МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



**ОТЧЁТ**

**Расчётно-графическое задание**

**«*Отрисовка и анимация многогранников с помощью OpenGL»***

по дисциплине:

**«*Графические системы*»**

|  |  |
| --- | --- |
| Выполнил: *Астафьев В.Е.*  Группа  *АО-71* | Проверил:  Кухто А.В. |

Новосибирск

2018

**Цель работы:**

Отрисовка и анимация многогранников с помощью OpenGL

**Программный код**:

**obj.h:**

#pragma once

#ifndef OBJ\_H

#define OBJ\_H

#include <GL/glew.h>

/// .obj модель вершин

typedef struct {

float x, y, z; // Cooridnates

} obj\_vertex;

/// .obj model face

typedef struct {

int count; // Кол-во вершин

int\*v; // массив индексов вершин

} obj\_face;

/// .obj модель

typedef struct {

int vc; // Количество вершин

obj\_vertex\*v; // Массив вершин

int fc; // Количество лицевых вершин

obj\_face\*f; // массив лиц

} obj;

/// чтение модели из obj файла

/// Вернуть ошибку.Модель храниться в pObj

int read\_obj(const char\*path, obj\*pObj);

/// Освобождает .obj модель памяти

void free\_obj(obj o);

/// клонирует .obj модель

obj obj\_clone(obj o);

/// Определяет, не является ли .model триангулированной

int obj\_need\_triangulation(obj o);

/// Уменьшает все грани модели .obj до треугольников

obj obj\_triangulate(obj o);

#endif

**settings\_window.h:**

#pragma once

#ifndef SETTINGS\_WINDOW\_H

#define SETTINGS\_WINDOW\_H

#include <Windows.h>

/// Кнопка обратного вызова

typedef void (CALLBACK\*settings\_window\_click\_callback)(HWND hWnd, HWND hControl, LPARAM lParam);

/// Создает окно размера с обратным вызовом

HWND create\_window(settings\_window\_click\_callback callback, LPARAM lParam, int w, int h);

// Добавляет элемент управления кнопки в окно и возвращает его обработчик

HWND add\_button(HWND hWnd, int x, int y, int w, int h, LPCTSTR text);

/// Добавляет текстовый элемент управления в окно и возвращает его обработчик

HWND add\_text(HWND hWnd, int x, int y, int w, int h, LPCTSTR text);

