МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

РГЗ

по дисциплине: Компьютерная графика
тема: «Знакомство с библиотекой OpenGL»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверил: Осипов Олег Васильевич

Белгород 2024 г.

Оглавление

1 Формулировка задачи

Разработать программное обеспечение использующее библиотеку OpenGL, пайплайн шейдеров, включающий использование точечных источников света и текстур из изображений.

В качестве задания была выбрана программа, визуализирующая аудио.

2 Вывод необходимых формул, выбор и запись расчётных методов и алгоритмов.

Для разбиения звуковой дорожки и извлечение амплитуды каждой из частоты будем использовать алгоритм быстрого преобразования Фурье и библиотеку FFTW. В результате получаем массив комплексных чисел, амплитуда для частоты определяется как: $A_i = \sqrt{(Re_i^2 + Im_i^2)}$. Частота для i элемента в получившемся массиве определяется как: $F = \frac{i \cdot F_s}{N}$, где F_s - частота дискретизации исходной дорожки, а N - количество переданных данных для алгоритма. Продуктивной часть полученного массива будет только первая половина, так как такова специфика алгоритма быстрого преобразования Фурье, объясняющаяся симметрией.

Природа человеческого уха такова, что линейное отображение амплитуд частот не будет отображать субъективное ощущение звука. Дело в том, что каждая последующая нота высчитывается не как арифметическая прогрессия, а как геометрическая. Следовательно для корректного субъективного отображения амплитуд необходимо группировать частоты в соответствие с экспоненциальной функцией. Определим минимальную и максимальную частоту. Человеческое ухо эффективно способно различать от F_{min} =20 Гц до F_{max} =15000 Гц, следовательно они и будут границами распознавания. Получим систему уравнений:

$$\begin{cases} a^0 + b = F_{min} \\ a^{\frac{N}{2}-1} + b = F_{max} \end{cases}$$

$$\begin{cases} b = F_{min} - 1 \\ a = \sqrt[M]{F_{max} - b} \end{cases}$$

Где M - количество плиток. Получили уравнение, с помощью которого сможем группировать частоты и находить среднюю амплитуду в пределах группы. Для загрузки и получения частот, с которым будет работать FFTW, используем библиотеку adamstark/AudioFile. Также введём дополнительную настройку приложения через аргументы. Для парсинга аргументов приложения будем использовать Taywee/args.

В вершинном шейдере будем определять освещённость точки при помощи модели освещённости по Фонгу. Фоновый компонент можно получить умножением интенсивности света L_a на коэффициент рассеивания поверхности K_a.
$I_a = L_a K_a$
Рассеянный компонент отражает освещение шероховатых поверхностей, где K_d - коэффициент отражения, s - направление света, n - направление нормали:
$I_d = L_d K_d (s \cdot n)$

r - направление преимущественного распространения отражённого света, v - направление наблюдателя. I_s - блик:
$r = -s + 2(s \cdot n)n$
$I_s = L_s K_s (r \cdot v)^f$
f = от 1 до 200.
Итоговое освещение. Его мы будем умножать на цвет (значение из текстуры):
$I = I_a + I_d + I_s$.

3 Исходная программа.

