**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Компьютерная графика»

**Лабораторная работа № 6**

Тема: Создание шейдерных анимационных эффектов в OpenGL

Студент: Лошманов Юрий Андреевич

Группа: 80-306

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2021

1. **Постановка задачи**

Для поверхности, созданной в л.р. №4-5, обеспечить выполнение следующего шейдерного эффекта - Анимация. Изменение цвета источника рассеянного света по синусоидальному закону.

1. **Описание программы**

Программа состоит из окна для взаимодействия с многогранником, из горячих клавиш для визуализации, вращения многогранника и изменения некоторых его параметров.

1. **Набор тестов**

1. Изначальный вид.

2. Вращение многогранника.

3. Увеличение точности аппроксимации.

4. Приближения источника света к многограннику.

5. Отдаление источника света от многогранника.

1. **Результаты выполнения тестов**

1. Изначальный вид.

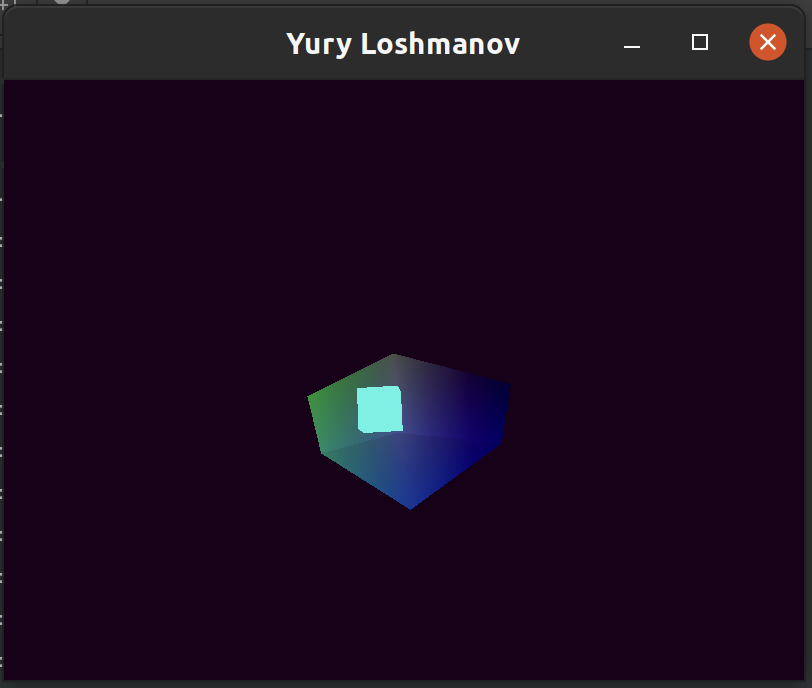


Рис. 4.1 Результат открытия окна

2. Вращение многогранника.