#endif

**main.c:**

**#include <GL/glew.h> // The OpenGL Extension Wrangler Library**

**#include <GLFW/glfw3.h> // OpenGL Framework**

**#include <CGLM/cglm.h> // C OpenGL Math**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <tchar.h>**

**#include "obj.h"**

**#include "settings\_window.h"**

**#define VEC3(x,y,z) ((vec3){x,y,z}) // Easy macros for vec3 initialization**

**/// Struct for storing figure buffers**

**typedef struct {**

**int count;**

**GLuint \*VAO, \*VBO;**

**} figure;**

**int fig\_count; // Total amount of figures**

**figure\*figs; // Figures array**

**figure\*fig\_current; // Currently selected figure**

**vec3 translate; // Translation vector**

**vec3 rotate; // Rotation vector**

**vec3 scale; // Scale vector**

**/// Returns file contents**

**const char\*read\_file(const char\*path) {**

**FILE\*file;**

**long size;**

**char\*buffer;**

**if (fopen\_s(&file, path, "r"))**

**return NULL;**

**fseek(file, 0, SEEK\_END);**

**size = ftell(file);**

**fseek(file, 0, SEEK\_SET);**

**buffer = calloc(size + 1, 1);**

**fread(buffer, 1, size, file);**

**fclose(file);**

**return buffer;**

**}**

**/// Compiles shader of from file**

**GLboolean create\_shader(const char\*shader\_path, GLenum type, GLuint\*shader) {**

**GLint success;**

**GLint log\_size;**

**GLchar\*log;**

**const char\*source = read\_file(shader\_path); // Trying to read shader source**

**if (!source) {**

**puts("Failed to read shader file");**

**return GL\_FALSE;**

**}**

**\*shader = glCreateShader(type); // Allocating shader**

**glShaderSource(\*shader, 1, &source, NULL); // Assigning shader source**

**glCompileShader(\*shader); // Compiling shader**

**glGetShaderiv(\*shader, GL\_COMPILE\_STATUS, &success); // Checking if compilation was successful**

**if (!success) { // If not**

**puts("Failed to compile shader");**

**glGetShaderiv(\*shader, GL\_INFO\_LOG\_LENGTH, &log\_size); // Get error length**

**log = calloc(log\_size + 1, 1); // Allocate memory**

**glGetShaderInfoLog(\*shader, log\_size, NULL, log); // Get error string**

**puts(log); // Print to console**

**free(log);**

**return GL\_FALSE;**

**}**

**return GL\_TRUE;**

**}**

**/// Compiles shaders to shader program**

**GLboolean create\_program(const char\*vshader\_path, const char\*fshader\_path, GLuint\*program) {**

**GLuint vshader;**

**GLuint fshader;**

**GLint success;**

**GLint log\_size;**

**GLchar\*log;**

**// Trying to create vertex shader**

**if (!create\_shader(vshader\_path, GL\_VERTEX\_SHADER, &vshader)) {**

**puts("Vertex shader creation failed");**

**return GL\_FALSE;**

**}**

**// Trying to create fragment shader**

**if (!create\_shader(fshader\_path, GL\_FRAGMENT\_SHADER, &fshader)) {**

**puts("Fragment shader creation failed");**

**return GL\_FALSE;**

**}**

**\*program = glCreateProgram(); // Allocating shader program**

**glAttachShader(\*program, vshader); // Attaching shaders**

**glAttachShader(\*program, fshader);**

**glLinkProgram(\*program); // Linking program**

**glDeleteShader(vshader); // Freeing shaders**

**glDeleteShader(fshader);**

**glGetProgramiv(\*program, GL\_LINK\_STATUS, &success); // Checking success**

**if (!success) { // If failed**

**puts("Failed to link shader program");**

**glGetProgramiv(\*program, GL\_INFO\_LOG\_LENGTH, &log\_size); // Get error length**

**log = calloc(log\_size + 1, 1); // Allocate string**

**glGetProgramInfoLog(program, log\_size, NULL, log); // Get error string**

**puts(log); // Print error**

**free(log);**

**return GL\_FALSE;**

**}**

**return GL\_TRUE;**

**}**

**/// Prints OpenGL version string to console**

**void print\_gl\_info() {**

**const GLubyte\*rend = glGetString(GL\_RENDERER); // OpenGL renderer information**

**const GLubyte\*vers = glGetString(GL\_VERSION); // OpenGL version information**

**printf("Renderer: %s\n", rend);**

