**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

**Лабораторная работа №5**

по дисциплине: Компьютерная графика

тема: «Алгоритмы удаления невидимых поверхностей»

Выполнил: ст. группы ПВ-223

Пахомов Владислав Андреевич

Проверили:

ст. пр. Осипов Олег Васильевич

Белгород 2024 г.

**Лабораторная работа №5  
Алгоритмы удаления невидимых поверхностей  
Вариант 8**

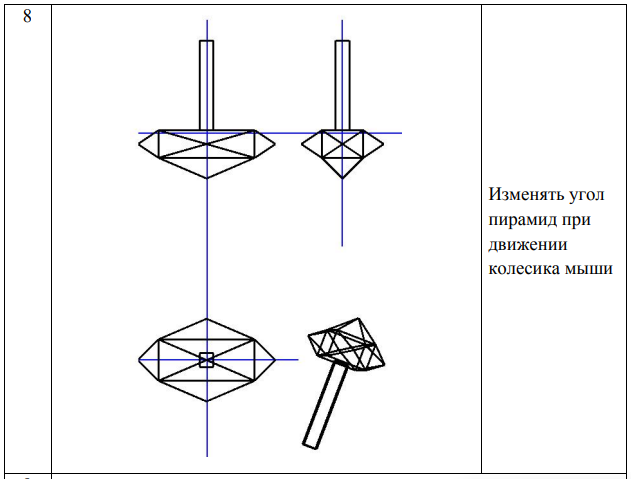
**Цель работы:** изучить алгоритмы удаления невидимых поверхностей и создать

программу для визуализации объёмной трёхмерной модели с закрашенными гранями.

**Задания для выполнения к работе:**

1. Разработать алгоритм и составить программу для построения на экране трёхмерной модели с закрашенными гранями в соответствии с номером варианта лабораторной работы №4

**Задание:**



Исходный код:

**Color.h**

#pragma once

#include <math.h>

// Cтруктура для задания цвета

struct COLOR

{

unsigned char RED; // Компонента красного цвета

unsigned char GREEN; // Компонента зелёного цвета

unsigned char BLUE; // Компонента синего цвета

unsigned char ALPHA; // Прозрачность (альфа канал)

COLOR(int red, int green, int blue, int alpha = 255)

: RED(red), GREEN(green), BLUE(blue), ALPHA(alpha) { }

} ;

// Пиксель в буфере кадра

struct PIXEL

{

unsigned char RED; // Компонента красного цвета

unsigned char GREEN; // Компонента зелёного цвета

unsigned char BLUE; // Компонента синего цвета

float Z; // Глубина пикселя

PIXEL() : RED(0), GREEN(0), BLUE(0), Z(INFINITY) { }

} ;

**Frame.h**

#ifndef FRAME\_H

#define FRAME\_H

#include <math.h>

#include <algorithm>

#include "Matrix.h"

#include "Color.h"

#include "Perspective.h"

#include "Shaders.h"

#define INSIDE 0 // 0000

#define LEFT 1 // 000

#define RIGHT 2 // 0010

#define BOTTOM 4 // 0100

#define TOP 8 // 1000

#define SHOW\_POLYGON 0b01

#define SHOW\_GRID 0b10

// Cтруктура для задания цвета

typedef struct HSVCOLOR

{

double H; // Компонента красного цвета

double S; // Компонента зелёного цвета

double V; // Компонента синего цвета

unsigned char ALPHA; // Прозрачность (альфа канал)

HSVCOLOR() : H(0), S(0), V(0), ALPHA(255) { }

HSVCOLOR(double hue, double saturation, double value, int alpha = 255)

: H(hue), S(saturation), V(value), ALPHA(alpha)

{

if (hue < 0) H = 0;

else if (hue > 360) H = 360;

if (saturation < 0) S = 0;

else if (saturation > 1) S = 1;

if (value < 0) value = 0;

else if (value > 1) V = 1;

if (alpha < 0) ALPHA = 0;

else if (alpha > 255) ALPHA = 255;