main.cpp

```
// main.cpp : Определяет точку входа для приложения.
#include <Windows.h>
#include <MMSystem.h>
#include <commctrl.h>
#include "Painter.h"

#include "gl/gl.h"
#include "AudioFile.h"
#include <fftw3.h>
#include "args.hxx"

#include <iostream>

#if defined(_WIN64)
#pragma comment(lib, "win64_lib/opengl32.lib")
#pragma comment(lib, "win64_lib/glew32.lib")
#elif defined(_WIN32)
#pragma comment(lib, "win32_lib/opengl32.lib")
#pragma comment(lib, "win32_lib/glew32.lib")
#endif


// Каркасное Windows-приложение с использованием библиотеки OpenGL

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

HGLRC hglrc; // Контекст OpenGL

void get_command_line_args(int* argc, char*** argv)
{
    // Get the command line arguments as wchar_t strings
    wchar_t** wargv = CommandLineToArgvW(GetCommandLineW(), argc);
    if (!wargv) { *argc = 0; *argv = NULL; return; }

    // Count the number of bytes necessary to store the UTF-8 versions of those strings
    int n = 0;
    for (int i = 0; i < *argc; i++)
        n += WideCharToMultiByte(CP_UTF8, 0, wargv[i], -1, NULL, 0, NULL, NULL) + 1;
```

```
    // Allocate the argv[] array + all the UTF-8 strings
    *argv = (char**)malloc((*argc + 1) * sizeof(char*) + n);
    if (!*argv) { *argc = 0; return; }

    // Convert all wargv[] --> argv[]
    char* arg = (char*)&((*argv)[*argc + 1]);
    for (int i = 0; i < *argc; i++)
    {
        (*argv)[i] = arg;
        arg += WideCharToMultiByte(CP_UTF8, 0, wargv[i], -1, arg, n, NULL, NULL) + 1;
    }
    (*argv)[*argc] = NULL;
}

// Создание консоли для вывода сообщений об ошибках
void CreateLogConsole(void)
{
    // Создадим консоль и перенаправим в неё стандартный поток ошибок
    AllocConsole();

    // Меняем кодовую страницу на 1251 для поддержки кириллицы
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);

    // Устанавливаем шрифт консоли на Consolas или другой шрифт, поддерживающий кириллицу
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    // Меняем цвет текста на красный
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_INTENSITY);

    FILE* stream;
    freopen_s(&stream, "CONOUT$", "w", stderr);
    freopen_s(&stream, "CONOUT$", "w", stdout);
}

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInst, LPSTR lpszArgs, int nWinMode)
{
    int argc = 0;
    char **argv = NULL;
    get_command_line_args(&argc, &argv);

    args::ArgumentParser parser(
        "There Is Sound In Space! A sound visualization application made by Pakhomov Vladislav Andreevich",
        "Have fun!");
    args::HelpFlag help(parser, "help", "Display help menu", { 'h', "help" });
    args::Positional<std::string> audioPathArg(parser, "audiopath", R"(
Required. Audio path to be played in .wav format with sample rate 44100 Hz.
Potentially you can use bigger sample rate but it wasn't tested
and could affect quality of frequencies recognition.
Using audio with less frequency cuts down frequencies, so it's not recommended as well.
Use mono signal, since application can't work with stereo.
)");
    args::ValueFlag<int> maxTilesArg(parser, "maxtiles", R"(
Optional. Limiter of amount of tiles. Actual amount of tiles depends on samples amount that are
processed every step. Default = 64.
```

```
)", {'t', "tiles"});
    args::ValueFlag<double> gravityArg(parser, "gravity", R"(
Optional. This value affects how fast tiles are falling down. The bigger value the
bigger velocity of falling. Default = 0.001.
)", { 'g', "gravity" });
    args::ValueFlag<double> sensitivityArg(parser, "sensitivity", R"(
Optional. This value affects sensitivity of frequency amplitude. The bigger value the
bigger sensitivity. Default = 0.005.
)", { 's', "sensitivity" });
    args::ValueFlag<int> qualityArg(parser, "quality", R"(
Optional. Affects amount of samples being processed. Insert 1-15. Bigger value
gives better amplitude recognition results but produces delays. Default = 14.
)", { 'q', "quality" });
    try
    {
        parser.ParseCLI(argc, argv);
    }
    catch (args::Help)
    {
        std::ostringstream message;
        message << parser;
        MessageBoxA(NULL, message.str().c_str(), "Parse info", MB_ICONINFORMATION);
        return 0;
    }
    catch (args::ParseError e)
    {
        std::ostringstream message;
        message << e.what() << std::endl;
        message << parser;
        MessageBoxA(NULL, message.str().c_str(), "Parse info", MB_ICONINFORMATION);
        return 1;
    }
    catch (args::ValidationError e)
    {
        std::ostringstream message;
        message << e.what() << std::endl;
        message << parser;
        MessageBoxA(NULL, message.str().c_str(), "Parse info", MB_ICONINFORMATION);
        return 1;
    }

    std::string audioPath;
    if (audioPathArg) {
        audioPath = args::get(audioPathArg);
    } else {
        MessageBoxA(NULL, "Audio path is required. Call program with argument -h for details", "Parse info",
↪   MB_ICONERROR);

        return 1;
    }

    if (maxTilesArg) {
        auto tmp = args::get(maxTilesArg);
        if (tmp > 0) {
```

```
            MAX_TILES_AMOUNT = tmp;
            tiles_amplitudes = std::vector<double>(MAX_TILES_AMOUNT, 0);
        }
    }

    if (gravityArg) {
        auto tmp = args::get(gravityArg);
        if (tmp > 0)
            GRAVITY = tmp;
    }

    if (sensitivityArg) {
        auto tmp = args::get(sensitivityArg);
        if (tmp > 0)
            SENSITIVITY = tmp;
    }

    if (qualityArg) {
        auto tmp = args::get(qualityArg);
        if (tmp > 0 && tmp < 16)
            SAMPLE_AMOUNT = 1 << tmp;
    }

    textureTile = BMP("textures\\tile.bmp");
    textureBackground = BMP("textures\\stars.bmp");

    char szWinName[] = "Graphics Window Class"; // Имя класса окна

    HWND hWnd; // Дескриптор главного окна

    WNDCLASSA wcl; // Определитель класса окна
    wcl.hInstance = hThisInstance; // Дескриптор приложения
    wcl.lpszClassName = szWinName;// Имя класса окна
    wcl.lpfnWndProc = WindowProc; // Функция обработки сообщений
    wcl.style = 0; // Стиль по умолчанию
    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);// Иконка
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // Курсор
    wcl.lpszMenuName = NULL; // Без меню
    wcl.cbClsExtra = 0; // Без дополнительной информации
    wcl.cbWndExtra = 0;

    wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); //Белый фон

    if (!RegisterClassA(&wcl)) // Регистрируем класс окна
        return 0;

    hWnd = CreateWindowA(szWinName, // Создать окно
        "There Is Sound In Space!",
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, // Стиль окна
        CW_USEDEFAULT, // x-координата
        CW_USEDEFAULT, // y-координата
        CW_USEDEFAULT, // Ширина
        CW_USEDEFAULT, // Высота
        HWND_DESKTOP, // Без родительского окна
```

```
        NULL, // Без меню
        hThisInstance, // Дескриптор приложения
        NULL); // Без дополнительных аргументов

    ShowWindow(hWnd, nWinMode); // Показать окно

    setvbuf(stderr, NULL, _IONBF, 0); // Отключение буферизации потока ошибок stderr для того, чтобы лог-файл, в который
↪   выводится этот поток обновлялся сразу

    // Создание консоли