**printf("OpenGL version: %s\n", vers);**

**}**

**/// Converts .obj model to OpenGL buffers**

**void obj\_to\_gl(obj o, int\*count, GLuint\*\*vao, GLuint\*\*vbo) {**

**\*count = o.fc;**

**GLuint\*VAO = malloc(sizeof(GLuint)\*\*count); // Allocate buffer arrays**

**GLuint\*VBO = malloc(sizeof(GLuint)\*\*count);**

**glGenVertexArrays(\*count, VAO); // Allocate buffers**

**glGenBuffers(\*count, VBO);**

**for (int i = 0; i < o.fc; ++i) { // One VAO and one VBO for each model face**

**int vb\_size = sizeof(float)\*o.f[i].count \* 3; // Precalculate VBO size**

**float\*verts = malloc(vb\_size); // Allocate temporary buffer**

**for (int j = 0; j < o.f[i].count; ++j) {**

**verts[j \* 3 + 0] = o.v[o.f[i].v[j]].x; // Flatten .obj model face to buffer**

**verts[j \* 3 + 1] = o.v[o.f[i].v[j]].y;**

**verts[j \* 3 + 2] = o.v[o.f[i].v[j]].z;**

**}**

**glBindVertexArray(VAO[i]); // Bind VAO**

**glBindBuffer(GL\_ARRAY\_BUFFER, VBO[i]); // Bind VBO to it**

**glBufferData(GL\_ARRAY\_BUFFER, vb\_size, verts, GL\_STATIC\_DRAW); // Fill VBO**

**glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(float) \* 3, 0); // Set shader parameter**

**glEnableVertexAttribArray(0); // Enable parameter**

**}**

**glBindVertexArray(0); // Unbind VAO**

**\*vao = VAO;**

**\*vbo = VBO;**

**}**

**/// Create figure from .obj file**

**figure create\_figure(const char\*obj\_path) {**

**figure fig;**

**obj o,t;**

**read\_obj(obj\_path, &t); // Reading .obj mode**

**o = obj\_triangulate(t); // Triangulating it**

**obj\_to\_gl(o, &fig.count, &fig.VAO, &fig.VBO); // Converting to buffers**

**return fig;**

**}**

**/// Initializes predefined figures and it's buttons**

**void init\_figures(HWND hWnd, figure\*\*figures, HWND\*\*buttons, int\*count) {**

**\*count = 5;**

**\*figures = malloc(sizeof(figure\*) \* 5); // Allocate arrays**

**\*buttons = malloc(sizeof(HWND\*) \* 5);**

**(\*figures)[0] = create\_figure("tetra.obj"); // Read figures**

**(\*figures)[1] = create\_figure("cube.obj");**

**(\*figures)[2] = create\_figure("octa.obj");**

**(\*figures)[3] = create\_figure("dodeca.obj");**

**(\*figures)[4] = create\_figure("icaso.obj");**

**(\*buttons)[0] = add\_button(hWnd, 0, 00, 100, 22, \_T("Тетраэдр")); // Add buttons**

**(\*buttons)[1] = add\_button(hWnd, 0, 22, 100, 22, \_T("Гексаэдр"));**

**(\*buttons)[2] = add\_button(hWnd, 0, 44, 100, 22, \_T("Октаэдр"));**

**(\*buttons)[3] = add\_button(hWnd, 0, 66, 100, 22, \_T("Додекаэдр"));**

**(\*buttons)[4] = add\_button(hWnd, 0, 88, 100, 22, \_T("Икосаэдр"));**

**fig\_current = &((\*figures)[0]); // Select first figure**

**figs = \*figures; // Store figures to global**

**}**

**/// Updates transformation hint text**

**void update\_label(HWND text, float value) {**

**char\*buffer = malloc(16);**

**sprintf\_s(buffer, 16, "%.1f", value);**

**SetWindowTextA(text, buffer);**

**free(buffer);**

**}**

**/// Called when button is clicked**

**void CALLBACK on\_settings\_click(HWND hWnd, HWND hControl, LPARAM lParam) {**

**LPVOID\*data = lParam; // Extract data**

**int count = data[0]; // Extract figure count**

**figure\*figures = data[1]; // Extract figures**

**HWND\*fig\_buttons = data[2]; // Extract figure buttons**

**HWND\*buttons = data[3]; // Extract buttons**

**HWND\*texts = data[4]; // Extract labels**

**for (int i = 0; i < count; ++i) // Check if figure button is clicked**

**if (hControl == fig\_buttons[i]) {**

**fig\_current = &figures[i]; // If so, select corresponding figure**

**return;**

**}**

**// Check which transformation button is clicked**

**if (hControl == buttons[0]) translate[0] -= .1f; // Pos X-**

**if (hControl == buttons[1]) translate[0] += .1f; // Pos X+**

**if (hControl == buttons[2]) translate[1] -= .1f; // Pos Y-**

**if (hControl == buttons[3]) translate[1] += .1f; // Pos Y+**

**if (hControl == buttons[4]) translate[2] -= .1f; // Pos