#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

#include <limits>

#include <string>

// OpenGL и связанные библиотеки

#include <GL/glew.h>

#include <GLFW/glfw3.h>

// GLM для работы с матрицами и векторами

#include <glm/glm.hpp>

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

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

// Eigen для выполнения PCA

#include <Eigen/Dense>

// Для использования M\_PI

#define \_USE\_MATH\_DEFINES

#include <cmath>

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

// Logging setup

enum LogLevel {

DEBUG,

INFO,

ERROR

};

LogLevel currentLogLevel = DEBUG;

std::ofstream logFile;

void log(LogLevel level, const std::string& message) {

if (level >= currentLogLevel) {

switch (level) {

case DEBUG:

logFile << "[DEBUG]: " << message << std::endl;

break;

case INFO:

logFile << "[INFO]: " << message << std::endl;

break;

case ERROR:

logFile << "[ERROR]: " << message << std::endl;

break;

}

}

}

// Data structures

struct Section {

int Z = 0;

std::vector<std::pair<int, int>> points;

};

struct Header {

uint16\_t version;

int N;

uint16\_t pointsPerSection;

uint16\_t logIndex;

double datetime;

uint16\_t frontDiameter;

uint16\_t middleDiameter;

uint16\_t backDiameter;

uint16\_t tipDiameter;

uint16\_t logLength;

uint8\_t curvature;

int16\_t curvatureDirection;

int16\_t taper;

int16\_t taperBase;

float physicalVolume;

uint16\_t flags;

float encoderPulsePrice;

};

// Global variables for control

float rotationX = 0.0f;

float rotationY = 0.0f;

float scale = 1.0f;

float positionX = 0.0f;

Header header;

std::vector<Section> sections;

std::vector<Section> cylinderSections; // Sections for the cylinder

GLuint VAO, VBO;

GLuint cylinderVAO, cylinderVBO;

double lastX = 400, lastY = 300;

bool firstMouse = true;

bool mousePressed = false;

float sensitivity = 0.005f;

bool cylinderCreated = false;

// Определяем numSegments глобально

const int numSegments = 36; // Number of points around the circle

// Function prototypes

Header readHeader(std::ifstream& file);

std::vector<Section> readSections(std::ifstream& file, int N, std::streampos fileSize);

void processInput(GLFWwindow\* window);

void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods);

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos);

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset);

GLuint compileShader(GLenum type, const char\* source);

size\_t createBuffersFromSections();

void createCylinderModel(const Header& originalHeader);

void computeSectionCenters();

void computeModelAxis(Eigen::Vector3f& axisDirection);

void alignModelToAxis(const Eigen::Vector3f& axisDirection);

float computeMinRadius();

void createCylinderSections(float minRadius);

void saveCylinderModel(const std::string& filename, const Header& originalHeader);

// Shader sources

const char\* vertexShaderSource = R"(

#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);

}

)";

const char\* fragmentShaderSource = R"(

#version 330 core

uniform vec4 ourColor;

out vec4 FragColor;

void main() {

FragColor = ourColor;

}

)";

// Data structures for centered sections

struct CenteredSection {

int Z = 0;

float centerX = 0.0f;

float centerY = 0.0f;

std::vector<std::pair<float, float>> centeredPoints;

};

std::vector<CenteredSection> centeredSections;

// Function to read header

Header readHeader(std::ifstream& file) {

Header header;

file.read(reinterpret\_cast<char\*>(&header.version), sizeof(header.version));

file.read(reinterpret\_cast<char\*>(&header.N), sizeof(header.N));

file.read(reinterpret\_cast<char\*>(&header.pointsPerSection), sizeof(header.pointsPerSection));

file.read(reinterpret\_cast<char\*>(&header.logIndex), sizeof(header.logIndex));

file.read(reinterpret\_cast<char\*>(&header.datetime), sizeof(header.datetime));

file.read(reinterpret\_cast<char\*>(&header.frontDiameter), sizeof(header.frontDiameter));

file.read(reinterpret\_cast<char\*>(&header.middleDiameter), sizeof(header.middleDiameter));