    // CreateLogConsole();

    // Получение контекста устройства отображения
    HDC hdc = GetDC(hWnd);

    // Установка формата пикселей
    PIXELFORMATDESCRIPTOR pfd = {};
    pfd.nSize = sizeof(pfd);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 32;
    pfd.iLayerType = PFD_MAIN_PLANE;

    int pixelFormat = ChoosePixelFormat(hdc, &pfd);

    SetPixelFormat(hdc, pixelFormat, &pfd);

    // Создание контекста OpenGL
    hglrc = wglCreateContext(hdc);

    wglMakeCurrent(hdc, hglrc);

    // Инициализация библиотеки GLEW
    glewInit();

    // Инициализация OpenGL
    InitOpenGL();

    UpdateWindow(hWnd); // Перерисовать окно

    audioFile.load(audioPath);
    PlaySoundA(audioPath.c_str(), hThisInstance, SND_FILENAME | SND_ASYNC);

    begin_time = GetTickCount();
    last_time = begin_time;

    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) // Запустить цикл обработки сообщений
    {
        TranslateMessage(&msg); // Разрешить использование клавиатуры
        DispatchMessage(&msg); // Вернуть управление операционной системе Windows
    }
```

```
    return (int) msg.wParam;

}


// Следующая функция вызывается операционной системой Windows и получает в качестве
// параметров сообщения из очереди сообщений данного приложения
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static DWORD start_time; // Начальный момент запуска программы

    switch (message)
    {
        // Обработка сообщения на создание окна
        case WM_CREATE:
        {
            // Создаем таймер, посылающий сообщения
            // функции окна примерно 30 раз в секунду
            SetTimer(hWnd, 1, 1000/170, NULL);

            start_time = GetTickCount();
            running_time = start_time;
        }
        break;


        // Обработка сообщения на перерисовку окна
        case WM_PAINT:
        {
            PAINTSTRUCT ps;

            HDC hdc = BeginPaint(hWnd, &ps);

            // Определяем ширину и высоту окна
            RECT rect = ps.rcPaint;
            GetClientRect(hWnd, &rect);

            int width = rect.right - rect.left;
            int height = rect.bottom - rect.top;

            if (height < 0) height = 0;

            // Вычислим время, которое нужно затратить для рисования одного кадра
            char repaint_time[500];
            DWORD t1 = GetTickCount();

            glViewport(0, 0, width, height); // Область вывода

            Draw(width, height);

            SwapBuffers(hdc); // Вывести содержимое буфера на экран

            EndPaint(hWnd, &ps);
```

```
        }
        break;

        case WM_MOUSEMOVE:
        {
            if (wParam == MK_LBUTTON)
            {
                // Вычислим, на сколько переместился курсор мыши между двумя событиями WM_MOUSEMOVE
                int x = LOWORD(lParam), y = HIWORD(lParam);
                int dx = x - mousePosition.x;
                int dy = y - mousePosition.y;

                // Изменим матрицу поворота в соответствии с тем, как пользователь переместил курсор мыши
                changeRotateMatrix(dx, dy);

                // Сохраним текущую позицию мыши
                mousePosition = { x, y };

                InvalidateRect(hWnd, NULL, false);
            }
        }
        break;

        case WM_LBUTTONDOWN:

            // Запоминаем координаты курсора мыши при щелчке
            mousePosition = { LOWORD(lParam), HIWORD(lParam) };

            // Перерисовать окно
            InvalidateRect(hWnd, NULL, false);
            break;

        case WM_KEYDOWN:

                        if (wParam == VK_F1)
            {
                MessageBoxA(hWnd, "Made by student of group SC-223 Pakhomov Vladislav Andreevich", "About",
↪  MB_ICONINFORMATION);
            }

            if (wParam == VK_ESCAPE)
            {
                PostQuitMessage(0);
            }
            break;

        // Обработка сообщения на изменение размера окна
        case WM_SIZE:

            // Перерисовать окно
            InvalidateRect(hWnd, NULL, false);
            break;

        case WM_TIMER:
```

```
        {
            // При срабатывании таймера пересчитаем время от запуска программы
            running_time = (GetTickCount() - start_time) / 1000.0f;

            float current_time = GetTickCount();
            float play_time_previous = last_time - begin_time;
            float play_time_current = current_time - begin_time;
            int slice_end = (play_time_current / 1000) * audioFile.getSampleRate();

            double exp_base_log = std::log(std::pow(MAX_FREQUENCY - MIN_FREQUENCY + 1, 1. / MAX_TILES_AMOUNT));

            if (slice_end > SAMPLE_AMOUNT) {
                // Create an array to hold the output (complex numbers)
                fftw_complex* output = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * SAMPLE_AMOUNT);

                // Create a plan for the FFT
                fftw_plan plan = fftw_plan_dft_r2c_1d(SAMPLE_AMOUNT, audioFile.samples[0].data() - SAMPLE_AMOUNT +
↪  slice_end, output, FFTW_ESTIMATE);

                // Execute the FFT
                fftw_execute(plan);

                std::vector<std::pair<double, int>> prepared_freqs(MAX_TILES_AMOUNT, { 0, 0 });

                for (int i = 0; i < (SAMPLE_AMOUNT / 2) - 1; i++) {
                    double current_frequency = audioFile.getSampleRate() * i / SAMPLE_AMOUNT;

                    if ((current_frequency - MIN_FREQUENCY + 1) <= 0) continue;

                    double amplitude_index = std::log(current_frequency - MIN_FREQUENCY + 1) / exp_base_log;

                    if (amplitude_index < 0 || amplitude_index >= tiles_amplitudes.size()) break;

                    double amplitude = std::sqrt(std::pow(output[i][0], 2) + std::pow(output[i][1], 2));

                    prepared_freqs[amplitude_index].first += amplitude;
                    prepared_freqs[amplitude_index].second++;
                }

                int tmp_actual_tiles_amount = 0;

                for (int i = 0; i < MAX_TILES_AMOUNT; i++) {
                    if (prepared_freqs[i].second == 0) continue;

                    tiles_amplitudes[tmp_actual_tiles_amount] = std::max(
                        std::min(1., SENSITIVITY * prepared_freqs[i].first / prepared_freqs[i].second),
                        tiles_amplitudes[tmp_actual_tiles_amount] - (play_time_current - play_time_previous) * GRAVITY);

                    tmp_actual_tiles_amount++;
                }

                actual_tiles_amount = tmp_actual_tiles_amount;

                // Clean up
```

```
                fftw_destroy_plan(plan);
                fftw_free(output);
            }

            last_time = current_time;

            InvalidateRect(hWnd, NULL, false);
            break;
        }
        case WM_DESTROY: // Завершение программы

            wglMakeCurrent(GetDC(hWnd), NULL);
            wglDeleteContext(hglrc);
            PostQuitMessage(0);
            break;

        default:
            // Все сообщения, не обрабатываемые в данной функции, направляются на обработку по умолчанию
            return DefWindowProcA(hWnd, message, wParam, lParam);
    }

    return 0;
}
```