Z-**

**if (hControl == buttons[5]) translate[2] += .1f; // Pos Z+**

**if (hControl == buttons[6]) rotate[0] -= .1f; // Rot X-**

**if (hControl == buttons[7]) rotate[0] += .1f; // Rot X+**

**if (hControl == buttons[8]) rotate[1] -= .1f; // Rot Y-**

**if (hControl == buttons[9]) rotate[1] += .1f; // Rot Y+**

**if (hControl == buttons[10]) rotate[2] -= .1f; // Rot Z-**

**if (hControl == buttons[11]) rotate[2] += .1f; // Rot Z+**

**if (hControl == buttons[12]) scale[0] -= .1f; // Scale X-**

**if (hControl == buttons[13]) scale[0] += .1f; // Scale X+**

**if (hControl == buttons[14]) scale[1] -= .1f; // Scale Y-**

**if (hControl == buttons[15]) scale[1] += .1f; // Scale Y+**

**if (hControl == buttons[16]) scale[2] -= .1f; // Scale Z-**

**if (hControl == buttons[17]) scale[2] += .1f; // Scale Z+**

**// Check reset buttons**

**if (hControl == buttons[18]) { // Reset Pos**

**translate[0] = 0;**

**translate[1] = 0;**

**translate[2] = 0;**

**}**

**if (hControl == buttons[19]) { // Reset Rot**

**rotate[0] = 0;**

**rotate[1] = 0;**

**rotate[2] = 0;**

**}**

**if (hControl == buttons[20]) { // Reset Scale**

**scale[0] = 1;**

**scale[1] = 1;**

**scale[2] = 1;**

**}**

**// Update labels**

**update\_label(texts[0], translate[0]); // Pos**

**update\_label(texts[1], translate[1]);**

**update\_label(texts[2], translate[2]);**

**update\_label(texts[3], rotate[0]); // Rot**

**update\_label(texts[4], rotate[1]);**

**update\_label(texts[5], rotate[2]);**

**update\_label(texts[6], scale[0]); // Scale**

**update\_label(texts[7], scale[1]);**

**update\_label(texts[8], scale[2]);**

**}**

**/// Initialize settings window**

**void init\_settings() {**

**LPVOID\*data = malloc(sizeof(LPVOID) \* 5); // Allocate data to pass to callback**

**HWND hWnd = create\_window(on\_settings\_click, data, 458, 148); // Create window**

**init\_figures(hWnd, &data[1], &data[2], &data[0]); // Initialize figures**

**HWND\*btns = malloc(sizeof(HWND) \* 21); // Allocate transformation buttons array**

**HWND\*texts = malloc(sizeof(HWND) \* 9); // Allocate transformation value hints array**

**add\_text(hWnd, 112, 0, 100, 22, \_T("Pos")); // Hints**

**add\_text(hWnd, 222, 0, 100, 22, \_T("Rot"));**

**add\_text(hWnd, 332, 0, 100, 22, \_T("Scale"));**

**btns[0] = add\_button(hWnd, 162, 22, 25, 22, \_T("X-")); // Pos**

**btns[1] = add\_button(hWnd, 187, 22, 25, 22, \_T("X+"));**

**btns[2] = add\_button(hWnd, 162, 44, 25, 22, \_T("Y-"));**

**btns[3] = add\_button(hWnd, 187, 44, 25, 22, \_T("Y+"));**

**btns[4] = add\_button(hWnd, 162, 66, 25, 22, \_T("Z-"));**

**btns[5] = add\_button(hWnd, 187, 66, 25, 22, \_T("Z+"));**

**texts[0] = add\_text(hWnd, 112, 22, 50, 22, \_T(""));**

**texts[1] = add\_text(hWnd, 112, 44, 50, 22, \_T(""));**

**texts[2] = add\_text(hWnd, 112, 66, 50, 22, \_T(""));**

**btns[6] = add\_button(hWnd, 272, 22, 25, 22, \_T("X-")); // Rot**

**btns[7] = add\_button(hWnd, 297, 22, 25, 22, \_T("X+"));**

**btns[8] = add\_button(hWnd, 272, 44, 25, 22, \_T("Y-"));**

**btns[9] = add\_button(hWnd, 297, 44, 25, 22, \_T("Y+"));**

**btns[10] = add\_button(hWnd, 272, 66, 25, 22, \_T("Z-"));**

**btns[11] = add\_button(hWnd, 297, 66, 25, 22, \_T("Z+"));**

**texts[3] = add\_text(hWnd, 222, 22, 50, 22, \_T(""));**

**texts[4] = add\_text(hWnd, 222, 44, 50, 22, \_T(""));**

**texts[5] = add\_text(hWnd, 222, 66, 50, 22, \_T(""));**

**btns[12] = add\_button(hWnd, 382, 22, 25, 22, \_T("X-")); // Scale**

**btns[13] = add\_button(hWnd, 407, 22, 25, 22, \_T("X+"));**

**btns[14] = add\_button(hWnd, 382, 44, 25, 22, \_T("Y-"));**

**btns[15] = add\_button(hWnd, 407, 44, 25, 22, \_T("Y+"));**

**btns[16] = add\_button(hWnd, 382, 66, 25, 22, \_T("Z-"));**

**btns[17] = add\_button(hWnd, 407, 66, 25, 22, \_T("Z+"));**

**texts[6] = add\_text(hWnd, 332, 22, 50, 22, \_T(""));**

**texts[7] = add\_text(hWnd, 332, 44, 50, 