

BURSA ULUDAĞ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2023-2024 EĞİTİM ÖĞRETİM YILI BAHAR DÖNEMİ

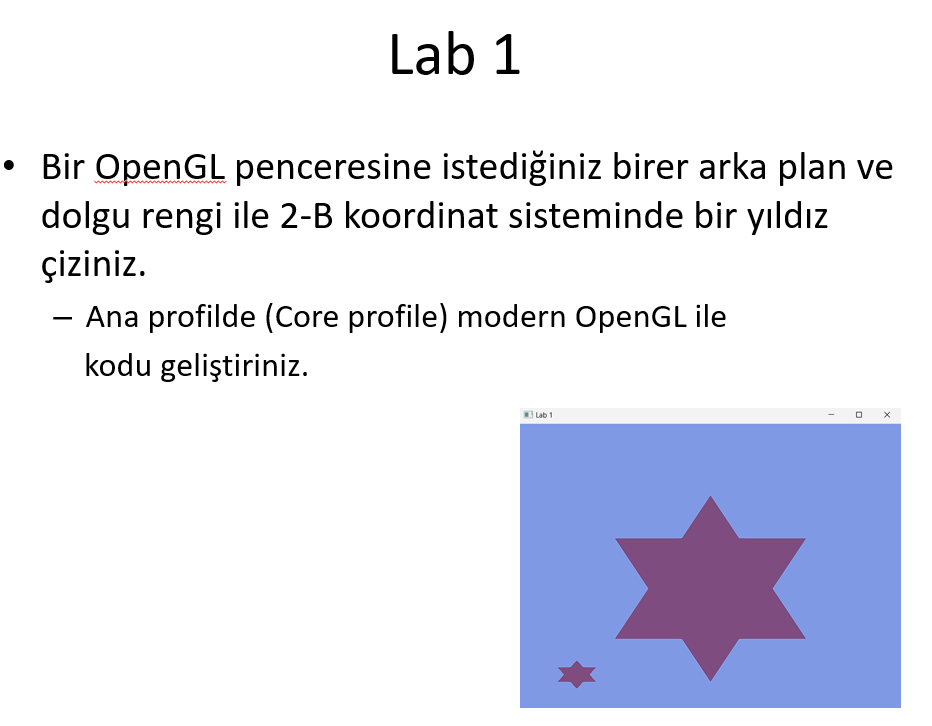
BİLGİSAYAR GRAFİKLERİ RAPORU

MURAT BERK YETİŞTİRİR

032290008

[032290008@ogr.uludag.edu.tr](mailto:032290008@ogr.uludag.edu.tr)

**SORU:** 02\_Grafik çıktı öğeleri ders notunun 72. slaytına bakınız. Geliştirdiğiniz kodun yanı sıra pencere çıktısının görüntüsünü içeren raporunuzu pdf formatında kaydedip yükleyiniz.

****

**CEVAP KODU:**

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <iostream>

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);

void processInput(GLFWwindow\* window);

// settings

const unsigned int SCR\_WIDTH = 800;

const unsigned int SCR\_HEIGHT = 600;

const char\* vertexShaderSource = "#version 330 core\n"

"layout (location = 0) in vec3 aPos;\n"

"void main()\n"

"{\n"

" gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"

"}\0";

const char\* fragmentShaderSource = "#version 330 core\n"

"out vec4 FragColor;\n"

"void main()\n"

"{\n"

" FragColor = vec4(0.4f, 0.2f, 0.7f, 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

// ------------------------------------

// vertex shader

unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

glCompileShader(vertexShader);

// check for shader compile errors

int success;

char infoLog[512];

glGetShaderiv(vertexShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

// fragment shader

unsigned int fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);

glCompileShader(fragmentShader);

// check for shader compile errors

glGetShaderiv(fragmentShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

// link shaders

unsigned int shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

// check for linking errors

glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

if (!success) {

glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::PROGRAM::LINKING\_FAILED\n" << infoLog << std::endl;

}

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

// set up vertex data (and buffer(s)) and configure vertex attributes

// ------------------------------------------------------------------

float vertices[] = {

-0.5f, -0.5f, 0.0f, // left

0.5f, -0.5f, 0.0f, // right

0.0f, 0.7f, 0.0f, // top

0.5f, 0.3f, 0.0f, // bottom-right

-0.5f, 0.3f, 0.0f, // bottom-left

0.0f, -0.8f, 0.0f, // bottom-middle

- 0.9f, -0.85f, 0.0f, // left

-0.6f, -0.85f, 0.0f, // right

-0.75f, -0.6f, 0.0f, // top

-0.6f, -0.70f, 0.0f, // bottom-right

-0.9f, -0.70f, 0.0f, // bottom-left

-0.75f, -0.95f, 0.0f // bottom-middle

};

unsigned int VBO, VAO;

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

// bind the Vertex Array Object first, then bind and set vertex buffer(s), and then configure vertex attributes(s).