file.read(reinterpret\_cast<char\*>(&header.backDiameter), sizeof(header.backDiameter));

file.read(reinterpret\_cast<char\*>(&header.tipDiameter), sizeof(header.tipDiameter));

file.read(reinterpret\_cast<char\*>(&header.logLength), sizeof(header.logLength));

file.read(reinterpret\_cast<char\*>(&header.curvature), sizeof(header.curvature));

file.seekg(1, std::ios::cur);

file.read(reinterpret\_cast<char\*>(&header.curvatureDirection), sizeof(header.curvatureDirection));

file.read(reinterpret\_cast<char\*>(&header.taper), sizeof(header.taper));

file.read(reinterpret\_cast<char\*>(&header.taperBase), sizeof(header.taperBase));

file.read(reinterpret\_cast<char\*>(&header.physicalVolume), sizeof(header.physicalVolume));

file.read(reinterpret\_cast<char\*>(&header.flags), sizeof(header.flags));

file.read(reinterpret\_cast<char\*>(&header.encoderPulsePrice), sizeof(header.encoderPulsePrice));

file.seekg(83, std::ios::cur);

return header;

}

// Function to read sections

std::vector<Section> readSections(std::ifstream& file, int N, std::streampos fileSize) {

std::vector<Section> sections;

for (int i = 0; i < N; ++i) {

Section section;

file.read(reinterpret\_cast<char\*>(&section.Z), sizeof(section.Z));

log(INFO, "После чтения Z для сечения " + std::to\_string(i) + ": позиция в файле = " + std::to\_string(file.tellg()));

log(INFO, "Сечение " + std::to\_string(i) + ": Z = " + std::to\_string(section.Z));

uint16\_t M;

file.read(reinterpret\_cast<char\*>(&M), sizeof(M));

log(INFO, "После чтения количества точек для сечения " + std::to\_string(i) + ": позиция в файле = " + std::to\_string(file.tellg()));

log(INFO, "Количество точек в сечении " + std::to\_string(i) + ": " + std::to\_string(M));

for (int j = 0; j < M; ++j) {

int16\_t x, y;

file.read(reinterpret\_cast<char\*>(&x), sizeof(x));

file.read(reinterpret\_cast<char\*>(&y), sizeof(y));

section.points.emplace\_back(x, y);

log(INFO, "Точка " + std::to\_string(j) + ": X = " + std::to\_string(x) + ", Y = " + std::to\_string(y));

log(INFO, "После чтения точки " + std::to\_string(j) + " для сечения " + std::to\_string(i) + ": позиция в файле = " + std::to\_string(file.tellg()));

}

sections.push\_back(section);

if (file.tellg() >= fileSize) {

log(ERROR, "Файл прочитан до конца на сечении " + std::to\_string(i));

break;

}

}

return sections;

}

// Input processing

void processInput(GLFWwindow\* window) {

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

positionX -= 1.0f;

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

positionX += 1.0f;

// Check for 'C' key to create cylinder

if (glfwGetKey(window, GLFW\_KEY\_C) == GLFW\_PRESS && !cylinderCreated) {

createCylinderModel(header); // Передаем оригинальный заголовок

cylinderCreated = true;

}

}

// Mouse button callback

void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods) {

if (button == GLFW\_MOUSE\_BUTTON\_LEFT) {

if (action == GLFW\_PRESS) {

mousePressed = true;

firstMouse = true;

}

else if (action == GLFW\_RELEASE) {

mousePressed = false;

}

}

}

// Mouse movement callback

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos) {

if (mousePressed) {

if (firstMouse) {

lastX = xpos;

lastY = ypos;

firstMouse = false;

}

float xoffset = static\_cast<float>(xpos - lastX);

float yoffset = static\_cast<float>(ypos - lastY);

lastX = xpos;

lastY = ypos;

rotationY += xoffset \* sensitivity;

rotationX += yoffset \* sensitivity;

}

}

// Scroll callback for zooming

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset) {

scale += static\_cast<float>(yoffset) \* 0.1f;

if (scale < 0.1f)

scale = 0.1f;