22, \_T(""));**

**texts[8] = add\_text(hWnd, 332, 66, 50, 22, \_T(""));**

**btns[18] = add\_button(hWnd, 162, 88, 50, 22, \_T("Reset")); // Pos**

**btns[19] = add\_button(hWnd, 272, 88, 50, 22, \_T("Reset")); // Rot**

**btns[20] = add\_button(hWnd, 382, 88, 50, 22, \_T("Reset")); // Scale**

**data[3] = btns; // Store it into data**

**data[4] = texts;**

**on\_settings\_click(hWnd, NULL, data); // Kinda hacky way to update labels**

**}**

**int main() {**

**GLFWwindow\*window; // Main rendering window**

**GLuint program; // Shader program**

**GLint col\_loc, model\_loc, proj\_loc, view\_loc; // Shader uniforms**

**mat4 model, proj, view; // Matrices**

**int w, h; // Viewbuffer width and height**

**if (!glfwInit()) { // Try to initialize GLFW**

**puts("Failed to initialize GLFW");**

**return -1;**

**}**

**glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 2); // Require OpenGL at least 2.0**

**glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 0);**

**glfwWindowHint(GLFW\_RESIZABLE, GL\_FALSE); // Forbid window resizing**

**window = glfwCreateWindow(400, 400, "RGZ", NULL, NULL); // Create render window**

**if (!window) {**

**glfwTerminate();**

**puts("Failed to initialize window");**

**return -1;**

**}**

**glfwMakeContextCurrent(window); // Assing render context to it**

**glEnable(GL\_DEPTH\_TEST); // Enable depth**

**glDepthFunc(GL\_LEQUAL);**

**glfwSwapInterval(1); // Slow down rendering a bit**

**if (glewInit()) { // Try to initialize GLEW**

**puts("Failed to initialize GLEW");**

**return -1;**

**}**

**print\_gl\_info(); // Print OpenGL information**

**if (!create\_program("vertex.shader", "fragment.shader", &program)) { // Load shaders**

**glfwDestroyWindow(window);**

**glfwTerminate();**

**puts("Failed to create shader program");**

**return -1;**

**}**

**col\_loc = glGetUniformLocation(program, "col"); // Get uniform locations**

**model\_loc = glGetUniformLocation(program, "model");**

**proj\_loc = glGetUniformLocation(program, "proj");**

**view\_loc = glGetUniformLocation(program, "view");**

**translate[0] = 0; // Set default transformations**

**translate[1] = 0;**

**translate[2] = 0;**

**rotate[0] = 0;**

**rotate[1] = 0;**

**rotate[2] = 0;**

**scale[0] = 1;**

**scale[1] = 1;**

**scale[2] = 1;**

**init\_settings(); // Initialize settings window**

**glfwGetFramebufferSize(window, &w, &h); // Get framebuffer size**

**glm\_perspective(glm\_rad(45.f), (float)w / h, .1f, 100.f, proj); // Get projection matrix**

**glm\_lookat(VEC3(0, 2, 3), GLM\_VEC3\_ZERO, VEC3(0, 1, 0), view); // Get view matrix**

**glUseProgram(program); // Activate shader program**

**glUniformMatrix4fv(proj\_loc, 1, GL\_FALSE, &proj[0][0]); // Store projection and view matrices**

**glUniformMatrix4fv(view\_loc, 1, GL\_FALSE, &view[0][0]); // to uniforms**

**glLineWidth(2.f); // Increase line width**

**glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set default render context color**

**while (!glfwWindowShouldClose(window)) { // Rendering loop**

**glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // Clear color and depth buffers**

**glm\_mat4\_copy(GLM\_MAT4\_IDENTITY, model); // Calculate model matrix**

**glm\_translate(model, translate);**

**glm\_rotate(model, glm\_rad(45 \* rotate[0]), GLM\_XUP);**

**glm\_rotate(model, glm\_rad(45 \* rotate[1]), GLM\_YUP);**

**glm\_rotate(model, glm\_rad(45 \* rotate[2]), GLM\_ZUP);**

**glm\_scale(model, scale);**

**glUniformMatrix4fv(model\_loc, 1, GL\_FALSE, &model[0][0]); // Store model matrix**

**if (fig\_current) { // If any figure is selected**

**for (int i = 0; i < fig\_current->count; ++i) { // Draw each of its faces**

**glBindVertexArray(fig\_current->VAO[i]); // Activate face VAO**

**glUniform3f(col\_loc, .0f, .0f, .0f); // Set face color (same as background so its invisible**

**glDrawArrays(GL\_TRIANGLES, 0, 