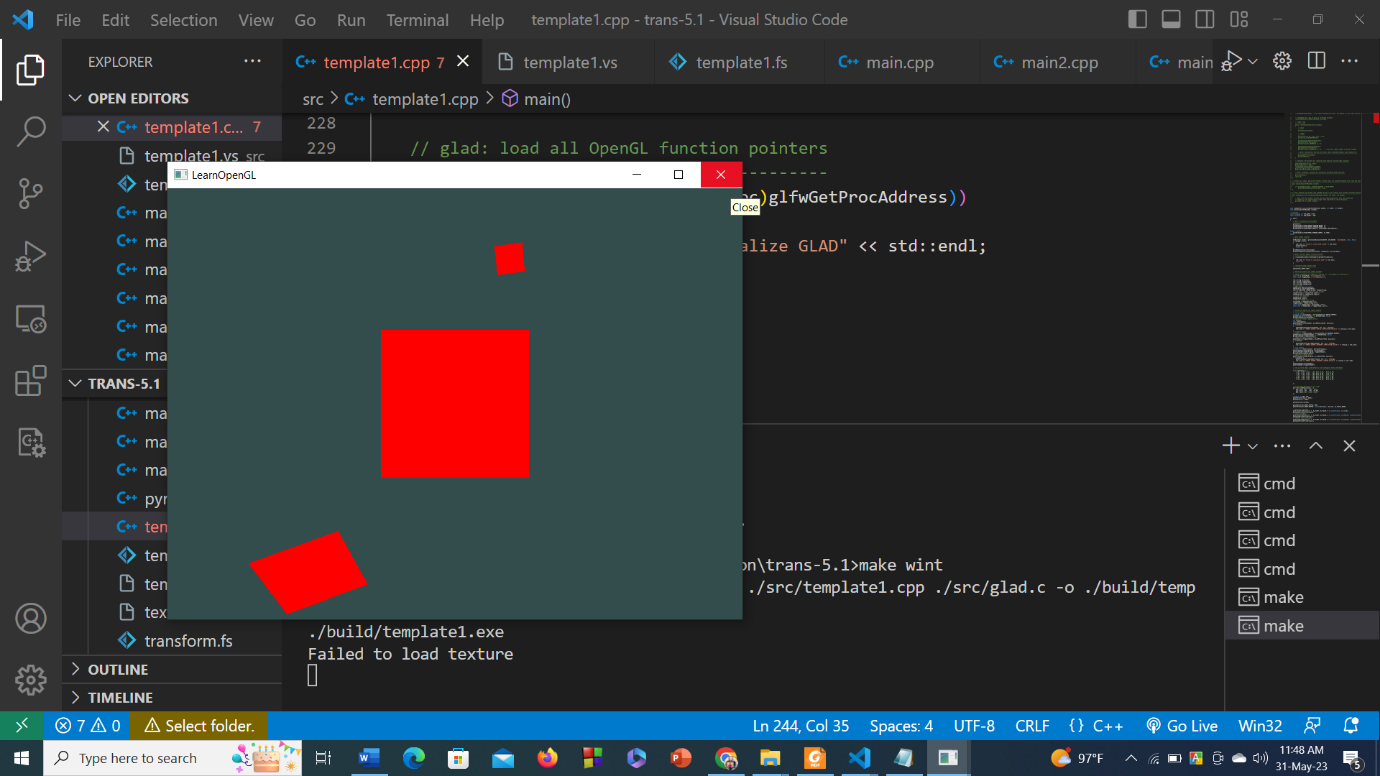
**b) Shader/Texture:** Two different colors for each (Sent via vertices).