}

// Shader compilation

GLuint compileShader(GLenum type, const char\* source) {

GLuint shader = glCreateShader(type);

glShaderSource(shader, 1, &source, nullptr);

glCompileShader(shader);

return shader;

}

// Функция для нахождения лучшей прямой через центры сечений

void computeBestFitLine(Eigen::Vector3f& pointOnLine, Eigen::Vector3f& lineDirection) {

size\_t N = centeredSections.size();

Eigen::MatrixXf positions(N, 3);

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

positions(i, 0) = centeredSections[i].centerX;

positions(i, 1) = centeredSections[i].centerY;

positions(i, 2) = static\_cast<float>(centeredSections[i].Z) / 10.0f;

}

// Вычисляем среднее положение

pointOnLine = positions.colwise().mean();

// Вычитаем среднее

positions.rowwise() -= pointOnLine.transpose();

// Выполняем SVD

Eigen::JacobiSVD<Eigen::MatrixXf> svd(positions, Eigen::ComputeThinU | Eigen::ComputeThinV);

// Направление линии соответствует первому сингулярному вектору

lineDirection = svd.matrixV().col(0);

}

// Create buffers from sections and add end caps

size\_t createBuffersFromSections() {

std::vector<float> vertices;

// Generate side triangles

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

const auto& section1 = sections[i];

const auto& section2 = sections[i + 1];

size\_t M1 = section1.points.size();

size\_t M2 = section2.points.size();

size\_t M = std::min(M1, M2);

if (M < 3) continue;

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x0 = static\_cast<float>(section1.points[j].first) / 10.0f;

float y0 = static\_cast<float>(section1.points[j].second) / 10.0f;

float z0 = static\_cast<float>(section1.Z) / 10.0f;

float x1 = static\_cast<float>(section1.points[nextJ].first) / 10.0f;

float y1 = static\_cast<float>(section1.points[nextJ].second) / 10.0f;

float z1 = static\_cast<float>(section1.Z) / 10.0f;

float x2 = static\_cast<float>(section2.points[j].first) / 10.0f;

float y2 = static\_cast<float>(section2.points[j].second) / 10.0f;

float z2 = static\_cast<float>(section2.Z) / 10.0f;

float x3 = static\_cast<float>(section2.points[nextJ].first) / 10.0f;

float y3 = static\_cast<float>(section2.points[nextJ].second) / 10.0f;

float z3 = static\_cast<float>(section2.Z) / 10.0f;

// First triangle

vertices.push\_back(x0); vertices.push\_back(y0); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z1);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z2);

// Second triangle

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z1);

vertices.push\_back(x3); vertices.push\_back(y3); vertices.push\_back(z3);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z2);

}

}

// Add end cap for the first section

{

const auto& section = sections.front();

size\_t M = section.points.size();

if (M >= 3) {

float z0 = static\_cast<float>(section.Z) / 10.0f;

float centerX = 0.0f;

float centerY = 0.0f;

for (const auto& p : section.points) {

centerX += static\_cast<float>(p.first) / 10.0f;

centerY += static\_cast<float>(p.second) / 10.0f;

}

centerX /= M;

centerY /= M;

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x1 = static\_cast<float>(section.points[j].first) / 10.0f;

float y1 = static\_cast<float>(section.points[j].second) / 10.0f;

float x2 = static\_cast<float>(section.points[nextJ].first) / 10.0f;

float y2 = static\_cast<float>(section.points[nextJ].second) / 10.0f;

// Triangle

vertices.push\_back(centerX); vertices.push\_back(centerY); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z0);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z0);

}

}

}

// Add end cap for the last section

{

const auto& section = sections.back();

size\_t M = section.points.size();

if (M >= 3) {

float z0 = static\_cast<float>(section.Z) / 10.0f;

float centerX = 0.0f;

float centerY = 0.0f;

for (const auto& p : section.points) {

centerX += static\_cast<float>(p.first) / 10.0f;

centerY += static\_cast<float>(p.second) / 10.0f;

}

centerX /= M;

centerY /= M;

