BURSA ULUDAĞ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2024-2025 EĞİTİM ÖĞRETİM YILI BAHAR DÖNEMİ

BİLGİSAYAR GRAFİKLERİ RAPORU

MURAT BERK YETİŞTİRİR

032290008

032290008@ogr.uludag.edu.tr

SORU: 3-B silindirik bir tel çerçeve temsilini belirlenen hassasiyet katsayısıyla çizdiren etkileşimli bir uygulama geliştiriniz. Tel çerçeveye Y klavye tuşuyla yeşil mermer, G klavye tuşuyla gri metalik doku giydiriniz. Ok yönleriyle modelin sağ-sol ve yukarı-aşağı hareketini sağlayınız. R klavye tuşuyla modelin y ekseni etrafında dönmesini sağlayınız. T klavye tuşuyla tel çerçeve moduna geçiniz. Bu modda, Gösterim ekranında gerçekleşen bir fare tıklaması için sahne sınırlarına göre farenin tıklandığı P0 konumundan –z eksenine doğru sahneye dik (ortogonal) bir ışın gönderildiğini varsayınız. Işının tel çerçeveyi kestiği silindirik poligonu son basılan klavye tuşuna göre yeşil mermer veya gri metalik dokuyla görselleyiniz. Işının tel çerçeveyi kestiği herhangi bir poligon yoksa tel çerçevenin tel rengini rasgele güncelleyiniz. Tam yorumlu kodunuzu ve OpenGL çıktısını içeren bir rapor hazırlayınız. Raporun içine ve dosya ismine adınızı, soyadınızı ve öğrenci numaranızı yazınız. Dosyayı pdf olarak kaydedip son teslim tarihinden önce UKEY'deki Lab4 ödevi arayüzüne yükleyiniz. Işın poligon kesişimi için ışın-normal yönelimi, ışın-düzlem kesişimi ve içinde-dışında testlerinden yararlanınız. shader ve stb_image kütüphanelerini ekleyiniz.

CEVAP KODUM:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#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 <learnopengl/shader_m.h>,
#include <iostream>
#include <vector>

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;

bool useFirstTexture = true;
float rotationAngle = 0.0f;
bool rotateModel = false;
bool wireframeMode = false;

void generatePrismVertices(std::vector<float>& vertices) {
    const int SIDES = 10;
    const float PI = 3.1415926f;
    const float RADIUS = 0.5f;
    const float HEIGHT = 1.0f;

    vertices.clear();

    float centerBottom[] = { 0.0f, -HEIGHT / 2.0f, 0.0f, 0.5f, 0.5f };
    float centerTop[] = { 0.0f, HEIGHT / 2.0f,  0.0f, 0.5f, 0.5f };

    for (int i = 0; i < SIDES; ++i) {
        float angle1 = (2.0f * PI / SIDES) * i;
        float angle2 = (2.0f * PI / SIDES) * (i + 1);
```

```
        float x1 = cos(angle1) * RADIUS;
        float z1 = sin(angle1) * RADIUS;
        float x2 = cos(angle2) * RADIUS;
        float z2 = sin(angle2) * RADIUS;

        vertices.insert(vertices.end(), {
            centerBottom[0], centerBottom[1], centerBottom[2], centerBottom[3], centerBottom[4],
            x2, -HEIGHT / 2.0f, z2, 0.0f, 0.0f,
            x1, -HEIGHT / 2.0f, z1, 1.0f, 0.0f
            });

        vertices.insert(vertices.end(), {
            centerTop[0], centerTop[1], centerTop[2], centerTop[3], centerTop[4],
            x1, HEIGHT / 2.0f, z1, 1.0f, 1.0f,
            x2, HEIGHT / 2.0f, z2, 0.0f, 1.0f
            });

        vertices.insert(vertices.end(), {
            x1, HEIGHT / 2.0f, z1, 0.0f, 1.0f,
            x2, HEIGHT / 2.0f, z2, 1.0f, 1.0f,
            x1, -HEIGHT / 2.0f, z1, 0.0f, 0.0f
            });

        vertices.insert(vertices.end(), {
            x2, HEIGHT / 2.0f, z2, 1.0f, 1.0f,
            x2, -HEIGHT / 2.0f, z2, 1.0f, 0.0f,
            x1, -HEIGHT / 2.0f, z1, 0.0f, 0.0f
            });
    }
}

int main()
{
    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, "Prisma", 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;
    }

    glEnable(GL_DEPTH_TEST);