Painter.h

```
#ifndef PAINTER_H
#define PAINTER_H

#include "Matrix.h"
#include "Vertex.h"

#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "gl/glew.h"

#include "BMP.h"
#include "AudioFile.h"


// Время от начала запуска программы
float running_time = 0;
float begin_time = 0;
float last_time = 0;
int actual_tiles_amount = 0;

int MAX_TILES_AMOUNT = 64;
double GRAVITY = 0.001;
double TILES_SPACING = 0.01;
double MAX_FREQUENCY = 15000;
double MIN_FREQUENCY = 20;
```

```
double SENSITIVITY = 0.005;
int SAMPLE_AMOUNT = 16384;

std::vector<double> tiles_amplitudes(MAX_TILES_AMOUNT, 0);

AudioFile<double> audioFile;

// Положение курсора мыши в окне
struct
{
        int x, y;

} mousePosition;


// Матрица поворота, которая изменяется при движении мыши
Matrix rotateMatrix;
BMP textureBackground;
BMP textureTile;

void changeRotateMatrix(int dx, int dy)
{
        // Умножение rotateMatrix на матрицу поворота вокруг оси y и на матрицу поворота вокруг оси x
        rotateMatrix = rotateMatrix * Matrix::RotationY(dx / 50.0f) * Matrix::RotationX(dy / 50.f);
}


// Шейдерная программа
GLuint shaderProgram;

// Функция для создания и компиляции шейдера
GLuint CreateShader(GLenum type, const char* source)
{
        // Создание шейдерной подпрограммы
        GLuint shader = glCreateShader(type);

        if (shader == 0)
        {
                char error[] = "Ошибка создания шейдера\n";
                fprintf(stderr, error);
                OutputDebugStringA(error);
        }

        // Копирование исходного текста шейдерной подпрограммы в объект шейдера
        glShaderSource(shader, 1, &source, nullptr);

        // Компиляция шейдерной пдпрограммы
        glCompileShader(shader);

        // Проверка на ошибки компиляции
        GLint success;

        glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
        if (!success)
```

```
	{
		char error[64] = "";
		if (type == GL_VERTEX_SHADER)
		{
			sprintf_s(error, "Ошибка компиляции вершинного шейдера\n");
		}
		if (type == GL_FRAGMENT_SHADER)
		{
			sprintf_s(error, "Ошибка компиляции фрагментного шейдера\n");
		}
		fprintf(stderr, error);
		OutputDebugStringA(error);

		// Определяем длину сообщения об ошибке
		GLint logLength;
		glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);

		// Вывод сообщения об ошибке компиляции
		char* errorLog = new char[logLength];
		glGetShaderInfoLog(shader, 512, nullptr, errorLog);
		fprintf(stderr, "Функция %s:\n%s", __FUNCTION__, errorLog);
		OutputDebugStringA(errorLog);
		delete[] errorLog;
	}
	return shader;
}

std::vector<char> load_vector(std::istream& in) {
	std::vector<char> result(0);
	char c;
	while (!in.eof()) {
		result.push_back(in.get());
	}
	return result;
}


// Функция для инициализации OpenGL
void InitOpenGL() {
	std::fstream vertHFile("shaders\\main.vert", std::ios::in);
	if (!vertHFile.is_open()) throw std::invalid_argument("Error: File Not Found.");

	std::vector<char> vertFileInfo;
	vertHFile.seekg(0, std::ios_base::end);
	std::streampos vertFileSize = vertHFile.tellg();
	vertFileInfo.resize(vertFileSize);
	vertHFile.seekg(0, std::ios_base::beg);
	vertHFile.read(&vertFileInfo[0], vertFileSize);

	std::fstream fragHFile("shaders\\main.frag", std::ios::in);
	if (!fragHFile.is_open()) throw std::invalid_argument("Error: File Not Found.");

	std::vector<char> fragFileInfo;
	fragHFile.seekg(0, std::ios_base::end);
```

```
        std::streampos fragFileSize = fragHFile.tellg();
        fragFileInfo.resize(fragFileSize);
        fragHFile.seekg(0, std::ios_base::beg);
        fragHFile.read(&fragFileInfo[0], fragFileSize);

        // Компиляция шейдеров
        GLuint vertexShader = CreateShader(GL_VERTEX_SHADER, vertFileInfo.data());
        GLuint fragmentShader = CreateShader(GL_FRAGMENT_SHADER, fragFileInfo.data());

        // Создание шейдерной программы
        shaderProgram = glCreateProgram();

        // Подключение двух шейдерных подпрограмм
        glAttachShader(shaderProgram, vertexShader);
        glAttachShader(shaderProgram, fragmentShader);

        // Компоновка шейдерной программы
        glLinkProgram(shaderProgram);

        // Проверить результат компоновки
        GLint status;
        glGetProgramiv(shaderProgram, GL_LINK_STATUS, &status);
        if (status == GL_FALSE)
        {
                char error[] = "Ошибка компоновки шейдерной программы\n";
                fprintf(stderr, error);
                OutputDebugStringA(error);

                // Вывод сообщения об ошибке
                GLint logLen;
                glGetProgramiv(shaderProgram, GL_INFO_LOG_LENGTH, &logLen);
                if (logLen > 0)
                {
                        char* errorLog = new char[logLen];
                        GLsizei written;
                        glGetProgramInfoLog(shaderProgram, logLen, &written, errorLog);
                        fprintf(stderr, "Функция %s:\n%s", __FUNCTION__, errorLog);
                        OutputDebugStringA(errorLog);
                        delete[] errorLog;
                }
        }

        // Удаление шейдеров, поскольку они больше не нужны
        glDeleteShader(vertexShader);
        glDeleteShader(fragmentShader);
        vertHFile.close();
        fragHFile.close();

        // Включение сортировки по глубине
        glEnable(GL_DEPTH_TEST);