**c) Transformations and Coordinate System:** Their size will change using keyboard.

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

|  |
| --- |
| #include "glad.h"  #include "glfw3.h"  #define STB\_IMAGE\_IMPLEMENTATION  #include "stb\_image.h"  // #include "learnopengl/filesystem.h"  // #include "learnopengl/shader\_s.h"  //1703105  #include "glm/glm.hpp"  #include "glm/gtc/matrix\_transform.hpp"  #include <glm/gtc/type\_ptr.hpp>  #include <sstream>  #include <fstream>  #include <iostream>  #include <cmath>  //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 fragmentShaderGreen = glCreateShader(GL\_FRAGMENT\_SHADER);  //     unsigned int fragmentShaderBlue = glCreateShader(GL\_FRAGMENT\_SHADER); // the second fragment shader that outputs the color yellow  //     // int shaderProgramRed = glCreateProgram();  //     unsigned int shaderProgramGreen = glCreateProgram();  //     unsigned int shaderProgramBlue = glCreateProgram(); // the second shader program  //     glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);  //     glCompileShader(vertexShader);  //     glShaderSource(fragmentShaderGreen, 1, &fragmentShader1Source, NULL);  //     glCompileShader(fragmentShaderGreen);  //     glShaderSource(fragmentShaderBlue, 1, &fragmentShader2Source, NULL);  //     glCompileShader(fragmentShaderBlue);  //     // link the first program object  //     // 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);  //     glAttachShader(shaderProgramBlue, vertexShader);  //     glAttachShader(shaderProgramBlue, fragmentShaderBlue);  //     glLinkProgram(shaderProgramBlue);  //     // set up vertex data (and buffer(s)) and configure vertex attributes  //     // ------------------------------------------------------------------  //     float firstTriangle[] = {  //         0.0f, 0.5f, 0.0f,  // left  //         0.2f, 0.4f, 0.0f,  // right  //         0.0f, 0.2f, 0.0f,  // top    //     };  //     float secondTriangle[] = {  //         0.0f, 0.3f,0.0f,  // left  //         0.2f, 0.0f,0.0f,  // right  //         0.0f, -0.3f ,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);  //         glUseProgram(shaderProgramGreen);  //         glBindVertexArray(VAOs[0]);  //         glDrawArrays(GL\_TRIANGLES, 0, 3);    //         glUseProgram(shaderProgramBlue);  //         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(shaderProgramRed);  //     glDeleteProgram(shaderProgramGreen);  //     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);  // }    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/shader/4.1.texture.vs", "src/shader/4.1.texture.fs");      const char\* vertexPath = "src/template1.vs";      const char\* fragmentPath = "src/template1.fs";      std::string vertexCode;      std::string fragmentCode;      std::ifstream vShaderFile;      std::ifstream fShaderFile;      // open files      vShaderFile.open(vertexPath);      fShaderFile.open(fragmentPath);      std::stringstream vShaderStream, fShaderStream;      // read file's buffer contents into streams      vShaderStream << vShaderFile.rdbuf();      fShaderStream << fShaderFile.rdbuf();      // close file handlers      vShaderFile.close();      fShaderFile.close();      // convert stream into string      vertexCode = vShaderStream.str();      fragmentCode = fShaderStream.str();      const char\* vShaderCode = vertexCode.c\_str();      const char \* fShaderCode = fragmentCode.c\_str();      // build and compile our shader program      // ------------------------------------      // vertex shader      unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);      glShaderSource(vertexShader, 1, &vShaderCode, 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, &fShaderCode, 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, -0.5f,  1.0f, 0.0f, 0.0f,   0.0f, 0.0f,           0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,   1.0f, 0.0f,           0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,   1.0f, 1.0f,           0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,   1.0f, 1.0f,          -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,   0.0f, 1.0f,          -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,   0.0f, 0.0f,        };      // world space positions of our cubes      glm::vec3 cubePositions[] = {          glm::vec3( 0.0f,  0.0f,  0.0f),        };      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, 8 \* sizeof(float), (void\*)0);      glEnableVertexAttribArray(0);      // color attribute      glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(3 \* sizeof(float)));      glEnableVertexAttribArray(1);      // texture coord attribute      glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(6 \* sizeof(float)));      glEnableVertexAttribArray(2);      // 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("I:/8th semester/Graphics/labtest2//transfromation/trans-5.1/texturecontainer.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("I:/8th semester/Graphics/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);      glUseProgram(shaderProgram);      glUniform1i(glGetUniformLocation(shaderProgram, "texture1"), 0);      glUniform1i(glGetUniformLocation(shaderProgram, "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          glUseProgram(shaderProgram);          // create transformations          glm::mat4 view          = glm::mat4(1.0f);          glm::mat4 projection    = glm::mat4(1.0f);          view  = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));          projection = glm::perspective(glm::radians(45.0f), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);          glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"      ), 1, GL\_FALSE, &view[0][0]);          glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL\_FALSE, &projection[0][0]);          // render container          glBindVertexArray(VAO);          for (unsigned int i = 0; i < 3; i++)          {              // calculate the model matrix for each object and pass it to shader before drawing              glm::mat4 model = glm::mat4(1.0f);              model = glm::scale(model, glm::vec3(1.0f));              model = glm::translate(model, cubePositions[i]);              float angle = 20.0f \* i;              model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));              glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL\_FALSE, &model[0][0]);              glDrawArrays(GL\_TRIANGLES, 0, 6);          }          // 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)  {      //Keyboard Example, F KEY = GLFW\_KEY\_F      //Keyboard Example, 1 KEY = GLFW\_KEY\_1      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);  }  #version 330 core  out vec4 FragColor;  in vec3 ourColor;  in vec2 TexCoord;  // texture samplers  uniform sampler2D texture1;  uniform sampler2D texture2;  void main()  {      FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);      //FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);  }  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aColor;  layout (location = 2) in vec2 aTexCoord;  out vec3 ourColor;  out vec2 TexCoord;  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main()  {    //gl\_Position = projection \* view \* model \* vec4(aPos, 1.0f);  gl\_Position = projection \* view \* model \* vec4(aPos, 1.0f);  ourColor = aColor;  TexCoord = vec2(aTexCoord.x, aTexCoord.y);  } |