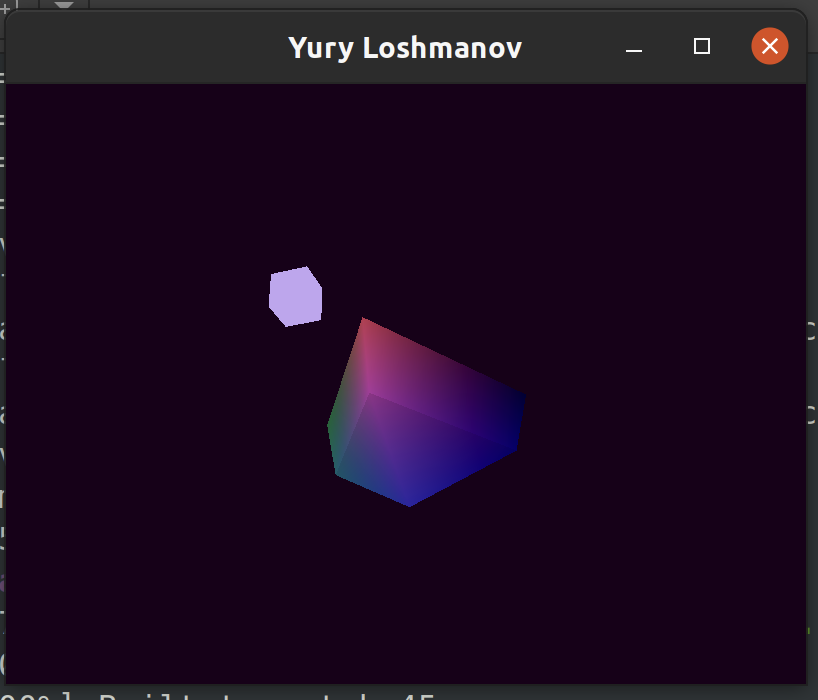


Рис. 4.2 Результат вращения многогранника

3. Увеличение точности аппроксимации.

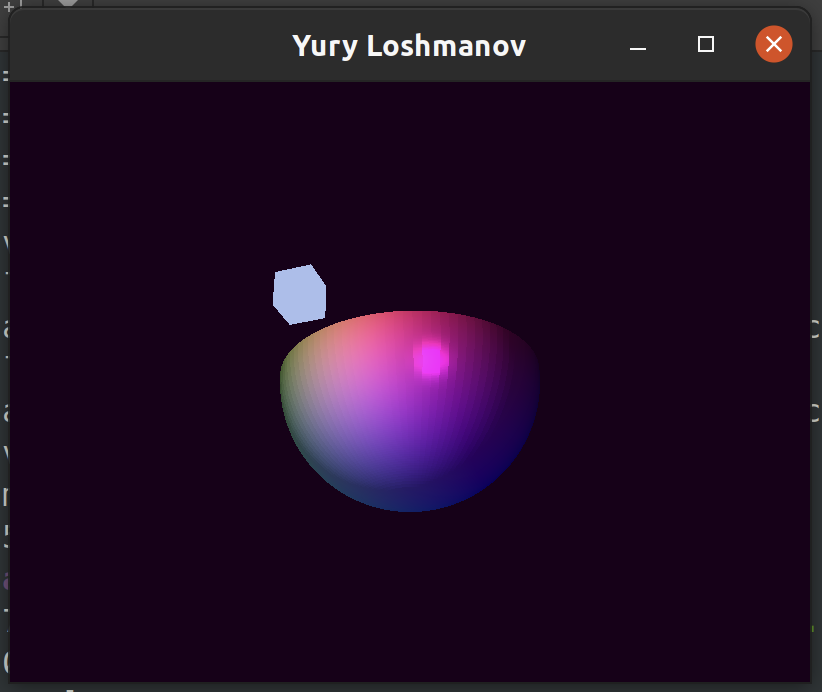


Рис. 4.3 Результат увеличение точности аппроксимации

4. Приближение источника света к многограннику.