        // Режим рисования только лицевых граней
        //glEnable(GL_CULL_FACE);
}
```

```
void DrawSoundTiles(GLuint shaderProgram, int width, int height) {
	std::vector<float> positionData;
	std::vector<float> normalsData;
	std::vector<float> texCoordsData;
	float tile_width = (2 + TILES_SPACING * (1 - actual_tiles_amount)) / actual_tiles_amount;

	for (int i = 0; i < actual_tiles_amount; i++) {
		float x_left = -1 + i * (tile_width + TILES_SPACING);
		float x_rght = x_left + tile_width;
		float ampl = tiles_amplitudes[i];
		float z_tile_width = tile_width / 2;

		std::vector<float> tmp_position_data = {
			x_rght, 0, z_tile_width,
			x_rght, ampl, z_tile_width,
			x_left, 0, z_tile_width,

			x_rght, ampl, z_tile_width,
			x_left, 0, z_tile_width,
			x_left, ampl, z_tile_width,


			x_rght, 0, -z_tile_width,
			x_rght, ampl, -z_tile_width,
			x_left, 0, -z_tile_width,

			x_rght, ampl, -z_tile_width,
			x_left, 0, -z_tile_width,
			x_left, ampl, -z_tile_width,

			x_rght, 0, -z_tile_width,
			x_rght, ampl, z_tile_width,
			x_rght, 0, z_tile_width,

			x_rght, ampl, z_tile_width,
			x_rght, 0, -z_tile_width,
			x_rght, ampl, -z_tile_width,

			x_left, 0, -z_tile_width,
			x_left, ampl, z_tile_width,
			x_left, 0, z_tile_width,

			x_left, ampl, z_tile_width,
			x_left, 0, -z_tile_width,
			x_left, ampl, -z_tile_width,

			x_rght, ampl, z_tile_width,
			x_left, ampl, -z_tile_width,
			x_left, ampl, z_tile_width,

			x_rght, ampl, z_tile_width,
			x_left, ampl, -z_tile_width,
			x_rght, ampl, -z_tile_width,
```

```
        x_rght, 0, z_tile_width,
        x_left, 0, -z_tile_width,
        x_left, 0, z_tile_width,

        x_rght, 0, z_tile_width,
        x_left, 0, -z_tile_width,
        x_rght, 0, -z_tile_width,
};

std::vector<float> tmp_normals_data = {
        .0f, .0f, 1.0f,
.0f, .0f, 1.0f,
.0f, .0f, 1.0f,

.0f, .0f, 1.0f,
.0f, .0f, 1.0f,
.0f, .0f, 1.0f,

.0f, .0f, -1.0f,
.0f, .0f, -1.0f,
.0f, .0f, -1.0f,

.0f, .0f, -1.0f,
.0f, .0f, -1.0f,
.0f, .0f, -1.0f,

1.0f, .0f, .0f,
1.0f, .0f, .0f,
1.0f, .0f, .0f,

1.0f, .0f, .0f,
1.0f, .0f, .0f,
1.0f, .0f, .0f,

-1.0f, .0f, .0f,
-1.0f, .0f, .0f,
-1.0f, .0f, .0f,

-1.0f, .0f, .0f,
-1.0f, .0f, .0f,
-1.0f, .0f, .0f,

.0f, 1.0f, .0f,
.0f, 1.0f, .0f,
.0f, 1.0f, .0f,

.0f, 1.0f, .0f,
.0f, 1.0f, .0f,
.0f, 1.0f, .0f,

.0f, -1.0f, .0f,
.0f, -1.0f, .0f,
.0f, -1.0f, .0f,
```

```
.0f, -1.0f, .0f,
.0f, -1.0f, .0f,
.0f, -1.0f, .0f,
};

std::vector<float> tmp_tex_coords_data = {
        1.0, 0.0,
        1.0, 1.0,
        0.0, 0.0,

        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 0.0,
        1.0, 1.0,
        0.0, 0.0,

        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,


        0.0, 0.0,
        1.0, 1.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,

        0.0, 0.0,
        1.0, 1.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,


        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,

        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 1.0,
```

```
                0.0, 0.0,
                1.0, 0.0,
        };

        positionData.insert(positionData.end(), tmp_position_data.begin(), tmp_position_data.end());
        normalsData.insert(normalsData.end(), tmp_normals_data.begin(), tmp_normals_data.end());
        texCoordsData.insert(texCoordsData.end(), tmp_tex_coords_data.begin(), tmp_tex_coords_data.end());
}

Matrix modelViewMatrix =
        Matrix::Scale(0.2, .2, .2) *
        Matrix::Translation(0, -0.1, -.3);

Matrix projectionMatrix = Matrix::Perspective(45.0f, (float)width / height, 0.05f, 6.0f);

float normalMatrix[9] = {
        modelViewMatrix.M[0][0], modelViewMatrix.M[0][1], modelViewMatrix.M[0][2],
        modelViewMatrix.M[1][0], modelViewMatrix.M[1][1], modelViewMatrix.M[1][2],
        modelViewMatrix.M[2][0], modelViewMatrix.M[2][2], modelViewMatrix.M[2][2]
};

Matrix generalMatrix = modelViewMatrix * projectionMatrix;

float Kd[3] = { .5, .5, .5 };
float Ka[3] = { .5, .5, .5 };
float Ks[3] = { .5, .5, .5 };


// Получение идентификатора uniform-параметра для матриц
GLuint generalMatrixLocation = glGetUniformLocation(shaderProgram, "MVP");
GLuint modelViewMatrixLocation = glGetUniformLocation(shaderProgram, "ModelViewMatrix");
GLuint projectionMatrixLocation = glGetUniformLocation(shaderProgram, "ProjectionMatrix");
GLuint normalMatrixLocation = glGetUniformLocation(shaderProgram, "NormalMatrix");

glUniformMatrix4fv(generalMatrixLocation, 1, GL_FALSE, &generalMatrix.M[0][0]);
glUniformMatrix4fv(modelViewMatrixLocation, 1, GL_FALSE, &modelViewMatrix.M[0][0]);
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix.M[0][0]);
glUniformMatrix3fv(normalMatrixLocation, 1, GL_FALSE, normalMatrix);

GLuint KdLocation = glGetUniformLocation(shaderProgram, "Kd");
GLuint KaLocation = glGetUniformLocation(shaderProgram, "Ka");
GLuint KsLocation = glGetUniformLocation(shaderProgram, "Ks");
GLuint ShininessLocation = glGetUniformLocation(shaderProgram, "Shininess");
GLuint EmissionLocation = glGetUniformLocation(shaderProgram, "Emission");

glUniform3fv(KdLocation, 1, Kd);
glUniform3fv(KaLocation, 1, Ka);
glUniform3fv(KsLocation, 1, Ks);
glUniform1f(ShininessLocation, 0.5);
glUniform3f(EmissionLocation, .1, .1, .1);

GLuint lights0IntensityLocation = glGetUniformLocation(shaderProgram, "lights[0].Intensity");