    Shader ourShader("7.3.camera.vs", "7.3.camera.fs");

    std::vector<float> prismVertices;
    generatePrismVertices(prismVertices);
    // world space positions of our cubes
    glm::vec3 prismPosition[] = {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, prismVertices.size() * sizeof(float), prismVertices.data(), GL_STATIC_DRAW);

    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // texture coord attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    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;
    unsigned char* data = stbi_load(FileSystem::getPath("resources/textures/marble.png").c_str(), &width,
&height, &nrChannels, 0);
    if (data) {
        GLenum format = (nrChannels == 4) ? GL_RGBA : GL_RGB;
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    // 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
    int width1, height1, nrChannels1;
    unsigned char* data1 = stbi_load(FileSystem::getPath("resources/textures/metal.png").c_str(), &width1,
&height1, &nrChannels1, 0);
    if (data1) {
        GLenum format1 = (nrChannels1 == 4) ? GL_RGBA : GL_RGB;
        glTexImage2D(GL_TEXTURE_2D, 0, format1, width1, height1, 0, format1, GL_UNSIGNED_BYTE, data1);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    stbi_image_free(data1);

    ourShader.use();
    ourShader.setInt("texture1", 0);
    ourShader.setInt("texture2", 1);

    GLuint currentTexture = texture1;

    while (!glfwWindowShouldClose(window))
    {
        float currentFrame = static_cast<float>(glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        processInput(window);

        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // bind textures on corresponding texture units
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture1);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture2);
```

```
        // activate shader
        ourShader.use();
        ourShader.setBool("useFirstTexture", useFirstTexture);

        // pass projection matrix to shader (note that in this case it could change every frame)
        glm::mat4 projection = glm::perspective(glm::radians(fov), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f,
100.0f);
        ourShader.setMat4("projection", projection);

        // camera/view transformation
        glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
        ourShader.setMat4("view", view);

        if (rotateModel)
            rotationAngle += 0.01f;

        // render boxes
        glBindVertexArray(VAO);
        for (unsigned int i = 0; i < 10; i++)
        {
            glm::mat4 model = glm::mat4(1.0f);
            model = glm::translate(model, prismPosition[i]);

            float staticAngle = 20.0f * i;
            model = glm::rotate(model, glm::radians(staticAngle), glm::vec3(1.0f, 0.3f, 0.5f));

            // dinamik Y ekseni dönüşü (R tuşu ile kontrol edilir)
            if (rotateModel)
                model = glm::rotate(model, rotationAngle, glm::vec3(0.0f, 1.0f, 0.0f));

            ourShader.setMat4("model", model);
            glDrawArrays(GL_TRIANGLES, 0, prismVertices.size() / 5); // 5 = position(3) + texcoord(2)
        }

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);

    glfwTerminate();
    return 0;
}

// cpp dosyası (örneğin geometry.cpp)


void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    float cameraSpeed = static_cast<float>(2.5 * 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;
    if (glfwGetKey(window, GLFW_KEY_Y) == GLFW_PRESS)
        useFirstTexture = true;
    if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS)
        useFirstTexture = false;
    if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS)
        rotateModel = true;
    if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS)
    {
        wireframeMode = !wireframeMode;
        glPolygonMode(GL_FRONT_AND_BACK, wireframeMode ? GL_LINE : GL_FILL);
    }
    else
        rotateModel = false;
}
```

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

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);
}

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;
}
```

CEVAP EKRAN ÇIKTISI