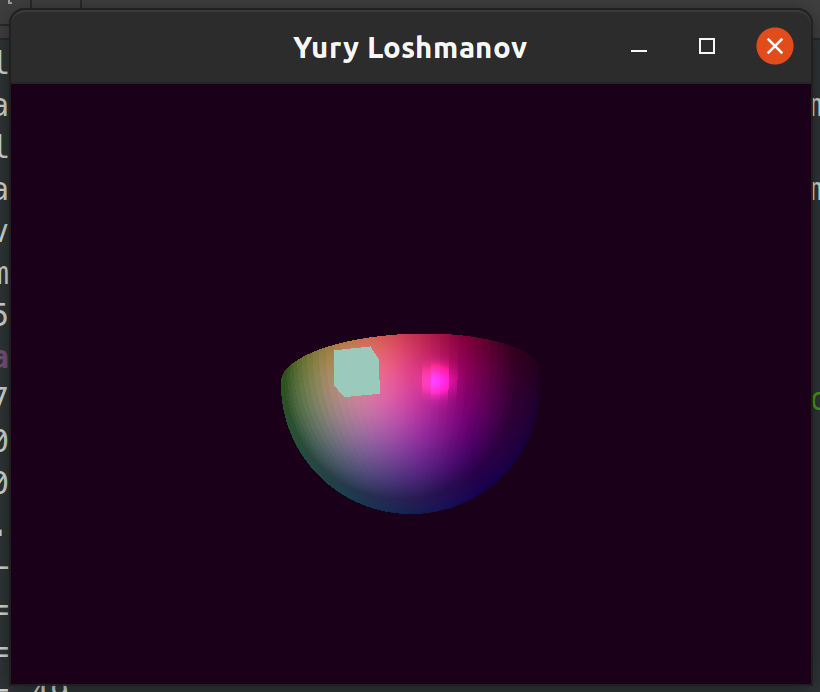


Рис. 4.5 Результат приближения источника света к многограннику

5. Отдаление источника света от многогранника.

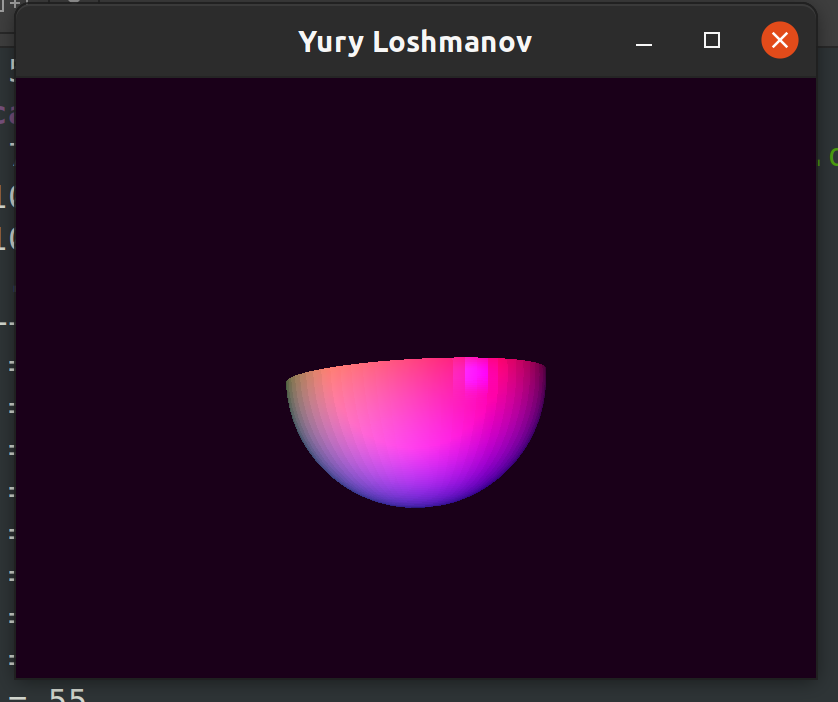


Рис. 4.6 Результат отдаления источника света от многогранника

1. **Листинг программы**

**main.cpp**

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtc/type\_ptr.hpp>

#include "math\_help.hpp"

// do not use nullptr in OpenGL functions

// #define nullptr NULL

void key\_callback(GLFWwindow \*window, int key, int scancode, int action, int mode);

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

std::vector<char> shader\_source\_code(const std::string &filename);

GLuint createShader(GLuint shaderType, const std::string &file);

GLuint attachAndLinkShaders(GLuint vertexShader, GLuint fragmentShader);

std::pair<std::vector<float>, std::vector<unsigned>> customFigure(size\_t precision);

std::pair<std::vector<float>, std::vector<unsigned>> cubeFigure();

const GLuint WIDTH = 800, HEIGHT = 600;

int figurePrecision = 2;

bool recalculateFigure = false;

float FOV = 45.0f;

float scaleX = 1.0f;

float scaleY = 1.0f;

float scaleZ = 1.0f;

float scaleSpeed = 1.0f;

float RotateX = -175.0f;

float RotateY = 55.0f;

float RotateZ = 0.0f;