GLuint lights0PositionLocation = glGetUniformLocation(shaderProgram, "lights[0].Position");
GLuint lights1IntensityLocation = glGetUniformLocation(shaderProgram, "lights[1].Intensity");
```

```
GLuint lights1PositionLocation = glGetUniformLocation(shaderProgram, "lights[1].Position");
GLuint lights2IntensityLocation = glGetUniformLocation(shaderProgram, "lights[2].Intensity");
GLuint lights2PositionLocation = glGetUniformLocation(shaderProgram, "lights[2].Position");

glUniform3f(lights0IntensityLocation, 0, 0, 1);
glUniform3f(lights1IntensityLocation, 1, 1, 1);
glUniform3f(lights2IntensityLocation, 0, 0, 0);
glUniform4f(lights0PositionLocation, std::sin(running_time * 5), 0, 0, 0);
glUniform4f(lights1PositionLocation, std::sin(3.14 + running_time * 5), .0, 0, 0);
glUniform4f(lights2PositionLocation, 0, 0, 0, 0);

GLuint vboHandles[4];

// Генерируем буфер для хранения координат вершин
glGenBuffers(3, vboHandles);
GLuint positionBufferHandle = vboHandles[0];
GLuint texCoordsBufferHandle = vboHandles[1];
GLuint normalsBufferHandle = vboHandles[2];

// Заполняем сгенерированный буфер координат
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * positionData.size(), positionData.data(), GL_STATIC_DRAW);

// Заполняем сгенерированный буфер координат текстур
glBindBuffer(GL_ARRAY_BUFFER, texCoordsBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * texCoordsData.size(), texCoordsData.data(), GL_STATIC_DRAW);

// Заполняем сгенерированный буфер координат текстур
glBindBuffer(GL_ARRAY_BUFFER, normalsBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * normalsData.size(), normalsData.data(), GL_STATIC_DRAW);

// Создание объекта массива вершин, который будет определять отношения между буферами и входными атрибутами
GLuint VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Активация массива вершинных атрибутов
glEnableVertexAttribArray(0); // Координаты вершин
glEnableVertexAttribArray(1); // Координаты текстур вершин
glEnableVertexAttribArray(2); // Координаты нормалей вершин

// Привяжем индекс 0 к буферу с координатами
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);

// Привяжем индекс 2 к буферу с текстурными координатами вершин
glBindBuffer(GL_ARRAY_BUFFER, texCoordsBufferHandle);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);

// Привяжем индекс 3 к буферу с нормалями
glBindBuffer(GL_ARRAY_BUFFER, normalsBufferHandle);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);

textureTile.bindGLTexture(GL_TEXTURE0);
```

```
        // Рисование треугольника
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, positionData.size() / 3);

        textureTile.unbindGLTexture();

        // Отвяжем объект массива вершин
        glBindVertexArray(0);

        // Очистка
        glDeleteVertexArrays(1, &VAO);
        glDeleteBuffers(3, vboHandles);
}

void DrawSkybox(GLuint shaderProgram, int width, int height) {
        // Определение вершин треугольника
        float positionData[] = {
                1, -1, 1,
                1, 1, 1,
                -1, -1, 1,

                1, 1, 1,
                -1, -1, 1,
                -1, 1, 1,



                1, -1, -1,
                1, 1, -1,
                -1, -1, -1,

                1, 1, -1,
                -1, -1, -1,
                -1, 1, -1,

                1, -1, -1,
                1, 1, 1,
                1, -1, 1,

                1, 1, 1,
                1, -1, -1,
                1, 1, -1,

                -1, -1, -1,
                -1, 1, 1,
                -1, -1, 1,

                -1, 1, 1,
                -1, -1, -1,
                -1, 1, -1,

                1, 1, 1,
                -1, 1, -1,
                -1, 1, 1,
```

```
        1, 1, 1,
        -1, 1, -1,
        1, 1, -1,

        1, -1, 1,
        -1,-1, -1,
        -1, -1, 1,

        1, -1, 1,
        -1, -1, -1,
        1, -1, -1,
};

float normalsData[] = {
        .0f, .0f, 1.0f,
        .0f, .0f, 1.0f,
        .0f, .0f, 1.0f,

        .0f, .0f, 1.0f,
        .0f, .0f, 1.0f,
        .0f, .0f, 1.0f,

        .0f, .0f, -1.0f,
        .0f, .0f, -1.0f,
        .0f, .0f, -1.0f,

        .0f, .0f, -1.0f,
        .0f, .0f, -1.0f,
        .0f, .0f, -1.0f,

        1.0f, .0f, .0f,
        1.0f, .0f, .0f,
        1.0f, .0f, .0f,

        1.0f, .0f, .0f,
        1.0f, .0f, .0f,
        1.0f, .0f, .0f,

        -1.0f, .0f, .0f,
        -1.0f, .0f, .0f,
        -1.0f, .0f, .0f,

        -1.0f, .0f, .0f,
        -1.0f, .0f, .0f,
        -1.0f, .0f, .0f,

        .0f, 1.0f, .0f,
        .0f, 1.0f, .0f,
        .0f, 1.0f, .0f,

        .0f, 1.0f, .0f,
        .0f, 1.0f, .0f,
        .0f, 1.0f, .0f,
```

```
        .0f, -1.0f, .0f,
        .0f, -1.0f, .0f,
        .0f, -1.0f, .0f,

        .0f, -1.0f, .0f,
        .0f, -1.0f, .0f,
        .0f, -1.0f, .0f,


};

float texCoordsData[] = {
        1.0, 0.0,
        1.0, 1.0,
        0.0, 0.0,

        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 0.0,
        1.0, 1.0,
        0.0, 0.0,

        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,


        0.0, 0.0,
        1.0, 1.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,

        0.0, 0.0,
        1.0, 1.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,


        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,
```

```
        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0,