3); // Draw solid face**

**glUniform3f(col\_loc, 1.f, 1.f, 1.f); // Set edge color**

**glDrawArrays(GL\_LINE\_LOOP, 0, 3); // Draw face edges**

**}**

**}**

**glfwPollEvents(); // Pump GLFW events**

**glfwSwapBuffers(window); // Swap back buffer**

**}**

**for (int i = 0; i < fig\_count; ++i) { // Release buffers**

**glDeleteVertexArrays(figs[i].count, figs[i].VAO);**

**glDeleteBuffers(figs[i].count, figs[i].VBO);**

**free(figs[i].VAO);**

**free(figs[i].VBO);**

**}**

**glDeleteProgram(program); // Release shader program**

**glfwDestroyWindow(window); // Release window**

**glfwTerminate(); // Terminate GLFW**

**return(0);**

**}**

**obj.c:**

#include "obj.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

/// Читает следующий float в строке

/// Возвращает количество прочитанных символов и сохраняет результат в v

int read\_float(const char\*str, float\*v) {

const char\*end;

const char\*orig = str; // Запомним исходную позицию

// Пропустить все символы, пока не встретятся символы -, +,., Null, новая строка или цифра

for (; \*str != '-' && \*str != '+' && \*str != '.' && \*str != '\0' && \*str != '\n' && (\*str < '0' || \*str > '9'); ++str);

if (\*str == '\0' || \*str == '\n') // если встречается ноль или новая строка

return -1; // Ни один номер не был прочитан

\*v = strtof(str, &end); // Читаем float

return end - orig; // Возвращаем разницу между текущей позицией и оригиналом

}

/// Читает следующий int в строке

/// Возвращает количество прочитанных символов и сохраняет результат в v

int read\_int(const char\*str, int\*v) {

const char\*end;

const char\*orig = str; // Запомним исходную позицию

// Пропускаем все символы до -, +, ноль, новая строка или цифра

for (; \*str != '-' && \*str != '+' && \*str != '\0' && \*str != '\n' && (\*str < '0' || \*str > '9'); ++str);

if (\*str == '\0' || \*str == '\n') // если встречается ноль или новая строка

return -1; // Ни один номер не был прочитан

\*v = strtoll(str, &end, 10); // читаем int

return end - orig; // Возвращаем разницу между текущей позицией и оригиналом

}

/// Читает следующую .obj грань вершины в строке

/// Возвращает количество прочитанных символов и сохраняет результат в v

int read\_face\_vert(const char\*str, int\*v) {

const char\*orig = str; // Запомним исходную позицию

int r = read\_int(str, v); // Первый int - это действительный индекс вершины

if (r == -1) // если int не был прочитан = нет лицевой вершины

return -1;

str += r; // Перемещаем строку после первого int

// пропустить перезапуск

for (; \*str == '/' || (\*str >= '0' && \*str <= '9'); ++str);

return str - orig; // Возвращаем разницу между текущей позицией и оригиналом

}

/// Читает .obj модель из файла

/// Возвращает код ошибки и сохраняет результат в pObj

int read\_obj(const char\*path, obj\*pObj) {

FILE\*file;

int err;

obj o = { 0, 0, 0, 0 };// Инициализируем пустую модель

size\_t size;

char\*contents;

int eol = 0;

err = fopen\_s(&file, path, "r"); // Пробуем открыть файл для чтения

if (err)

return err;

fseek(file, 0, SEEK\_END); // Получить размер файла

size = ftell(file);

fseek(file, 0, SEEK\_SET); // Сброс указателя на начало

contents = calloc(size + 1, 1); // резервируем буфер достаточно большой

fread(contents, 1, size, file); // Чтение содержимого файла в него

fclose(file); // Закрыть файл .obj

// Анализируем каждый символ

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

if (eol) { // We're in 'skip line' mode

if (contents[i] == '\n') // Если встречается новая строка

eol = 0; // Exit eol mode

continue;// Пропустить все символы до тех пор

}

// Если объявление вершины выполнено

if (contents[i] == 'v' && contents[i + 1] == ' ') {

o.vc++; // Увеличить количество вершин

eol = 1;

}

// Если объявление лицевой выполнено

if (contents[i] == 'f' && contents[i + 1] == ' ') {

o.fc++; // Увеличить количество лицевых

eol = 1;