glm::vec3 cameraPos = glm::vec3(4.0f, 0.0f, 4.0f);

glm::vec3 cameraFront = glm::vec3(-1.0f, 0.0f, -1.0f);

glm::vec3 cameraUp = glm::vec3(0.0f, 0.0f, 1.0f);

glm::vec3 lightPos(2.0f, 0.0f, 0.5f);

float ambientStrength = 0.3;

float diffuseStrength = 0.7;

float specularStrength = 0.9;

int specularPow = 32;

int main() {

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwWindowHint(GLFW\_RESIZABLE, GL\_TRUE);

GLFWwindow \*window = glfwCreateWindow(WIDTH, HEIGHT, "Yury Loshmanov", NULL, NULL);

glfwMakeContextCurrent(window);

if (window == NULL) {

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return EXIT\_FAILURE;

}

glfwSetKeyCallback(window, key\_callback);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

if (!gladLoadGLLoader((GLADloadproc) glfwGetProcAddress)) {

std::cout << "Failed to initialize OpenGL context" << std::endl;

return EXIT\_FAILURE;

}

auto tempTriangles = customFigure(figurePrecision);

std::vector<float> vertices = tempTriangles.first;

std::vector<unsigned> indices = tempTriangles.second;

auto tempCubeTriangles = cubeFigure();

std::vector<float> cubeVertices = tempCubeTriangles.first;

std::vector<unsigned> cubeIndices = tempCubeTriangles.second;

GLuint vertexShader = createShader(GL\_VERTEX\_SHADER, "../shaders/shader.vert");

GLuint fragmentShader = createShader(GL\_FRAGMENT\_SHADER, "../shaders/shader.frag");

GLuint shaderProgram = attachAndLinkShaders(vertexShader, fragmentShader);

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

GLuint lightVertexShader = createShader(GL\_VERTEX\_SHADER, "../shaders/lightShader.vert");

GLuint lightFragmentShader = createShader(GL\_FRAGMENT\_SHADER, "../shaders/lightShader.frag");

GLuint lightShaderProgram = attachAndLinkShaders(lightVertexShader, lightFragmentShader);

glDeleteShader(lightVertexShader);

glDeleteShader(lightFragmentShader);

std::cout << "--- SHADERS INITIALIZED ---\n";

GLuint VAO1, VBO1, EBO1;

glGenVertexArrays(1, &VAO1);

glBindVertexArray(VAO1);

glGenBuffers(1, &VBO1);

glGenBuffers(1, &EBO1);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO1);

glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(float), vertices.data(), GL\_DYNAMIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO1);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(unsigned), indices.data(), GL\_DYNAMIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 9 \* sizeof(float), (void \*) 0);

glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 9 \* sizeof(float), (void \*) (3 \* sizeof(float)));

glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 3, GL\_FLOAT, GL\_FALSE, 9 \* sizeof(float), (void \*) (6 \* sizeof(float)));

glEnableVertexAttribArray(2);

GLuint lightVAO, lightVBO, lightEBO;

glGenVertexArrays(1, &lightVAO);

glBindVertexArray(lightVAO);

glGenBuffers(1, &lightVBO);

glGenBuffers(1, &lightEBO);

glBindBuffer(GL\_ARRAY\_BUFFER, lightVBO);

glBufferData(GL\_ARRAY\_BUFFER, cubeVertices.size() \* sizeof(float), cubeVertices.data(), GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, lightEBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, cubeIndices.size() \* sizeof(unsigned), cubeIndices.data(), GL\_STATIC\_DRAW);