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x1 = static\_cast<float>(section.points[j].first) / 10.0f;

float y1 = static\_cast<float>(section.points[j].second) / 10.0f;

float x2 = static\_cast<float>(section.points[nextJ].first) / 10.0f;

float y2 = static\_cast<float>(section.points[nextJ].second) / 10.0f;

// Triangle

vertices.push\_back(centerX); vertices.push\_back(centerY); vertices.push\_back(z0);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z0);

}

}

}

// Upload data to buffers

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

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

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

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

log(INFO, "Количество вершин для отрисовки: " + std::to\_string(vertices.size()));

return vertices.size();

}

// Function to compute centers of sections

void computeSectionCenters() {

centeredSections.clear();

for (const auto& section : sections) {

CenteredSection centeredSection;

centeredSection.Z = section.Z;

size\_t M = section.points.size();

float sumX = 0.0f;

float sumY = 0.0f;

for (const auto& p : section.points) {

float x = static\_cast<float>(p.first) / 10.0f;

float y = static\_cast<float>(p.second) / 10.0f;

sumX += x;

sumY += y;

}

centeredSection.centerX = sumX / M;

centeredSection.centerY = sumY / M;

for (const auto& p : section.points) {

float x = static\_cast<float>(p.first) / 10.0f - centeredSection.centerX;

float y = static\_cast<float>(p.second) / 10.0f - centeredSection.centerY;

centeredSection.centeredPoints.emplace\_back(x, y);

}

centeredSections.push\_back(centeredSection);

}

}

// Function to compute model axis using PCA

Eigen::Matrix3f computeModelAxis(Eigen::Vector3f& axisDirection, Eigen::Vector3f& meanPosition) {

// Собираем центры сечений в матрицу

Eigen::MatrixXf data(centeredSections.size(), 3);

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

data(i, 0) = centeredSections[i].centerX;

data(i, 1) = centeredSections[i].centerY;

data(i, 2) = static\_cast<float>(centeredSections[i].Z) / 10.0f;

}

// Вычитаем среднее

meanPosition = data.colwise().mean();

data.rowwise() -= meanPosition.transpose();

// Вычисляем матрицу ковариации

Eigen::Matrix3f cov = data.transpose() \* data;

// Вычисляем собственные значения и векторы

Eigen::SelfAdjointEigenSolver<Eigen::Matrix3f> eig(cov);

axisDirection = eig.eigenvectors().col(2).normalized();

// Определяем вектор поворота

Eigen::Vector3f zAxis(0.0f, 0.0f, 1.0f);

Eigen::Vector3f rotationAxis = axisDirection.cross(zAxis);

float angle = std::acos(axisDirection.dot(zAxis));

// Создаем матрицу поворота

Eigen::Matrix3f rotationMatrix = Eigen::Matrix3f::Identity();

if (rotationAxis.norm() > 1e-6) {

rotationAxis.normalize();

Eigen::AngleAxisf rotation(angle, rotationAxis);

rotationMatrix = rotation.toRotationMatrix();

}

return rotationMatrix;

}

// Function to align model to principal axis

void alignModelToAxis(const Eigen::Matrix3f& rotationMatrix, const Eigen::Vector3f& meanPosition) {

for (auto& section : centeredSections) {

// Поворот центра сечения

Eigen::Vector3f center(section.centerX, section.centerY, static\_cast<float>(section.Z) / 10.0f);

center -= meanPosition;

center = rotationMatrix \* center;

section.centerX = center.x();

section.centerY = center.y();

section.Z = static\_cast<int>((center.z()) \* 10.0f);

// Поворот точек сечения

for (auto& point : section.centeredPoints) {

Eigen::Vector3f p(point.first, point.second, 0.0f);

p = rotationMatrix \* p;

point.first = p.x();

point.second = p.y();

}

}

}

void applyInverseTransformationToCylinder(const Eigen::Matrix3f& rotationMatrix, const Eigen::Vector3f& meanPosition) {

Eigen::Matrix3f inverseRotation = rotationMatrix.transpose(); // Обратная матрица поворота

for (auto& section : cylinderSections) {

// Корректируем Z

float z = static\_cast<float>(section.Z) / 10.0f;

