**Roll No: 1703016**

**Lab Final**

**Lab Task Q1**

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

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

**b) Shader/Texture: Mix of 2 different Textures for each.**

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

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

**main.cpp**

|  |
| --- |
| // Roll: 1703016  #include "glad.h"  #include "glfw3.h"  #define STB\_IMAGE\_IMPLEMENTATION  #include "stb\_image.h"  // #include "learnopengl/filesystem.h"  // #include "learnopengl/shader\_s.h"  #include "glm/glm.hpp"  #include "glm/gtc/matrix\_transform.hpp"  #include <sstream>  #include <fstream>  #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/shader/4.1.texture.vs", "src/shader/4.1.texture.fs");  const char\* vertexPath = "src/shader/template1.vs";  const char\* fragmentPath = "src/shader/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,  -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,  -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, 0.0f, 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,  -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, 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, 0.0f, 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,  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, 0.0f, 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, 1.0f, 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, 0.0f, 0.0f,  -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, 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, 1.0f, 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, 0.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 0.0f, 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)  };  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("resources/textures/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("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" << 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);    **unsigned int number\_of\_cube = 2;**  **for (unsigned int i = 0; i < number\_of\_cube; 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 \* (0);**  **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, 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)  {  //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);  **if (glfwGetKey(window, GLFW\_KEY\_K) == GLFW\_PRESS) {**  **glm::vec3 cubePosition[] = {**  **glm::vec3( 0.0f, 5.0f, 0.0f),**  **glm::vec3( 2.0f, 8.0f, -15.0f),**  **glm::vec3(-1.5f, -5.2f, -2.5f)**  **};**    **unsigned int number\_of\_cube = 2;**  **for (unsigned int i = 0; i < number\_of\_cube; 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, cubePosition[i]);**  **// float angle = 20.0f \* (0);**  **// 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, 36);**  **}**  **}**  **}**  // 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 shader**

|  |
| --- |
| #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);  ourColor = aColor;  TexCoord = vec2(aTexCoord.x, aTexCoord.y);  } |