}

COLOR convertToRgb() {

int hi = int(floor(H / 60)) % 6;

double f = H / 60 - floor(H / 60);

int copyV = V \* 255;

int v = (int)(copyV);

int p = (int)(copyV \* (1 - S));

int q = (int)(copyV \* (1 - f \* S));

int t = (int)(copyV \* (1 - (1 - f) \* S));

if (hi == 0)

return { v, t, p, ALPHA };

if (hi == 1)

return { q, v, p, ALPHA };

else if (hi == 2)

return { p, v, t, ALPHA };

else if (hi == 3)

return { p, q, v, ALPHA };

else if (hi == 4)

return { t, p, v, ALPHA };

return { v, p, q, ALPHA };

}

} HSVCOLOR;

// Буфер кадра

class Frame

{

// Указатель на массив пикселей

// Буфер кадра будет представлять собой матрицу, которая располагается в памяти в виде непрерывного блока

PIXEL\* pixels;

public:

// Размеры буфера кадра

int width, height;

Matrix transform;

Perspective perspective;

Frame(int \_width, int \_height, Perspective \_perspective = Perspective::FRUSTUM, Matrix \_transform = Matrix()) :

width(\_width), height(\_height), perspective(\_perspective), transform(\_transform)

{

int size = width \* height;

// Создание буфера кадра в виде непрерывной матрицы пикселей

pixels = new PIXEL[size];

}

// Задаёт цвет color пикселю с координатами (x, y)

void SetPixel(int x, int y, COLOR color)

{

PIXEL\* pixel = pixels + (size\_t)y \* width + x; // Находим нужный пиксель в матрице пикселей

pixel->RED = color.RED;

pixel->GREEN = color.GREEN;

pixel->BLUE = color.BLUE;

}

// Возвращает цвет пикселя с координатами (x, y)

COLOR GetPixel(int x, int y)

{

PIXEL\* pixel = pixels + (size\_t)y \* width + x; // Находим нужный пиксель в матрице пикселей

return COLOR(pixel->RED, pixel->GREEN, pixel->BLUE);

}

int getCohenSutherland(int x, int y) {

int y\_max = height - 1;

int y\_min = 0;

int x\_max = width - 1;

int x\_min = 0;

// initialized as being inside

int code = INSIDE;

if (x < x\_min) // to the left of rectangle

code |= LEFT;

else if (x > x\_max) // to the right of rectangle

code |= RIGHT;

if (y < y\_min) // below the rectangle

code |= BOTTOM;

else if (y > y\_max) // above the rectangle

code |= TOP;

return code;

}

void cohenSutherlandClip(int x1, int y1,

int x2, int y2, bool\* accepted,

int\* resX1, int\* resY1, int\* resX2, int\* resY2)

{

int y\_max = height - 1;

int y\_min = 0;

int x\_max = width - 1;

int x\_min = 0;

int code1 = getCohenSutherland(x1, y1);

int code2 = getCohenSutherland(x2, y2);

bool accept = false;

while (true) {

if ((code1 == 0) && (code2 == 0)) {

accept = true;

break;

}

else if (code1 & code2) {

break;

}

else {

int code\_out;

int x, y;

if (code1 != 0)

code\_out = code1;

else

code\_out = code2;

if (code\_out & TOP) {

x = x1 + (x2 - x1) \* (y\_max - y1) / (y2 - y1);

y = y\_max;

}

else if (code\_out & BOTTOM) {

x = x1 + (x2 - x1) \* (y\_min - y1) / (y2 - y1);

y = y\_min;

}

else if (code\_out & RIGHT) {

y = y1 + (y2 - y1) \* (x\_max - x1) / (x2 - x1);

x = x\_max;

}

else if (code\_out & LEFT) {

y = y1 + (y2 - y1) \* (x\_min - x1) / (x2 - x1);

x = x\_min;

}

if (code\_out == code1) {

x1 = x;

y1 = y;

code1 = getCohenSutherland(x1, y1);

}

else {

x2 = x;

y2 = y;

code2 = getCohenSutherland(x2, y2);