Eigen::Vector3f center(0.0f, 0.0f, z);

center = inverseRotation \* center + meanPosition;

section.Z = static\_cast<int>(center.z() \* 10.0f);

// Корректируем точки

for (auto& point : section.points) {

float x = static\_cast<float>(point.first) / 10.0f;

float y = static\_cast<float>(point.second) / 10.0f;

Eigen::Vector3f p(x, y, z);

p = inverseRotation \* p + meanPosition;

point.first = static\_cast<int16\_t>(p.x() \* 10.0f);

point.second = static\_cast<int16\_t>(p.y() \* 10.0f);

}

}

}

// Function to compute minimum radius

float computeMinRadius() {

float minRadius = std::numeric\_limits<float>::max();

for (const auto& section : centeredSections) {

for (const auto& point : section.centeredPoints) {

float distance = std::sqrt(point.first \* point.first + point.second \* point.second);

if (distance < minRadius) {

minRadius = distance;

}

}

}

return minRadius;

}

// Function to create cylinder sections

void createCylinderSections(float minRadius) {

cylinderSections.clear();

for (const auto& section : centeredSections) {

Section cylSection;

cylSection.Z = section.Z;

for (int i = 0; i < numSegments; ++i) {

float angle = 2.0f \* static\_cast<float>(M\_PI) \* i / numSegments;

float x = minRadius \* std::cos(angle);

float y = minRadius \* std::sin(angle);

// Смещаем к центру сечения

int16\_t xi = static\_cast<int16\_t>((x + section.centerX) \* 10.0f);

int16\_t yi = static\_cast<int16\_t>((y + section.centerY) \* 10.0f);

cylSection.points.emplace\_back(xi, yi);

}

cylinderSections.push\_back(cylSection);

}

}

// Function to save cylinder model

void saveCylinderModel(const std::string& filename, const Header& originalHeader) {

std::ofstream outFile(filename, std::ios::binary);

if (outFile.is\_open()) {

// Copy the original header and adjust necessary fields

Header cylHeader = originalHeader;

cylHeader.N = static\_cast<int>(cylinderSections.size());

// Записываем заголовок

outFile.write(reinterpret\_cast<const char\*>(&cylHeader), sizeof(cylHeader));

// Write sections

for (const auto& section : cylinderSections) {

outFile.write(reinterpret\_cast<const char\*>(&section.Z), sizeof(section.Z));

uint16\_t M = static\_cast<uint16\_t>(section.points.size());

outFile.write(reinterpret\_cast<const char\*>(&M), sizeof(M));

for (const auto& point : section.points) {

int16\_t x = point.first;

int16\_t y = point.second;

outFile.write(reinterpret\_cast<const char\*>(&x), sizeof(x));

outFile.write(reinterpret\_cast<const char\*>(&y), sizeof(y));

}

}

outFile.close();

log(INFO, "Цилиндр сохранен в файл " + filename);

}

else {

log(ERROR, "Не удалось сохранить цилиндр в файл " + filename);

}

}

// Function to create the maximum cylinder model

void createCylinderModel(const Header& originalHeader) {

// Вычисляем центры сечений

computeSectionCenters();

// Находим лучшую прямую через центры сечений

Eigen::Vector3f pointOnLine, lineDirection;

computeBestFitLine(pointOnLine, lineDirection);

// Нормализуем направление линии

lineDirection.normalize();

// Вычисляем минимальный радиус

float minRadius = computeMinRadius();

log(INFO, "Минимальный радиус: " + std::to\_string(minRadius));

// Создаем цилиндрические сечения вдоль прямой оси

cylinderSections.clear();

for (const auto& section : centeredSections) {

// Проецируем центр сечения на линию оси

Eigen::Vector3f center(section.centerX, section.centerY, static\_cast<float>(section.Z) / 10.0f);

Eigen::Vector3f toCenter = center - pointOnLine;

float t = toCenter.dot(lineDirection); // Положение вдоль оси

Eigen::Vector3f projectedCenter = pointOnLine + t \* lineDirection;