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

glEnableVertexAttribArray(0);

glUseProgram(shaderProgram);

glCullFace(GL\_CW);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glm::mat4 projection = glm::perspective(glm::radians(FOV), (float) WIDTH / (float) HEIGHT, 0.1f, 100.0f);

while (!glfwWindowShouldClose(window)) {

float time = (float) glfwGetTime();

glm::vec3 lightColor(std::abs(std::sin(time)),

std::abs(std::cos(time)),

std::abs(std::sin(time \* 2 + 0.5)));

glClearColor(0.1f, 0.0f, 0.1f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glBindVertexArray(VAO1);

glUseProgram(shaderProgram);

glUniform3f(glGetUniformLocation(shaderProgram, "lightColor"), lightColor.x, lightColor.y, lightColor.z);

glUniform3f(glGetUniformLocation(shaderProgram, "lightPos"), lightPos.x, lightPos.y, lightPos.z);

glUniform1f(glGetUniformLocation(shaderProgram, "time"), time);

glUniform1f(glGetUniformLocation(shaderProgram, "ambientStrength"), ambientStrength);

glUniform1f(glGetUniformLocation(shaderProgram, "diffuseStrength"), diffuseStrength);

glUniform1f(glGetUniformLocation(shaderProgram, "specularStrength"), specularStrength);

glUniform1i(glGetUniformLocation(shaderProgram, "specularPow"), specularPow);

if (recalculateFigure) {

tempTriangles = customFigure(figurePrecision);

vertices = tempTriangles.first;

indices = tempTriangles.second;

glBindBuffer(GL\_ARRAY\_BUFFER, VBO1);

glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(float), vertices.data(), GL\_DYNAMIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO1);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(unsigned), indices.data(), GL\_DYNAMIC\_DRAW);

// glNamedBufferData(VBO1, vertices.size() \* sizeof(float), vertices.data(), GL\_DYNAMIC\_DRAW);

// glNamedBufferData(EBO1, indices.size() \* sizeof(unsigned), indices.data(), GL\_DYNAMIC\_DRAW);

recalculateFigure = false;

}

//scaleX = 2\*std::abs(std::sin((float)glfwGetTime() \* scaleSpeed));

//scaleY = std::abs(std::cos((float)glfwGetTime() \* scaleSpeed));

//scaleZ = std::abs(std::cos(time \* scaleSpeed / 6)) + 0.5f;

glm::mat4 model = glm::mat4(1.0f);

//model = glm::rotate(model, (float) glfwGetTime() / 2.0f, glm::vec3(0.0, 0.0, 1.0));

model = glm::scale(model, glm::vec3(scaleX, scaleY, scaleZ));

glm::mat4 view = glm::mat4(1.0f);

view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

view = glm::rotate(view, glm::radians(RotateX), glm::vec3(1.0, 0.0, 0.0));

view = glm::rotate(view, glm::radians(RotateY), glm::vec3(0.0, 1.0, 0.0));

view = glm::rotate(view, glm::radians(RotateZ), glm::vec3(0.0, 0.0, 1.0));

glm::vec3 tempCameraPos = glm::vec3(glm::vec4(cameraPos, 1.0) \* view);

glUniform3f(glGetUniformLocation(shaderProgram, "viewPos"), tempCameraPos.x, tempCameraPos.y, tempCameraPos.z);

glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0);

glBindVertexArray(lightVAO);

glUseProgram(lightShaderProgram);

model = glm::mat4(1.0f);

model = glm::translate(model, lightPos);

model = glm::scale(model, glm::vec3(0.1f, 0.1, 0.1));

glUniformMatrix4fv(glGetUniformLocation(lightShaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

glUniformMatrix4fv(glGetUniformLocation(lightShaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

glUniformMatrix4fv(glGetUniformLocation(lightShaderProgram, "projection"), 1, GL\_FALSE,

glm::value\_ptr(projection));

glUniform3f(glGetUniformLocation(lightShaderProgram, "lightColor"), lightColor.x, lightColor.y, lightColor.z);

glDrawElements(GL\_TRIANGLES, cubeIndices.size(), GL\_UNSIGNED\_INT, 0);

glfwSwapBuffers(window);

glfwPollEvents();