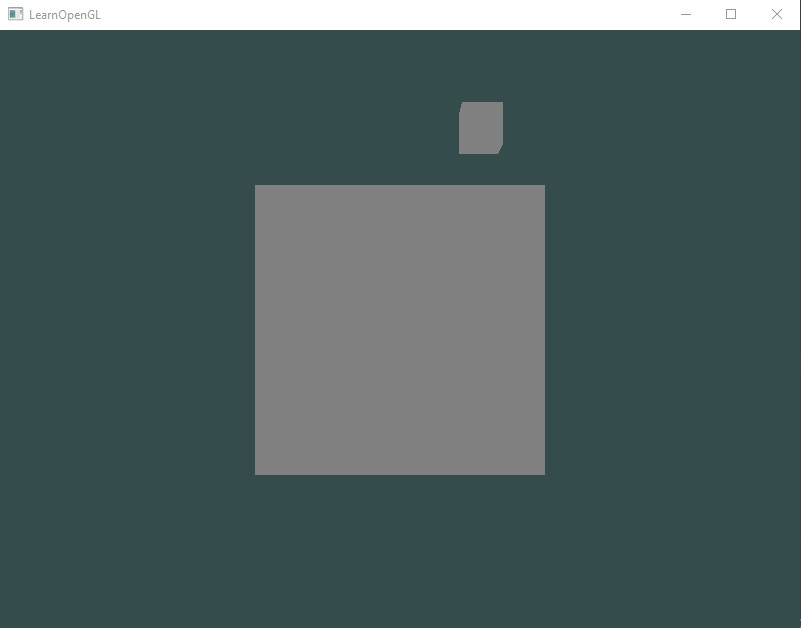
**fragment shader [Q1(a)]**

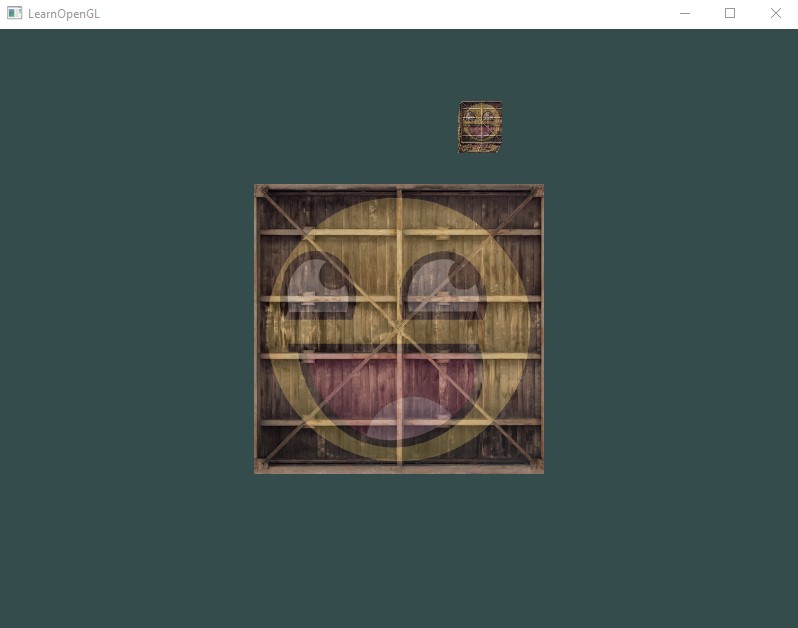
|  |
| --- |
| #version 330 core  out vec4 FragColor;  in vec3 ourColor;  in vec2 TexCoord;  // texture samplers  uniform sampler2D texture1;  uniform sampler2D texture2;  void main()  {  **FragColor = vec4(0.5f, 0.5f, 0.5f, 1.0f);**  } |

**fragment shader [Q1(b)]**

|  |
| --- |
| #version 330 core  out vec4 FragColor;  in vec3 ourColor;  in vec2 TexCoord;  // texture samplers  uniform sampler2D texture1;  uniform sampler2D texture2;  void main()  {  **FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);**  } |

**Output (ScreenShot):**

a)

b)

c)

**Lab Task Q2**

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

**a) Camera: Camera will move along the +x axis with time.**

**b) Lighting: 40% diffuse +50% specular**

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

**main.cpp**

|  |
| --- |
| #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 "learnopengl/shader\_m.h"  #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  glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);  glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);  glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);  bool firstMouse = true;  float yaw = -90.0f; // yaw is initialized to -90.0 degrees since a yaw of 0.0 results in a direction vector pointing to the right so we initially rotate a bit to the left.  float pitch = 0.0f;  float lastX = 800.0f / 2.0;  float lastY = 600.0 / 2.0;  float fov = 45.0f;  // timing  float deltaTime = 0.0f; // time between current frame and last frame  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 program  // ------------------------------------  Shader lightingShader("src/shader/template2\_lighting.vs", "src/shader/template2\_lighting.fs");  Shader lightCubeShader("src/shader/template2\_light\_cube.vs", "src/shader/template2\_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;  // 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!  // 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);  lightingShader.setVec3("viewPos", cameraPos);  // view/projection transformations  glm::mat4 projection = glm::perspective(glm::radians(fov), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);  **// camera transformation**  **glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);**  **view = glm::translate(view, glm::vec3(0.5f, -0.5f, 0.0f));**  **view = glm::rotate(view, (float)glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));**    // LightingShader  glm::mat4 model = glm::mat4(1.0f);  lightingShader.setMat4("projection", projection);  lightingShader.setMat4("view", view);  lightingShader.setMat4("model", model);  // render cube  glBindVertexArray(cubeVAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  // LightCubeShader  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);  // render lighting cube  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);  float cameraSpeed = static\_cast<float>(1.0 \* deltaTime);    if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)  cameraPos += cameraSpeed \* cameraFront;    if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)  cameraPos -= cameraSpeed \* cameraFront;    if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)  cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;    if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)  cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;  }  // 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;  float sensitivity = 0.1f; // change this value to your liking  xoffset \*= sensitivity;  yoffset \*= sensitivity;  yaw += xoffset;  pitch += yoffset;  // make sure that when pitch is out of bounds, screen doesn't get flipped  if (pitch > 89.0f)  pitch = 89.0f;  if (pitch < -89.0f)  pitch = -89.0f;  glm::vec3 front;  front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));  front.y = sin(glm::radians(pitch));  front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));  cameraFront = glm::normalize(front);  }  // glfw: whenever the mouse scroll wheel scrolls, this callback is called  // ----------------------------------------------------------------------  void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)  {  fov -= (float)yoffset;  if (fov < 1.0f)  fov = 1.0f;  if (fov > 45.0f)  fov = 45.0f;  } |