        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,
};

float angle = running_time;

Matrix modelViewMatrix =
        Matrix::Scale(1, 1, 1) *
        Matrix::RotationX(angle / 8) *      // Поворот куба вокруг оси x
        Matrix::RotationY(angle / 16) *  // Поворот куба вокруг оси y
        Matrix::RotationZ(angle / 4) *  // Поворот куба вокруг оси y
        Matrix::Translation(0, 0, 0);

Matrix projectionMatrix = Matrix::Perspective(45.0f, (float)width / height, 0.05f, 6.0f);

float normalMatrix[9] = {
        modelViewMatrix.M[0][0], modelViewMatrix.M[0][1], modelViewMatrix.M[0][2],
        modelViewMatrix.M[1][0], modelViewMatrix.M[1][1], modelViewMatrix.M[1][2],
        modelViewMatrix.M[2][0], modelViewMatrix.M[2][2], modelViewMatrix.M[2][2]
};

Matrix generalMatrix = modelViewMatrix * projectionMatrix;

float Kd[3] = { .5, .5, .5 };
float Ka[3] = { .5, .5, .5 };
float Ks[3] = { .5, .5, .5 };


// Получение идентификатора uniform-параметра для матриц
GLuint generalMatrixLocation = glGetUniformLocation(shaderProgram, "MVP");
GLuint modelViewMatrixLocation = glGetUniformLocation(shaderProgram, "ModelViewMatrix");
GLuint projectionMatrixLocation = glGetUniformLocation(shaderProgram, "ProjectionMatrix");
GLuint normalMatrixLocation = glGetUniformLocation(shaderProgram, "NormalMatrix");

glUniformMatrix4fv(generalMatrixLocation, 1, GL_FALSE, &generalMatrix.M[0][0]);
glUniformMatrix4fv(modelViewMatrixLocation, 1, GL_FALSE, &modelViewMatrix.M[0][0]);
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix.M[0][0]);
glUniformMatrix3fv(normalMatrixLocation, 1, GL_FALSE, normalMatrix);

GLuint KdLocation = glGetUniformLocation(shaderProgram, "Kd");
GLuint KaLocation = glGetUniformLocation(shaderProgram, "Ka");
GLuint KsLocation = glGetUniformLocation(shaderProgram, "Ks");
GLuint ShininessLocation = glGetUniformLocation(shaderProgram, "Shininess");
GLuint EmissionLocation = glGetUniformLocation(shaderProgram, "Emission");

glUniform3fv(KdLocation, 1, Kd);
glUniform3fv(KaLocation, 1, Ka);
glUniform3fv(KsLocation, 1, Ks);
```

```
glUniform1f(ShininessLocation, 0.5);
glUniform3f(EmissionLocation, 1, 1, 1);

GLuint lights0IntensityLocation = glGetUniformLocation(shaderProgram, "lights[0].Intensity");
GLuint lights0PositionLocation = glGetUniformLocation(shaderProgram, "lights[0].Position");
GLuint lights1IntensityLocation = glGetUniformLocation(shaderProgram, "lights[1].Intensity");
GLuint lights1PositionLocation = glGetUniformLocation(shaderProgram, "lights[1].Position");
GLuint lights2IntensityLocation = glGetUniformLocation(shaderProgram, "lights[2].Intensity");
GLuint lights2PositionLocation = glGetUniformLocation(shaderProgram, "lights[2].Position");

glUniform3f(lights0IntensityLocation, 0, 0, 0);
glUniform3f(lights1IntensityLocation, 0, 0, 0);
glUniform3f(lights2IntensityLocation, 0, 0, 0);
glUniform4f(lights0PositionLocation, 1, 0, 0, 0);
glUniform4f(lights1PositionLocation, -1, .0, 0, 0);
glUniform4f(lights2PositionLocation, 0, 1, 0, 0);

GLuint vboHandles[4];

// Генерируем буфер для хранения координат вершин
glGenBuffers(3, vboHandles);
GLuint positionBufferHandle = vboHandles[0];
GLuint texCoordsBufferHandle = vboHandles[1];
GLuint normalsBufferHandle = vboHandles[2];

// Заполняем сгенерированный буфер координат
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(positionData), positionData, GL_STATIC_DRAW);

// Заполняем сгенерированный буфер координат текстур
glBindBuffer(GL_ARRAY_BUFFER, texCoordsBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(texCoordsData), texCoordsData, GL_STATIC_DRAW);

// Заполняем сгенерированный буфер координат текстур
glBindBuffer(GL_ARRAY_BUFFER, normalsBufferHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(normalsData), normalsData, GL_STATIC_DRAW);

// Создание объекта массива вершин, который будет определять отношения между буферами и входными атрибутами
GLuint VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Активация массива вершинных атрибутов
glEnableVertexAttribArray(0); // Координаты вершин
glEnableVertexAttribArray(1); // Координаты текстур вершин
glEnableVertexAttribArray(2); // Координаты нормалей вершин

// Привяжем индекс 0 к буферу с координатами
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);

// Привяжем индекс 2 к буферу с текстурными координатами вершин
glBindBuffer(GL_ARRAY_BUFFER, texCoordsBufferHandle);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
```

```
        // Привяжем индекс 3 к буферу с нормалями
        glBindBuffer(GL_ARRAY_BUFFER, normalsBufferHandle);
        glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);

        textureBackground.bindGLTexture(GL_TEXTURE0);

        // Рисование треугольника
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 36);

        textureBackground.unbindGLTexture();

        // Отвяжем объект массива вершин
        glBindVertexArray(0);

        // Очистка
        glDeleteVertexArrays(1, &VAO);
        glDeleteBuffers(3, vboHandles);
}


// Основная функция рисования
void Draw(int width, int height)
{
        // Очистка буфера глубины и буфера цвета
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glUseProgram(shaderProgram);

        DrawSoundTiles(shaderProgram, width, height);
        DrawSkybox(shaderProgram, width, height);
}

#endif // PAINTER_H
```