}

glDeleteVertexArrays(1, &VAO1);

glDeleteBuffers(1, &VBO1);

glDeleteBuffers(1, &EBO1);

glDeleteProgram(shaderProgram);

glDeleteVertexArrays(1, &lightVAO);

glDeleteBuffers(1, &lightVBO);

glDeleteBuffers(1, &lightEBO);

glDeleteProgram(lightShaderProgram);

glfwTerminate();

return EXIT\_SUCCESS;

}

void key\_callback(GLFWwindow \*window, int key, int scancode, int action, int mode) {

//std::cout << key << std::endl;

float lightSpeed = 0.3f;

float rotateSpeed = 3.0f;

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

glfwSetWindowShouldClose(window, GL\_TRUE);

} else if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {

lightPos.x -= lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS) {

lightPos.x += lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS) {

lightPos.y -= lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS) {

lightPos.y += lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS) {

lightPos.z -= lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_E) == GLFW\_PRESS) {

lightPos.z += lightSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_Z) == GLFW\_PRESS) {

std::cout << "z = " << RotateZ << std::endl;

RotateZ += rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_X) == GLFW\_PRESS) {

std::cout << "z = " << RotateZ << std::endl;

RotateZ -= rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_C) == GLFW\_PRESS) {

std::cout << "y = " << RotateY << std::endl;

RotateY += rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_V) == GLFW\_PRESS) {

std::cout << "y = " << RotateY << std::endl;

RotateY -= rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_I) == GLFW\_PRESS) {

std::cout << "x = " << RotateX << std::endl;

RotateX += rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_O) == GLFW\_PRESS) {

std::cout << "x = " << RotateX << std::endl;

RotateX -= rotateSpeed;

} else if (glfwGetKey(window, GLFW\_KEY\_EQUAL) == GLFW\_PRESS) {

figurePrecision++;

recalculateFigure = true;

} else if (glfwGetKey(window, GLFW\_KEY\_MINUS) == GLFW\_PRESS) {

figurePrecision--;

recalculateFigure = true;

} else if (glfwGetKey(window, GLFW\_KEY\_F1) == GLFW\_PRESS) {

ambientStrength = std::max(std::min(ambientStrength - 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F2) == GLFW\_PRESS) {

ambientStrength = std::max(std::min(ambientStrength + 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F3) == GLFW\_PRESS) {

diffuseStrength = std::max(std::min(diffuseStrength - 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F4) == GLFW\_PRESS) {

diffuseStrength = std::max(std::min(diffuseStrength + 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F5) == GLFW\_PRESS) {

specularStrength = std::max(std::min(specularStrength - 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F6) == GLFW\_PRESS) {

specularStrength = std::max(std::min(specularStrength + 0.1f, 1.0f), 0.0f);

} else if (glfwGetKey(window, GLFW\_KEY\_F7) == GLFW\_PRESS) {

specularPow = std::max(std::min(specularPow - 1, 32), 1);

} else if (glfwGetKey(window, GLFW\_KEY\_F8) == GLFW\_PRESS) {

specularPow = std::max(std::min(specularPow + 1, 32), 1);

}

//std::cout << "amb: " << ambientStrength << ", diff: " << diffuseStrength << ", spec: " << specularStrength <<

// " | " << specularPow << '\n';

if (figurePrecision < 2) {

figurePrecision = 2;

}

}

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

glViewport(0, 0, width, height);

}

std::vector<char> shader\_source\_code(const std::string &filename) {

std::ifstream file(filename, std::ios::in | std::ios::ate);

if (!file.is\_open()) {

std::cout << "Failed to open file " << filename << '\n';

std::cout << "If such an error occurs, check executable file path and\n";

std::cout << "set shaders files path relatively to it" << std::endl;

throw std::runtime\_error("");

}

size\_t fileSize = (size\_t) file.tellg();

std::vector<char> buffer(fileSize);

file.seekg(0);

file.read(buffer.data(), fileSize);

file.close();

return buffer;

}

GLuint createShader(GLuint shaderType, const std::string &file) {

std::vector<char> shaderData = shader\_source\_code(file);

shaderData.push\_back('\0');

const char \*shaderSource = shaderData.data();