Section cylSection;

cylSection.Z = static\_cast<int>(projectedCenter.z() \* 10.0f);

for (int i = 0; i < numSegments; ++i) {

float angle = 2.0f \* static\_cast<float>(M\_PI) \* i / numSegments;

float x = minRadius \* std::cos(angle);

float y = minRadius \* std::sin(angle);

// Смещаем к центру сечения на оси

Eigen::Vector3f point = projectedCenter + Eigen::Vector3f(x, y, 0.0f);

int16\_t xi = static\_cast<int16\_t>(point.x() \* 10.0f);

int16\_t yi = static\_cast<int16\_t>(point.y() \* 10.0f);

cylSection.points.emplace\_back(xi, yi);

}

cylinderSections.push\_back(cylSection);

}

// Сохраняем цилиндр в файл

saveCylinderModel("cylinder\_model.lprf", originalHeader);

// Генерируем OpenGL буферы для цилиндра

// Создаем вершины для OpenGL

std::vector<float> vertices;

// Generate side triangles

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

const auto& section1 = cylinderSections[i];

const auto& section2 = cylinderSections[i + 1];

size\_t M = section1.points.size();

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x0 = static\_cast<float>(section1.points[j].first) / 10.0f;

float y0 = static\_cast<float>(section1.points[j].second) / 10.0f;

float z0 = static\_cast<float>(section1.Z) / 10.0f;

float x1 = static\_cast<float>(section1.points[nextJ].first) / 10.0f;

float y1 = static\_cast<float>(section1.points[nextJ].second) / 10.0f;

float z1 = static\_cast<float>(section1.Z) / 10.0f;

float x2 = static\_cast<float>(section2.points[j].first) / 10.0f;

float y2 = static\_cast<float>(section2.points[j].second) / 10.0f;

float z2 = static\_cast<float>(section2.Z) / 10.0f;

float x3 = static\_cast<float>(section2.points[nextJ].first) / 10.0f;

float y3 = static\_cast<float>(section2.points[nextJ].second) / 10.0f;

float z3 = static\_cast<float>(section2.Z) / 10.0f;

// First triangle

vertices.push\_back(x0); vertices.push\_back(y0); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z1);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z2);

// Second triangle

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z1);

vertices.push\_back(x3); vertices.push\_back(y3); vertices.push\_back(z3);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z2);

}

}

// Add end caps similarly

// First end cap

{

const auto& section = cylinderSections.front();

size\_t M = section.points.size();

float z0 = static\_cast<float>(section.Z) / 10.0f;

float centerX = static\_cast<float>(section.points[0].first) / 10.0f;

float centerY = static\_cast<float>(section.points[0].second) / 10.0f;

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x1 = static\_cast<float>(section.points[j].first) / 10.0f;

float y1 = static\_cast<float>(section.points[j].second) / 10.0f;

float x2 = static\_cast<float>(section.points[nextJ].first) / 10.0f;

float y2 = static\_cast<float>(section.points[nextJ].second) / 10.0f;

// Triangle

vertices.push\_back(centerX); vertices.push\_back(centerY); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z0);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z0);

}

}

// Last end cap

{

const auto& section = cylinderSections.back();

size\_t M = section.points.size();

float z0 = static\_cast<float>(section.Z) / 10.0f;

float centerX = static\_cast<float>(section.points[0].first) / 10.0f;

float centerY = static\_cast<float>(section.points[0].second) / 10.0f;

for (size\_t j = 0; j < M; ++j) {

size\_t nextJ = (j + 1) % M;

float x1 = static\_cast<float>(section.points[j].first) / 10.0f;

float y1 = static\_cast<float>(section.points[j].second) / 10.0f;

float x2 = static\_cast<float>(section.points[nextJ].first) / 10.0f;

float y2 = static\_cast<float>(section.points[nextJ].second) / 10.0f;

// Triangle

vertices.push\_back(centerX); vertices.push\_back(centerY); vertices.push\_back(z0);

vertices.push\_back(x2); vertices.push\_back(y2); vertices.push\_back(z0);

vertices.push\_back(x1); vertices.push\_back(y1); vertices.push\_back(z0);