**Output:**



**Roll No: 1703105**

**Lab Final**

**Lab Task Q[2]**

**Question:** Show an OpenGL program which will show a **less shinny** 3d colored cube which will be lighted by another 3d white colored cube where:

**a) Camera:**Camera will change its zoom from time to time.

**b) Lighting:** 20% ambient lighting + 20% specular lighting.

**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\_m.h>  #include <camera.h>  //1703105  #include <iostream>  void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);  void mouse\_callback(GLFWwindow\* window, double xpos, double ypos);  void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset);  void processInput(GLFWwindow \*window);  // settings  const unsigned int SCR\_WIDTH = 800;  const unsigned int SCR\_HEIGHT = 600;  // camera  Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));  float lastX = SCR\_WIDTH / 2.0f;  float lastY = SCR\_HEIGHT / 2.0f;  bool firstMouse = true;  // timing  float deltaTime = 0.0f;  float lastFrame = 0.0f;  // lighting  glm::vec3 lightPos(1.2f, 1.0f, 2.0f);  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);      glfwSetCursorPosCallback(window, mouse\_callback);      glfwSetScrollCallback(window, scroll\_callback);      // tell GLFW to capture our mouse      glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);      // 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 lightingShader("src/lighting.vs", "src/specular.fs");      //Shader lightingShader("src/s\_view.vs", "src/s\_view.fs");      //Shader lightingShader("src/gouraud.vs", "src/gauraud.fs");      Shader lightCubeShader("src/light\_cube.vs", "src/light\_cube.fs");      // set up vertex data (and buffer(s)) and configure vertex attributes      // ------------------------------------------------------------------      float vertices[] = {          -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,           0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,           0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,           0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,          -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,          -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,          -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,           0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,           0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,           0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,          -0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,          -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,          -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,          -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,          -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,          -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,          -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,          -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,           0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,           0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,           0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,           0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,           0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,           0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,          -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,           0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,           0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,           0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,          -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,          -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,          -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,           0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,           0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,           0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,          -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,          -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f      };      // first, configure the cube's VAO (and VBO)      unsigned int VBO, cubeVAO;      glGenVertexArrays(1, &cubeVAO);      glGenBuffers(1, &VBO);      glBindBuffer(GL\_ARRAY\_BUFFER, VBO);      glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);      glBindVertexArray(cubeVAO);      // position attribute      glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);      glEnableVertexAttribArray(0);      // normal attribute      glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)(3 \* sizeof(float)));      glEnableVertexAttribArray(1);      // second, configure the light's VAO (VBO stays the same; the vertices are the same for the light object which is also a 3D cube)      unsigned int lightCubeVAO;      glGenVertexArrays(1, &lightCubeVAO);      glBindVertexArray(lightCubeVAO);      glBindBuffer(GL\_ARRAY\_BUFFER, VBO);      // note that we update the lamp's position attribute's stride to reflect the updated buffer data      glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);      glEnableVertexAttribArray(0);      // render loop      // -----------      while (!glfwWindowShouldClose(window))      {          // per-frame time logic          // --------------------          float currentFrame = static\_cast<float>(glfwGetTime());          //deltaTime = currentFrame - lastFrame;          //lastFrame = currentFrame;          double  timeValue = glfwGetTime();          float deltaTime  = static\_cast<float>(sin(timeValue) / 2.0 + 0.5);          //int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");          //glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);          // input          // -----          processInput(window);          // render          // ------          glClearColor(0.1f, 0.1f, 0.1f, 1.0f);          glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);          //light source movig part          //lightPos.x = 1.0f + sin(glfwGetTime()) \* 2.0f;          //lightPos.y = sin(glfwGetTime() / 2.0f) \* 1.0f;          // be sure to activate shader when setting uniforms/drawing objects          lightingShader.use();          lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);          lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);          //lightingShader.setVec3("lightPos", lightPos.x,lightPos.y,1.0);          lightingShader.setVec3("lightPos", lightPos);          lightingShader.setVec3("viewPos", camera.Position);          // view/projection transformations          glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);          glm::mat4 view = camera.GetViewMatrix();          lightingShader.setMat4("projection", projection);          lightingShader.setMat4("view", view);          // world transformation          glm::mat4 model = glm::mat4(1.0f);          lightingShader.setMat4("model", model);          // render the cube          glBindVertexArray(cubeVAO);          glDrawArrays(GL\_TRIANGLES, 0, 36);          // also draw the lamp object          lightCubeShader.use();          lightCubeShader.setMat4("projection", projection);          lightCubeShader.setMat4("view", view);          model = glm::mat4(1.0f);          model = glm::translate(model, lightPos);          model = glm::scale(model, glm::vec3(0.2f)); // a smaller cube          lightCubeShader.setMat4("model", model);          glBindVertexArray(lightCubeVAO);          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, &cubeVAO);      glDeleteVertexArrays(1, &lightCubeVAO);      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);**  **// if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)**  **//     camera.ProcessKeyboard(FORWARD, deltaTime);**  **// if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)**  **//     camera.ProcessKeyboard(BACKWARD, deltaTime);**  **// if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)**  **//     camera.ProcessKeyboard(LEFT, deltaTime);**  **// if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)**  **//     camera.ProcessKeyboard(RIGHT, deltaTime);**  **}**  // 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);  }  //glfw: whenever the mouse moves, this callback is called  //-------------------------------------------------------  void mouse\_callback(GLFWwindow\* window, double xposIn, double yposIn)  {      float xpos = static\_cast<float>(xposIn);      float ypos = static\_cast<float>(yposIn);      if (firstMouse)      {          lastX = xpos;          lastY = ypos;          firstMouse = false;      }      float xoffset = xpos - lastX;      float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top      lastX = xpos;      lastY = ypos;      camera.ProcessMouseMovement(xoffset, yoffset);  }  //glfw: whenever the mouse scroll wheel scrolls, this callback is called  //----------------------------------------------------------------------  void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)  {      camera.ProcessMouseScroll(static\_cast<float>(yoffset));  }  Specular.fs  #version 330 core  out vec4 FragColor;  in vec3 Normal;  in vec3 FragPos;    uniform vec3 lightPos;  uniform vec3 viewPos;  uniform vec3 lightColor;  uniform vec3 objectColor;  void main()  {      // ambient  **float ambientStrength = 0.2;**      vec3 ambient = ambientStrength \* lightColor;        // diffuse      vec3 norm = normalize(Normal);      vec3 lightDir = normalize(lightPos - FragPos);      float diff = max(dot(norm, lightDir), 0.0);      vec3 diffuse = diff \* lightColor;        // specular  **float specularStrength = 0.2;**      vec3 viewDir = normalize(viewPos - FragPos);      vec3 reflectDir = reflect(-lightDir, norm);  **float spec = pow(max(dot(viewDir, reflectDir), 0.0), 20);**      vec3 specular = specularStrength \* spec \* lightColor;        vec3 result = (ambient + specular) \* objectColor;      FragColor = vec4(result, 1.0);  }  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  out vec3 FragPos;  out vec3 Normal;  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main()  {  FragPos = vec3(model \* vec4(aPos, 1.0));  Normal = mat3(transpose(inverse(model))) \* aNormal;    gl\_Position = projection \* view \* vec4(FragPos, 1.0);  }  #version 330 core  out vec4 FragColor;  void main()  {      FragColor = vec4(1.0); // set all 4 vector values to 1.0  }  #version 330 core  layout (location = 0) in vec3 aPos;  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main()  {  gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);  } |

**Output:**