GLuint shaderID;

shaderID = glCreateShader(shaderType);

glShaderSource(shaderID, 1, &shaderSource, NULL);

glCompileShader(shaderID);

int success;

char infoLog[512];

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

if (!success) {

glGetShaderInfoLog(shaderID, 512, NULL, infoLog);

std::cout << infoLog << std::endl;

throw std::runtime\_error("Failed to compile shader " + file);

}

return shaderID;

}

GLuint attachAndLinkShaders(GLuint vertexShader, GLuint fragmentShader) {

GLuint shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

int success;

char infoLog[512];

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

if (!success) {

glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

std::cout << "failed to link shaders\n";

std::cout << infoLog << std::endl;

throw std::runtime\_error("");

}

return shaderProgram;

}

std::pair<std::vector<float>, std::vector<unsigned>> customFigure(size\_t precision) {

float figureR = 1.0f;

float figureG = 1.0f;

float figureB = 1.0f;

float sphereX = 0.0f;

float sphereY = 0.0f;

float sphereRadius = 1.0f;

std::vector<float> radiuses = math::linspace(sphereRadius, 0.0f, precision);

std::vector<std::vector<std::pair<float, float>>> circles(precision);

float z = 0.0f;

float zStep = sphereRadius / circles.size();

for (size\_t i = 0; i < precision; i++) {

circles[i] = math::regularPolygon(precision + 1, std::sqrt(sphereRadius \* sphereRadius - z \* z), sphereX,

sphereY);

circles[i].push\_back(circles[i][0]);

//std::cout << std::sqrt(1.0f - z \* z) << " ";

z += zStep;

}

std::vector<float> triangles;

z = 0.0f;

zStep = 1.0f / circles.size();

for (size\_t i = 0; i < circles.size() - 1; i++) {

for (size\_t j = 0; j < circles[i].size() - 1; j++) {

glm::vec3 v1(circles[i][j + 1].first - circles[i][j].first,

circles[i][j + 1].second - circles[i][j].second,

z - z);

glm::vec3 v2(circles[i + 1][j].first - circles[i][j + 1].first,

circles[i + 1][j].second - circles[i][j + 1].second,

z + zStep - z);

glm::vec3 n = glm::cross(v1, v2);

//std::cout << n.x << " " << n.y << " " << n.z << std::endl;

triangles.insert(triangles.end(), {

circles[i][j].first, circles[i][j].second, z, n.x, n.y, n.z, figureR, figureG, figureB,

circles[i][j + 1].first, circles[i][j + 1].second, z, n.x, n.y, n.z, figureR, figureG, figureB,

circles[i + 1][j].first, circles[i + 1][j].second, z + zStep, n.x, n.y, n.z, figureR, figureG,

figureB,

});

v1 = glm::vec3(circles[i + 1][j + 1].first - circles[i][j + 1].first,

circles[i + 1][j + 1].second - circles[i][j + 1].second,

z + zStep - z);

v2 = glm::vec3(circles[i + 1][j].first - circles[i + 1][j + 1].first,

circles[i + 1][j].second - circles[i + 1][j + 1].second,

z + zStep - (z + zStep));

n = glm::cross(v1, v2);

triangles.insert(triangles.end(), {

circles[i][j + 1].first, circles[i][j + 1].second, z, n.x, n.y, n.z, figureR, figureG, figureB,

circles[i + 1][j + 1].first, circles[i + 1][j + 1].second, z + zStep, n.x, n.y, n.z, figureR,

figureG, figureB,

circles[i + 1][j].first, circles[i + 1][j].second, z + zStep, n.x, n.y, n.z, figureR, figureG,

figureB,

});

if (i + 1 == circles.size() - 1) {

v1 = glm::vec3(circles[i + 1][j + 1].first - circles[i + 1][j].first,

circles[i + 1][j + 1].second - circles[i + 1][j].second,

z + zStep - (z + zStep));