}

}

// Выделить массивы

o.f = malloc(sizeof(obj\_face)\*o.fc);

o.v = malloc(sizeof(obj\_vertex)\*o.vc);

o.fc = 0; // Сброс размеров для использования в качестве счетчиков

o.vc = 0;

eol = 0;

// Итерациуем по строке снова

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

if (eol) {

if (contents[i] == '\n')

eol = 0;

continue;

}

// Объявление вершины выполнено

if (contents[i] == 'v' && contents[i + 1] == ' ') {

i += 2; // Пропустить объявление

i += read\_float(&contents[i], &o.v[o.vc].x); // Читаем три floats

i += read\_float(&contents[i], &o.v[o.vc].y);

i += read\_float(&contents[i], &o.v[o.vc].z);

--i; // Вернемся на один символ, чтобы не пропустить новую строку

++o.vc; // Увеличить счетчик вершин

eol = 1; // Пропустить остаток строки

}

// Объявление лицевой выполнено

else if (contents[i] == 'f' && contents[i + 1] == ' ') {

i += 2; // Пропустить объявление

int orig = i; // Запомнить позицию

int off;

int t;

o.f[o.fc].count = 0;

// Подсчитать количество вершин в грани

do {

// Чтение лицевых вершин

off = read\_face\_vert(&contents[i], &t);

if (off != -1)// Если найдена грань вершины

o.f[o.fc].count++; // Увеличить счетчик

i += off;

} while (off != -1); // пока не найдена грань вершины

i = orig; // Сброс позиции строки

// Выделить массив вершин

o.f[o.fc].v = malloc(sizeof(obj\_vertex)\*o.f[o.fc].count);

o.f[o.fc].count = 0;

// Чтение лицевых вершин

do {

// Чтение лицевой вершины

off = read\_face\_vert(&contents[i], &t);

if (off != -1) // Если найдена грань вершины

// Сохраняем индекс вершины

o.f[o.fc].v[o.f[o.fc].count++] = t-1;

i += off;

} while (off != -1); // пока не найдена грань вершины

++o.fc; // Увеличить количество лицевых

eol = 1; // Пропустить остаток строки

}

}

\*pObj = o;

return 0;

}

/// освобождает память от .obj модели

void free\_obj(obj o) {

for (int i = 0; i < o.fc; ++i) // Для каждой лицевой

free(o.f[i].v); // Освободить массив индексов вершин

free(o.f); // Массив свободных лицевых

free(o.v); // Свободный массив вершин

}

/// Clone .obj model

obj obj\_clone(obj o) {

obj r;

r.fc = o.fc; // копируем размеры

r.vc = o.vc;

r.f = malloc(sizeof(obj\_face)\*r.fc); // Выделяем достаточно памяти

r.v = malloc(sizeof(obj\_vertex)\*r.vc);

for (int i = 0; i < r.fc; ++i) {

r.f[i].count = o.f[i].count; // Копируем количество индексов лицевых

r.f[i].v = malloc(sizeof(int)\*r.f[i].count); // Выделим память для индексов

memcpy(r.f[i].v, o.f[i].v, sizeof(int) \* r.f[i].count); // Копируем индексы

}

memcpy(r.v, o.v, sizeof(obj\_vertex) \* r.vc); // Копируем вершины

return r;

}

/// Определяем, нуждается ли модель в триангуляции

int obj\_need\_triangulation(obj o) {

for (int i = 0; i < o.fc; ++i)

if (o.f[i].count != 3)

return 1;

return 0;

}

/// Триангуляция граней модели

obj obj\_triangulate(obj o) {

obj r;

r.vc = o.vc;

r.v = malloc(sizeof(obj\_vertex)\*r.vc); // Распределяем и копируем вершины

memcpy(r.v, o.v, sizeof(obj\_vertex)\*r.vc);

r.fc = o.fc;

for (int i = 0; i < o.fc; ++i)

r.fc += o.f[i].count - 3; // должно добавить достаточно для всех треугольных граней

r.f = malloc(sizeof(obj\_face)\*r.fc); // Выделить массив граней

r.fc = 0; // Сбросить сумму для использования в качестве счетчика

for (int i = 0; i < o.fc; ++i) {// Заполнить массив граней

int c = o.f[i].count; // Получить количество вершин лица

if (c == 3) { // Если это уже треугольник

r.f[r.fc++] = o.f[i]; // Не требуется никаких действий. Просто копируем лицевые

r.f[r.fc].count = c;

r.f[r.fc].v = malloc(sizeof(obj\_vertex)\*c);

memcpy(r.f[r.fc].v, o.f[i].v, sizeof(obj\_vertex)\*c);