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3\* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

// note that this is allowed, the call to glVertexAttribPointer registered VBO as the vertex attribute's bound vertex buffer object so afterwards we can safely unbind

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// You can unbind the VAO afterwards so other VAO calls won't accidentally modify this VAO, but this rarely happens. Modifying other

// VAOs requires a call to glBindVertexArray anyways so we generally don't unbind VAOs (nor VBOs) when it's not directly necessary.

glBindVertexArray(0);

// 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(0.7f, 0.5f, 0.1f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

// draw our first triangle

glUseProgram(shaderProgram);

glBindVertexArray(VAO); // seeing as we only have a single VAO there's no need to bind it every time, but we'll do so to keep things a bit more organized

glDrawArrays(GL\_TRIANGLES, 0, 12);

// glBindVertexArray(0); // no need to unbind it every time

// glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)

// -------------------------------------------------------------------------------

glfwSwapBuffers(window);

glfwPollEvents();

}

// optional: de-allocate all resources once they've outlived their purpose:

// ------------------------------------------------------------------------

glDeleteVertexArrays(1, &VAO);

glDeleteBuffers(1, &VBO);

glDeleteProgram(shaderProgram);

// glfw: terminate, clearing all previously allocated GLFW resources.

// ------------------------------------------------------------------

glfwTerminate();

return 0;

}

// process all input: query GLFW whether relevant keys are pressed/released this frame and react accordingly

// ---------------------------------------------------------------------------------------------------------

void processInput(GLFWwindow\* window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

}

// glfw: whenever the window size changed (by OS or user resize) this callback function executes

// ---------------------------------------------------------------------------------------------

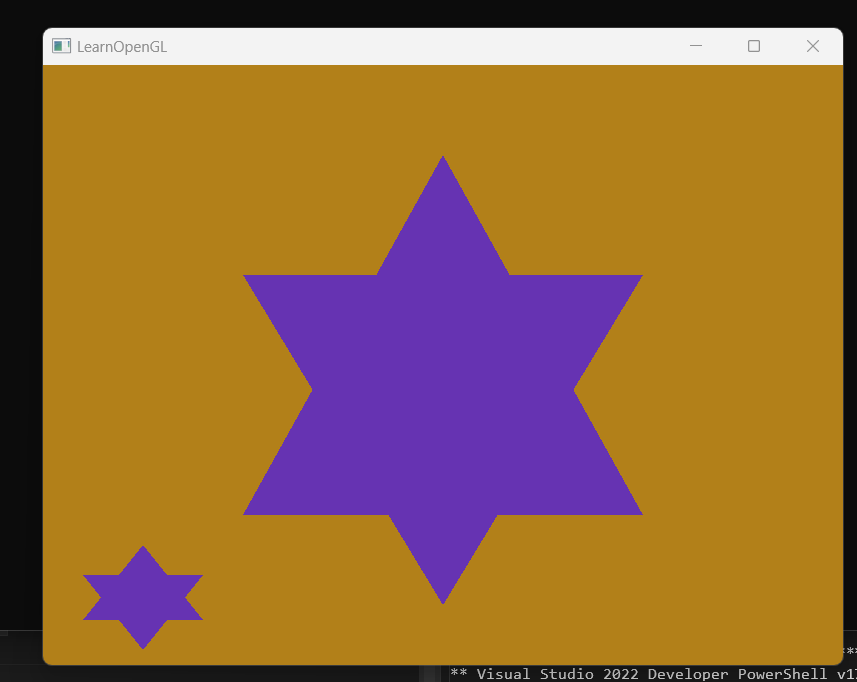
void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height)

{

glViewport(0, 0, width, height);

}

**CEVAP EKRAN GÖRÜNTÜSÜ:**

****