v2 = glm::vec3(sphereX - circles[i + 1][j + 1].first,

sphereY - circles[i + 1][j + 1].second,

sphereRadius - (z + zStep));

n = glm::cross(v1, v2);

triangles.insert(triangles.end(), {

circles[i + 1][j].first, circles[i + 1][j].second, z + zStep, n.x, n.y, n.z, figureR, figureG,

figureB,

circles[i + 1][j + 1].first, circles[i + 1][j + 1].second, z + zStep, n.x, n.y, n.z, figureR,

figureG, figureB,

sphereX, sphereY, sphereRadius, n.x, n.y, n.z, figureR, figureG, figureB,

});

}

}

z += zStep;

}

std::vector<unsigned> indices(triangles.size() / 9);

for (size\_t i = 0; i < indices.size(); i++) {

indices[i] = i;

}

return {triangles, indices};

}

std::pair<std::vector<float>, std::vector<unsigned>> cubeFigure() {

std::vector<float> vertices = {

1.0, 1.0, -1.0, //0

-1.0, 1.0, -1.0,

-1.0, -1.0, -1.0,

1.0, -1.0, -1.0,

1.0, 1.0, 1.0, //4

-1.0, 1.0, 1.0,

-1.0, -1.0, 1.0,

1.0, -1.0, 1.0,

};

std::vector<unsigned> indices = {

0, 1, 4,

1, 5, 4,

1, 2, 5,

2, 6, 5,

2, 3, 6,

3, 7, 6,

0, 7, 3,

0, 4, 7,

4, 5, 6,

6, 7, 4,

0, 2, 1,

0, 3, 2,

};

return {vertices, indices};

}

**math\_help.cpp**

s//

// Created by zero on 18/10/2020.

//

#pragma once

#include <vector>

#include <cmath>

#include <algorithm>

namespace math {

const double pi = acos(-1.0);

/// Evenly spaced numbers over a specified interval [t1, t2]

///

/// \param t1 Left border

/// \param t2 Right border

/// \param count Number of elements (including borders)

/// \return vector of evenly spaced numbers

std::vector<float> linspace(float t1, float t2, unsigned count = 100) {

if (count == 0) {

return {};

}

if (count == 1) {

return {t1};

}

if (count == 2) {

return {t1, t2};

}

bool reverse = false;

if (t1 > t2) {

std::swap(t1, t2);

reverse = true;

}

std::vector<float> res(0);

float distance = t2 - t1;

float step = distance / static\_cast<float>(count - 1);

float cur = t1;

while (cur <= t2 || std::abs(cur - t2) < 1e-5) {

res.push\_back(cur);

cur += step;

}

if (res.size() < count) {

res.push\_back(t2);

}

if (reverse) {

std::reverse(res.begin(), res.end());

}

return res;

}

/// Constructs 2d regular polygon

///

/// \param n Number of vertices

/// \param r Radius

/// \param x0, y0 Center

/// \return Vector of regular polygon points

std::vector<std::pair<float, float>> regularPolygon(size\_t n, float r, float x0, float y0) {

std::vector<std::pair<float, float>> res(n);

for (size\_t i = 0; i < n; i++) {

res[i] = {x0 + r \* cos(2 \* pi \* i / n), y0 + r \* sin(2 \* pi \* i / n)};

}

return res;

}

} //namespace math

1. **Выводы**

Выполнив данную лабораторную работу, я закрепил знания по использованию OpenGl и запуску графического интерфейса, перерисовыванию его в зависимости от изменения окна и отрисовки многогранника. Также познакомился с отображения интерактивной 3D графики. Также научился делать шейдерную анимацию эффектов.

**ЛИТЕРАТУРА**

1. Введение в 3D: основы [Электронный ресурс]URL: <https://habr.com/ru/post/494810/> (Дата обращения: 10.11.2021).

2. Туториал по OpenGL [Электронный ресурс]URL: <http://www.opengl-tutorial.org> (Дата обращения: 10.11.2021).