main.vert

```
#version 430

layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec2 VertexTexPosition;
layout (location=2) in vec3 VertexNormal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

struct LightInfo {
    vec4 Position;
    vec3 Intensity;
};
```

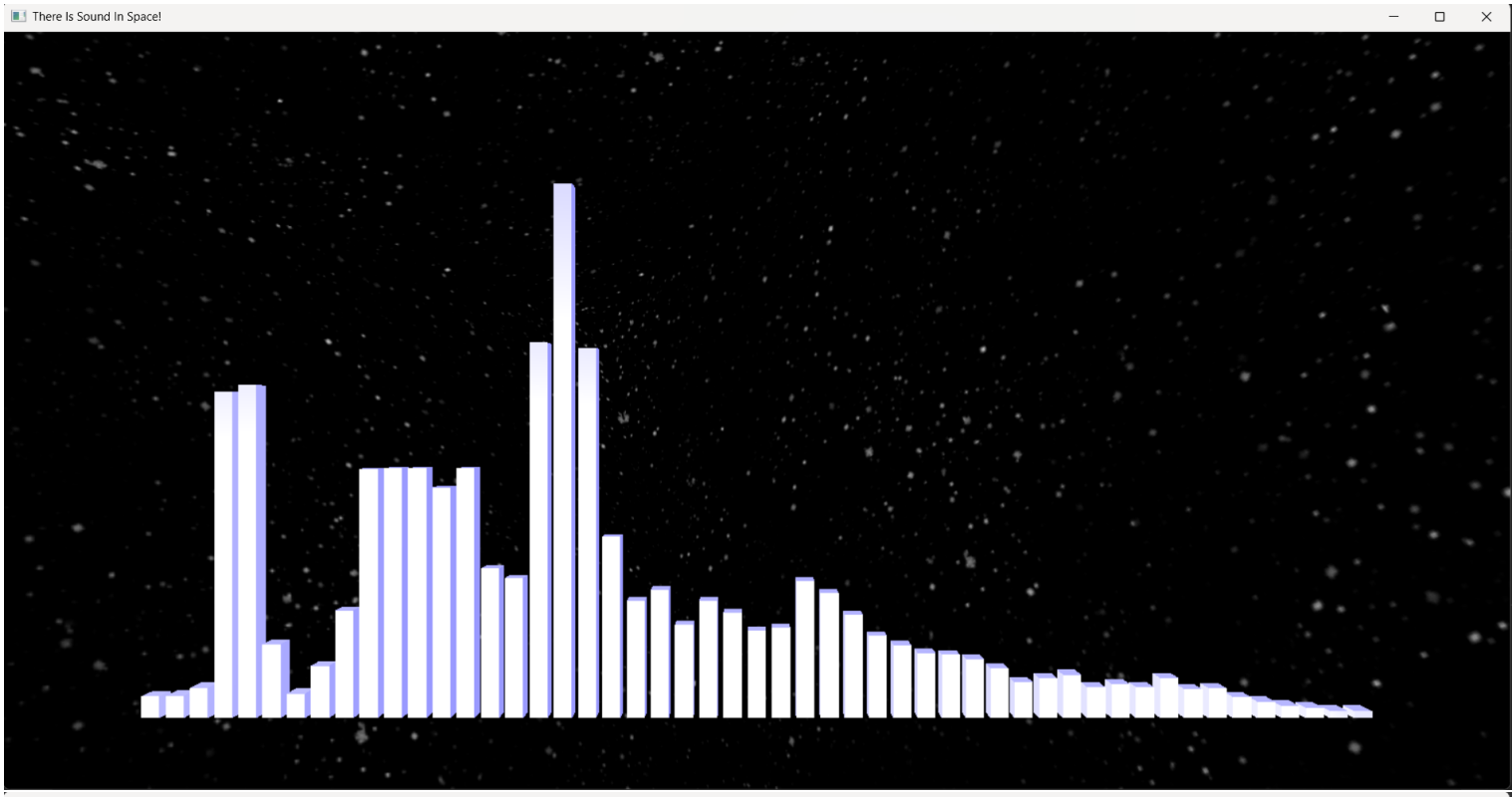
```
uniform LightInfo lights[3];

uniform vec3 Kd;
uniform vec3 Ka;
uniform vec3 Ks;
uniform float Shininess;
uniform vec3 Emission;



out vec3 LightIntensity;
out vec2 TexCoord;

vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
    vec3 s = normalize(vec3(lights[lightIndex].Position - position));
    vec3 v = normalize((-position).xyz);
    vec3 r = reflect( -s, norm );
    vec3 I = lights[lightIndex].Intensity;

    return I * ( Ka + Kd * max( dot(s, norm), 0.0 ) + Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main()
{
    TexCoord = VertexTexPosition;

        vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyePosition = ModelViewMatrix * vec4(VertexPosition,1.0);

    LightIntensity = Emission;

    for( int i = 0; i < 3; i++ )
       LightIntensity += ads( i, eyePosition, eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

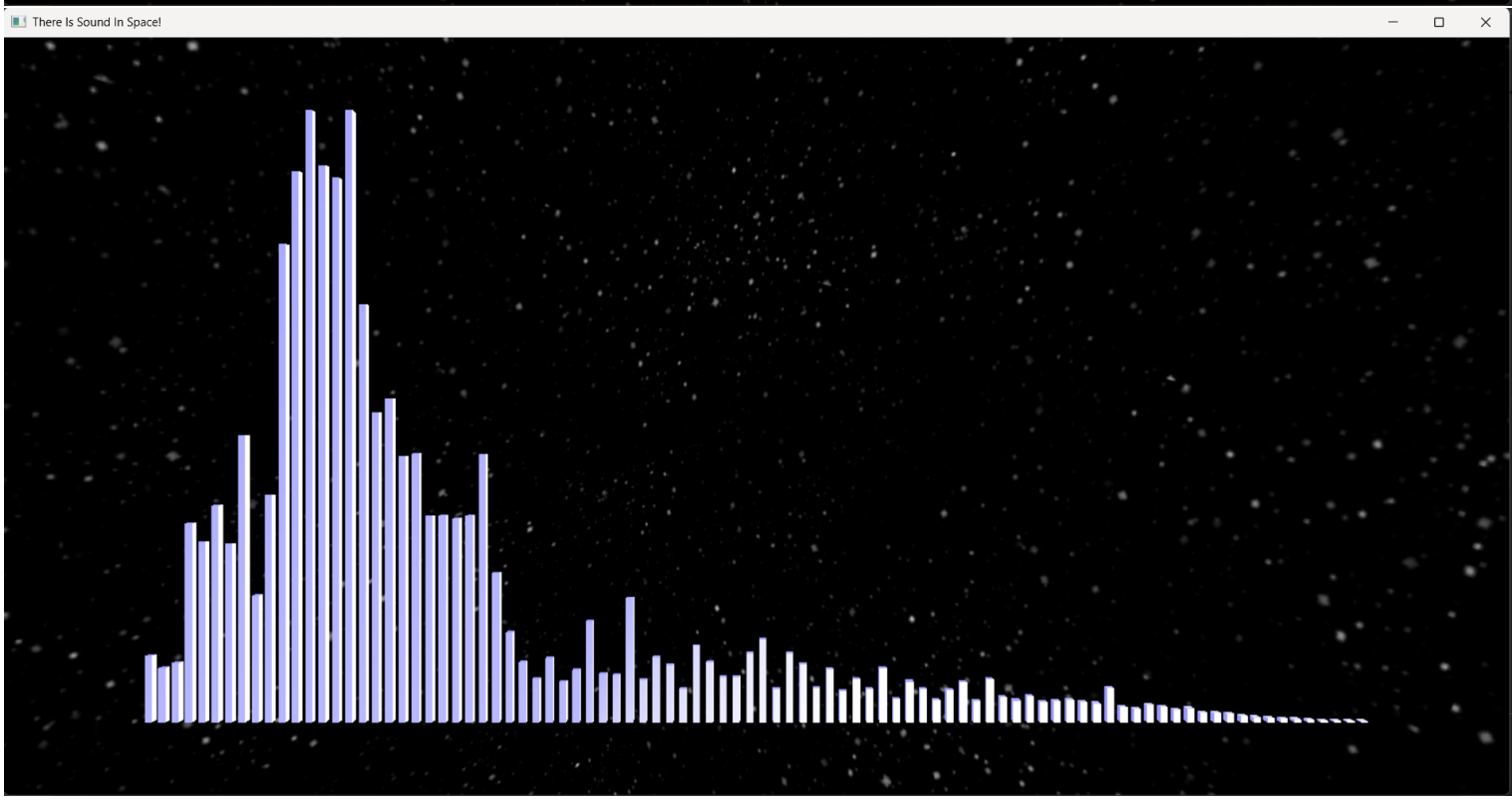
---

main.frag

```
#version 430

in vec3 LightIntensity;
in vec2 TexCoord;

layout( binding = 0 ) uniform sampler2D BaseTexture;

layout( location = 0 ) out vec4 FragColor;

void main() {
    vec4 texColor = texture(BaseTexture, TexCoord) * vec4(LightIntensity, 1.0);
        FragColor = texColor;
}
```

4 Результаты выполнения программы.





5 Заключение

Библиотека OpenGL позволяет эффективно использовать ресурсы системы для компьютерной графики.