**vertex shader**

|  |
| --- |
| #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);  } |

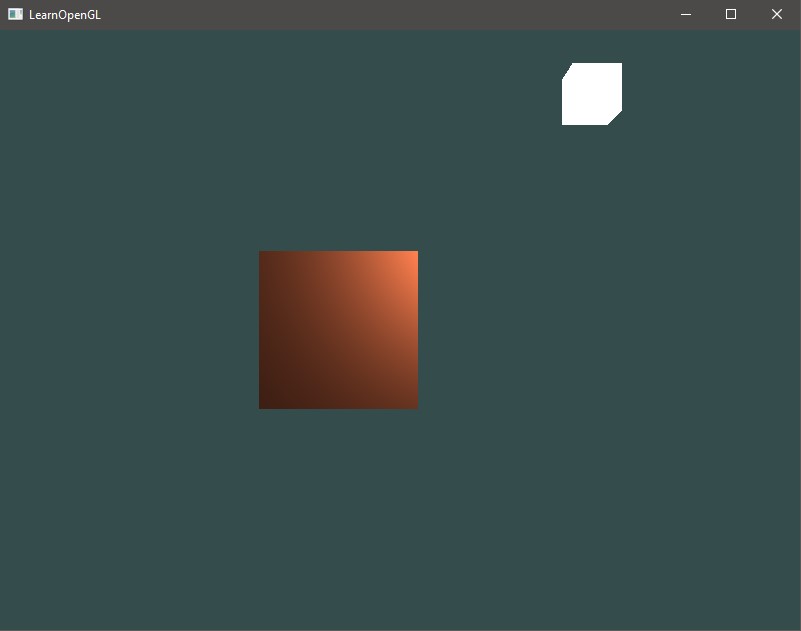
**fragment shader [Q2(a)]**

|  |
| --- |
| #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 = 0.1;  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 = 1.0;**  vec3 viewDir = normalize(viewPos - FragPos);  vec3 reflectDir = reflect(-lightDir, norm);  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 4);  vec3 specular = specularStrength \* spec \* lightColor;    **vec3 result = (specular) \* objectColor;**  FragColor = vec4(result, 1.0);  } |

**fragment shader [Q2(b)]**

|  |
| --- |
| #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 = 0.1;  vec3 ambient = ambientStrength \* lightColor;    // diffuse  **float diffuseStrength = 0.4;**  vec3 norm = normalize(Normal);  vec3 lightDir = normalize(lightPos - FragPos);  float diff = max(dot(norm, lightDir), 0.0);  **vec3 diffuse = diff \* lightColor \* diffuseStrength;**    // specular  **float specularStrength = 0.5;**  vec3 viewDir = normalize(viewPos - FragPos);  vec3 reflectDir = reflect(-lightDir, norm);  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 4);  vec3 specular = specularStrength \* spec \* lightColor;    **vec3 result = (diffuse + specular) \* objectColor;**  FragColor = vec4(result, 1.0);  } |

**Output (ScreenShot):**

a)

b)

