Roll No: 1703150

Lab Final

Lab Task Q1

Question:

Q1. Show an OpenGL program which will show:

a) Hello Triangle/Shapes: Two 2D-Square

b) Shader/Texture: Mix of two different textures for each.

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

Solution (Bold your own written code):

main.cpp

```
#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;

float rotate_dir = 90.0f;

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);

    // 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,0.0f,  1.0f, 0.0f, 0.0f,   0.0f,
0.0f,
         0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   1.0f,
0.0f,
         0.5f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   1.0f,
1.0f,
         0.5f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   1.0f,
1.0f,
        -0.5f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   0.0f,
1.0f,
```

```
        -0.5f, -0.5f, 0.0f,  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),
        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);

        for (unsigned int i = 0; i < 2; 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(rotate_dir), glm::vec3(0.0f, 0.0f, 1.0f));

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);
    else if (glfwGetKey(window, GLFW_KEY_L) ==
GLFW_PRESS)
        {
            rotate_dir = 90.0;
        }
        else if (glfwGetKey(window, GLFW_KEY_R) ==
GLFW_PRESS)
        {
            rotate_dir  = -90.0;
        }
}

// 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

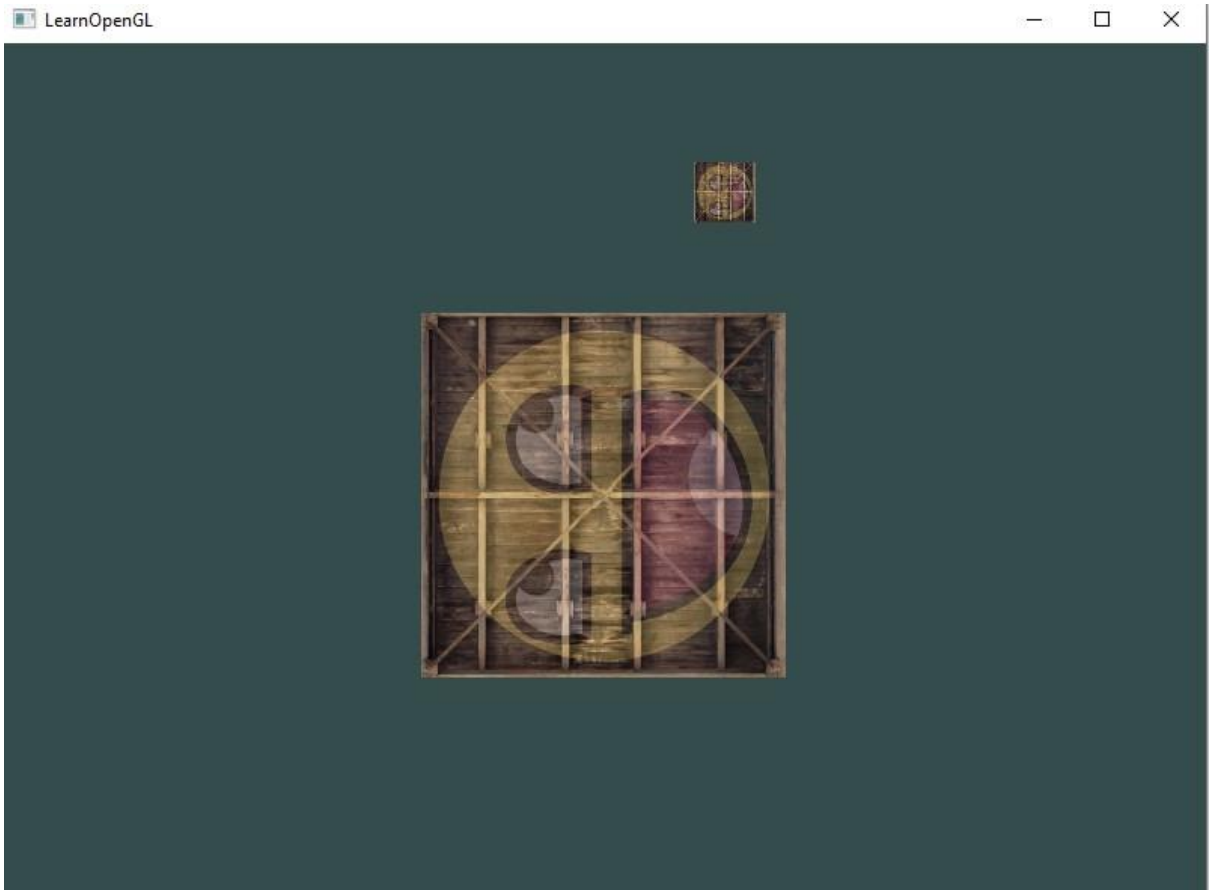
```
#version 330 core
out vec4 FragColor;

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) Left rotaion



Right rotation

Lab Task Q2

Question:

Q2. 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 move along the -z axis using keyboard.

b) Lighting: 20% ambient + 15% specular

Solution (Bold your own written code):

main.cpp

```
#include "glad.h"
#include "glfw3.h"

#include "glm/glm/glm.hpp"
#include "glm/glm/gtc/matrix_transform.hpp"
#include "glm/glm/gtc/type_ptr.hpp"

#include "shader_m.h"
#include "camera.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
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, 0.0f, 0.0f);

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/colors.vs",
"src/colors.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,
         0.5f, -0.5f, -0.5f,
         0.5f,  0.5f, -0.5f,
         0.5f,  0.5f, -0.5f,
        -0.5f,  0.5f, -0.5f,
        -0.5f, -0.5f, -0.5f,

        -0.5f, -0.5f,  0.5f,
         0.5f, -0.5f,  0.5f,
         0.5f,  0.5f,  0.5f,
         0.5f,  0.5f,  0.5f,
        -0.5f,  0.5f,  0.5f,
        -0.5f, -0.5f,  0.5f,

        -0.5f,  0.5f,  0.5f,
        -0.5f,  0.5f, -0.5f,
        -0.5f, -0.5f, -0.5f,
```

```
        -0.5f, -0.5f, -0.5f,
        -0.5f, -0.5f,  0.5f,
        -0.5f,  0.5f,  0.5f,

         0.5f,  0.5f,  0.5f,
         0.5f,  0.5f, -0.5f,
         0.5f, -0.5f, -0.5f,
         0.5f, -0.5f, -0.5f,
         0.5f, -0.5f,  0.5f,
         0.5f,  0.5f,  0.5f,

        -0.5f, -0.5f, -0.5f,
         0.5f, -0.5f, -0.5f,
         0.5f, -0.5f,  0.5f,
         0.5f, -0.5f,  0.5f,
        -0.5f, -0.5f,  0.5f,
        -0.5f, -0.5f, -0.5f,

        -0.5f,  0.5f, -0.5f,
         0.5f,  0.5f, -0.5f,
         0.5f,  0.5f,  0.5f,
         0.5f,  0.5f,  0.5f,
        -0.5f,  0.5f,  0.5f,
        -0.5f,  0.5f, -0.5f,
    };
    // 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, 3 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // 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);

    // we only need to bind to the VBO (to link it with
glVertexAttribPointer), no need to fill it; the VBO's
data already contains all we need (it's already bound,
but we do it again for educational purposes)
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
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(0.1f, 0.1f, 0.1f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

        // 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);

        // 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));
}
```

vertex shader

```
                #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);
}
```

fragment shader

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

void main()
{
    FragColor = vec4(lightColor * objectColor, 1.0);
}
```

Output (ScreenShot):

a)