}

}

// Upload data to buffers

glGenVertexArrays(1, &cylinderVAO);

glGenBuffers(1, &cylinderVBO);

glBindVertexArray(cylinderVAO);

glBindBuffer(GL\_ARRAY\_BUFFER, cylinderVBO);

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

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

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

}

int main() {

system("chcp 1251");

logFile.open("log.txt");

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

std::cerr << "Не удалось открыть файл для логирования." << std::endl;

return -1;

}

log(INFO, "Запуск программы.");

std::ifstream file("model3.lprf", std::ios::binary);

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

log(ERROR, "Не удалось открыть файл.");

return -1;

}

header = readHeader(file); // Инициализируем глобальную переменную

log(INFO, "Файл версии: " + std::to\_string(header.version));

log(INFO, "Количество сечений: " + std::to\_string(header.N));

file.seekg(130, std::ios::beg);

file.seekg(0, std::ios::end);

std::streampos fileSize = file.tellg();

file.seekg(130, std::ios::beg);

log(INFO, "Размер файла: " + std::to\_string(fileSize) + " байт");

sections = readSections(file, header.N, fileSize);

file.close();

if (!glfwInit()) {

log(ERROR, "Failed to initialize GLFW");

return -1;

}

GLFWwindow\* window = glfwCreateWindow(800, 600, "3D Object Viewer", nullptr, nullptr);

if (!window) {

log(ERROR, "Failed to create GLFW window");

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

if (glewInit() != GLEW\_OK) {

log(ERROR, "Failed to initialize GLEW");

return -1;

}

glfwSetCursorPosCallback(window, mouse\_callback);

glfwSetMouseButtonCallback(window, mouse\_button\_callback);

glfwSetScrollCallback(window, scroll\_callback);

GLuint vertexShader = compileShader(GL\_VERTEX\_SHADER, vertexShaderSource);

GLuint fragmentShader = compileShader(GL\_FRAGMENT\_SHADER, fragmentShaderSource);

GLuint shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

glUseProgram(shaderProgram);

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

size\_t BZ = createBuffersFromSections();

glEnable(GL\_DEPTH\_TEST);

while (!glfwWindowShouldClose(window)) {

processInput(window);

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

// Set wireframe mode

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

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

model = glm::translate(model, glm::vec3(positionX, 0.0f, 0.0f));

model = glm::rotate(model, rotationX, glm::vec3(1.0f, 0.0f, 0.0f));

model = glm::rotate(model, rotationY, glm::vec3(0.0f, 1.0f, 0.0f));

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

glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -500.0f));

glm::mat4 projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 1000.0f);

GLuint modelLoc = glGetUniformLocation(shaderProgram, "model");

GLuint viewLoc = glGetUniformLocation(shaderProgram, "view");

GLuint projectionLoc = glGetUniformLocation(shaderProgram, "projection");

GLuint colorLoc = glGetUniformLocation(shaderProgram, "ourColor");

glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));

glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));

glUniformMatrix4fv(projectionLoc, 1, GL\_FALSE, glm::value\_ptr(projection));

// Render original model

glUniform4f(colorLoc, 1.0f, 1.0f, 1.0f, 1.0f); // White color

glBindVertexArray(VAO);

glDrawArrays(GL\_TRIANGLES, 0, static\_cast<GLsizei>(BZ / 3));

glBindVertexArray(0);

// Render cylinder model if created

if (cylinderCreated) {

glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f); // Red color

glBindVertexArray(cylinderVAO);

// Исправлено: добавлен расчет количества вершин

size\_t numVertices = (cylinderSections.size() - 1) \* numSegments \* 6 + numSegments \* 6 \* 2;

glDrawArrays(GL\_TRIANGLES, 0, static\_cast<GLsizei>(numVertices));

glBindVertexArray(0);

}

glfwSwapBuffers(window);

glfwPollEvents();

}

glfwTerminate();

logFile.close();

return 0;

}