#include <iostream>

#include <fstream>

#include <vector>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <glm/glm.hpp>

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

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

#include <string>

#include <limits> // Для работы с границами min и max

enum LogLevel {

DEBUG,

INFO,

ERROR

};

LogLevel currentLogLevel = DEBUG; // Задаем уровень логирования

std::ofstream logFile;

// Функция логирования

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;

}

}

}

void exampleFunction() {

log(INFO, "Начало функции.");

log(DEBUG, "Подробное сообщение для отладки.");

log(ERROR, "Сообщение об ошибке.");

log(INFO, "Конец функции.");

}

// Структура для хранения сечений

struct Section {

int Z = 0; // Координата Z для всего сечения (расстояние от начала бревна)

std::vector<std::pair<int, int>> points; // Координаты X и Y всех точек в сечении

};

// Структура для заголовка файла

struct Header {

uint16\_t version; // Версия файла

int N; // Количество сечений

uint16\_t pointsPerSection; // Количество точек в сечении

uint16\_t logIndex; // Индекс бревна

double datetime; // Дата/время измерения

uint16\_t frontDiameter; // Диаметр переднего торца

uint16\_t middleDiameter; // Диаметр средней части

uint16\_t backDiameter; // Диаметр заднего торца

uint16\_t tipDiameter; // Диаметр вершинки

uint16\_t logLength; // Длина бревна

uint8\_t curvature; // Кривизна

int16\_t curvatureDirection; // Направление кривизны

int16\_t taper; // Сбег

int16\_t taperBase; // Сбег комля

float physicalVolume; // Физический объем

uint16\_t flags; // Флаги

float encoderPulsePrice; // Цена импульса энкодера

};

// Глобальные переменные для управления

float rotationX = 0.0f;

float rotationY = 0.0f;

float scale = 1.0f;

float cameraZ = -500.0f; // Позиция камеры по оси Z

std::vector<Section> sections; // Все сечения

GLuint VAO, VBO; // Буферы для хранения точек

// Функция для чтения заголовка из файла

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); // Пропуск резервного байта

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));

// Пропускаем резервные байты с 0 до 83

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

return header;

}

// Функция для чтения сечений из файла

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;

// Чтение координаты Z для сечения (integer)

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()));

// Логирование для проверки координаты Z

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

// Чтение количества точек (word)

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()));

// Логирование количества точек для проверки

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

// Чтение точек (X и Y в формате smallint)

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

int16\_t x, y;

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

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

section.points.emplace\_back(x, y); // Добавление точки в сечение

// Логирование точек

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

// Проверка позиции после чтения каждой точки

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); // Добавление сечения в вектор

// Проверка выхода за границы файла

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

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

break; // Выход из цикла, если файл прочитан до конца

}

}

return sections;

}

// Шейдеры

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

out vec4 FragColor;

void main() {

FragColor = vec4(1.0, 1.0, 1.0, 1.0);

}

)";

// Обработка ввода для вращения, перемещения и масштабирования

void processInput(GLFWwindow\* window) {

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

rotationX += 0.005f;

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

rotationX -= 0.005f;

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

rotationY += 0.005f;

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

rotationY -= 0.005f;

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

scale += 0.005f;

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

scale -= 0.005f;

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

cameraZ += 2.0f; // Двигаем камеру вперед

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

cameraZ -= 2.0f; // Двигаем камеру назад

}

// Функция для компиляции шейдеров

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

GLuint shader = glCreateShader(type);

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

glCompileShader(shader);

return shader;

}

// Функция для создания буферов на основе сечений (линии)

int createBuffersFromSections() {

std::vector<float> vertices;

// Преобразуем точки в формат вершин для OpenGL (3D с Z как координатой глубины)

for (const auto& section : sections) {

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

if (M == 0) continue; // Пропускаем пустые сечения

// Соединение точек внутри каждого сечения в виде окружности

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

// X, Y — координаты точек, Z — фиксированная координата для всего сечения

vertices.push\_back(static\_cast<float>(section.points[i].first) / 10.0f); // X

vertices.push\_back(static\_cast<float>(section.points[i].second) / 10.0f); // Y

vertices.push\_back(static\_cast<float>(section.Z) / 10.0f); // Z

// Соединяем каждую точку с следующей (или первой для замыкания окружности)

if (i + 1 < M) {

vertices.push\_back(static\_cast<float>(section.points[i + 1].first) / 10.0f);

vertices.push\_back(static\_cast<float>(section.points[i + 1].second) / 10.0f);

vertices.push\_back(static\_cast<float>(section.Z) / 10.0f);

}

else {

// Замыкание окружности: последняя точка соединяется с первой

vertices.push\_back(static\_cast<float>(section.points[0].first) / 10.0f);

vertices.push\_back(static\_cast<float>(section.points[0].second) / 10.0f);

vertices.push\_back(static\_cast<float>(section.Z) / 10.0f);

}

}

}

// Соединяем соответствующие точки между сечениями

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

const auto& section1 = sections[i];

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

size\_t M = std::min(section1.points.size(), section2.points.size());

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

// Соединяем точки между соседними сечениями

vertices.push\_back(static\_cast<float>(section1.points[j].first) / 10.0f);

vertices.push\_back(static\_cast<float>(section1.points[j].second) / 10.0f);

vertices.push\_back(static\_cast<float>(section1.Z) / 10.0f);

vertices.push\_back(static\_cast<float>(section2.points[j].first) / 10.0f);

vertices.push\_back(static\_cast<float>(section2.points[j].second) / 10.0f);

vertices.push\_back(static\_cast<float>(section2.Z) / 10.0f);

}

}

// Загружаем данные в буфер

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();

}

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;

}

// Пример вызова логирования

log(INFO, "Запуск программы.");

// Открытие бинарного файла

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

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

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

return -1;

}

// Чтение заголовка

Header header = readHeader(file);

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

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

// Переход к 130-му байту после заголовка

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

// Проверка размера файла

file.seekg(0, std::ios::end); // Переход в конец файла, чтобы узнать его размер

std::streampos fileSize = file.tellg(); // Получение размера файла

file.seekg(130, std::ios::beg); // Переход обратно на 130-й байт

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

// Чтение данных сечений

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

file.close();

// Инициализация GLFW

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);

// Инициализация GLEW

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

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

return -1;

}

// Компиляция шейдеров

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

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

// Линковка шейдерной программы

GLuint shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

glUseProgram(shaderProgram);

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

// Создание буферов на основе считанных данных

int BZ = createBuffersFromSections();

// Включаем тест глубины для корректного отображения 3D объектов

glEnable(GL\_DEPTH\_TEST);

// Основной цикл

while (!glfwWindowShouldClose(window)) {

processInput(window);

// Очистка экрана

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

// Матрицы

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

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, cameraZ)); // Позиция камеры по оси Z

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

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

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

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

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));

// Отрисовка линий

glBindVertexArray(VAO);

glDrawArrays(GL\_LINES, 0, BZ / 3); // так как у нас 3 координаты на вершину

glBindVertexArray(0);

// Обновляем экран

glfwSwapBuffers(window);

glfwPollEvents();

}

glfwTerminate();

// Закрытие файла после завершения

logFile.close();

return 0;

}