}

else { // Триангуляционнуем лицевую

int v = 1; // вершина смещения триангуляции

while (c >= 3) { // Пока лицевая не треугольник

obj\_face f; // Создать новую грань треугольника

f.v = malloc(sizeof(int) \* 3);

f.count = 3;

f.v[0] = o.f[i].v[0]; // Соединяем все вершины с нулевым индексом

f.v[1] = o.f[i].v[v];

f.v[2] = o.f[i].v[v + 1];

r.f[r.fc++] = f; // запомнить новую лицевую

v++; //Увеличить смещение

--c; // Уменьшить количество вершин

}

}

}

return r;

}

**settings\_window.c:**

#include "settings\_window.h"

#include <tchar.h>

/// Дополнительная структура для данных обратного вызова

typedef struct {

settings\_window\_click\_callback callback;

LPARAM lParam;

} WIN\_USERDATA;

/// Процедура главного окна

LRESULT CALLBACK wnd\_proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

if (uMsg == WM\_CREATE) {

LPCREATESTRUCT create = lParam; // Извлечение параметров создания

SetWindowLongPtr(hWnd, GWLP\_USERDATA, create->lpCreateParams);// Сохраняем параметр в userdata

}

if (uMsg == WM\_COMMAND && HIWORD(wParam) == BN\_CLICKED) { // Кнопка нажата

WIN\_USERDATA\*ud = GetWindowLongPtr(hWnd, GWLP\_USERDATA); // Извлечение обратного вызова из пользовательских данных

if (ud && ud->callback) // Проверяем, установлен ли он

ud->callback(hWnd, lParam, ud->lParam);// Вызов обратного вызова

return 0;

}

return DefWindowProc(hWnd, uMsg, wParam, lParam);

}

/// Creates window of size with callback

HWND create\_window(settings\_window\_click\_callback callback, LPARAM lParam, int w, int h) {

HINSTANCE hInstance = GetWindowLongPtr(GetConsoleWindow(), GWLP\_HINSTANCE); // Извлечение позиции в окне консоли

LPCTSTR class\_name = \_T("class\_name");

WNDCLASS class; // Создать класс окна по умолчанию

class.style = CS\_HREDRAW | CS\_VREDRAW;

class.lpfnWndProc = wnd\_proc;

class.cbClsExtra = 0;

class.cbWndExtra = 0;

class.hInstance = hInstance;

class.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI\_APPLICATION));

class.hCursor = LoadCursor(NULL, IDC\_ARROW);

class.hbrBackground = COLOR\_WINDOW + 1;

class.lpszMenuName = NULL;

class.lpszClassName = class\_name;

RegisterClass(&class); // Регистрация окна класса

WIN\_USERDATA\*ud = malloc(sizeof(WIN\_USERDATA)); // Создать структуру данных обратного вызова

ud->callback = callback;

ud->lParam = lParam;

HWND hWnd = CreateWindow( // Создать окно

class\_name,

\_T(""),

WS\_SYSMENU | WS\_CAPTION | WS\_VISIBLE,

CW\_USEDEFAULT, CW\_USEDEFAULT,

w, h,

NULL, NULL,

hInstance, ud

);

return hWnd; // Возвращаем свой дескриптор

}

/// Добавляет элемент управления кнопки в окно

HWND add\_button(HWND hWnd, int x, int y, int w, int h, LPCTSTR text) {

return CreateWindow( // Создать кнопку управления и установить в качестве дочернего окна

\_T("Button"), text,

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

x, y, w, h, hWnd, NULL,

GetWindowLongPtr(hWnd, GWLP\_HINSTANCE), NULL

);

}

/// Добавляет текстовый элемент управления в окно

HWND add\_text(HWND hWnd, int x, int y, int w, int h, LPCTSTR text) {

return CreateWindow( // Создать текстовый элемент управления и установить в качестве дочернего окна

\_T("Static"), text,

WS\_VISIBLE | WS\_CHILD | SS\_CENTER,

x, y, w, h, hWnd, NULL,

GetWindowLongPtr(hWnd, GWLP\_HINSTANCE), NULL

);

}

**Результат работы программы**:



 