}

}

}

if (accept) {

\*accepted = true;

\*resX1 = x1;

\*resY1 = y1;

\*resX2 = x2;

\*resY2 = y2;

}

else {

\*accepted = false;

}

}

// Рисование отрезка

void DrawLine(int x1, int y1, int x2, int y2, COLOR color)

{

bool shouldDraw = false;

int resX1, resY1, resX2, resY2;

cohenSutherlandClip(x1, y1, x2, y2, &shouldDraw, &resX1, &resY1, &resX2, &resY2);

if (!shouldDraw) return;

x1 = resX1;

y1 = resY1;

x2 = resX2;

y2 = resY2;

PIXEL\* pixel;

int dy = y2 - y1, dx = x2 - x1;

if (dx == 0 && dy == 0)

{

pixel = pixels + (size\_t)y1 \* width + x1;

pixel->RED = color.RED;

pixel->GREEN = color.GREEN;

pixel->BLUE = color.BLUE;

pixel->Z = -1;

return;

}

if (abs(dx) > abs(dy))

{

if (x2 < x1)

{

// Обмен местами точек (x1, y1) и (x2, y2)

std::swap(x1, x2);

std::swap(y1, y2);

dx = -dx; dy = -dy;

}

int y = y1;

int sign\_factor = dy < 0 ? 1 : -1;

int sumd = -2 \* (y - y1) \* dx + sign\_factor \* dx;

for (int x = x1; x <= x2; x++)

{

if (sign\_factor \* sumd < 0) {

y -= sign\_factor;

sumd += sign\_factor \* dx;

}

sumd += dy;

pixel = pixels + (size\_t)y \* width + x;

pixel->RED = color.RED;

pixel->GREEN = color.GREEN;

pixel->BLUE = color.BLUE;

pixel->Z = -1;

}

}

else

{

if (y2 < y1)

{

// Обмен местами точек (x1, y1) и (x2, y2)

std::swap(x1, x2);

std::swap(y1, y2);

dx = -dx; dy = -dy;

}

int x = x1;

int sign\_factor = dx > 0 ? 1 : -1;

int sumd = 2 \* (x - x1) \* dy + sign\_factor \* dy;

for (int y = y1; y <= y2; y++)

{

if (sign\_factor \* sumd < 0) {

x += sign\_factor;

sumd += sign\_factor \* dy;

}

sumd -= dx;

pixel = pixels + (size\_t)y \* width + x;

pixel->RED = color.RED;

pixel->GREEN = color.GREEN;

pixel->BLUE = color.BLUE;

pixel->Z = -1.1;

}

}

}

void Triangle(float x0, float y0, float z0, float w0, float x1, float y1, float z1, float w1, float x2, float y2, float z2, float w2, BaseShader\* shader, char drawMode = SHOW\_GRID | SHOW\_POLYGON, COLOR gridColor = { 255, 255, 255, 0 })

{

if (!drawMode) return;

if (drawMode & SHOW\_GRID) {

DrawLine(x0, y0, x1, y1, gridColor);

DrawLine(x1, y1, x2, y2, gridColor);

DrawLine(x0, y0, x2, y2, gridColor);

}

if (!(drawMode & SHOW\_POLYGON))

return;

bool swap\_y1\_y0\_first = false;

bool swap\_y2\_y1 = false;

bool swap\_y1\_y0\_second = false;

// Отсортируем точки таким образом, чтобы выполнилось условие: y0 < y1 < y2

if (y1 < y0)

{

std::swap(x1, x0);

std::swap(y1, y0);

std::swap(z1, z0);

std::swap(w1, w0);

swap\_y1\_y0\_first = true;

}

if (y2 < y1)

{

std::swap(x2, x1);

std::swap(y2, y1);

std::swap(z2, z1);

std::swap(w2, w1);

swap\_y2\_y1 = true;

}

if (y1 < y0)

{

std::swap(x1, x0);

std::swap(y1, y0);

std::swap(z1, z0);

std::swap(w1, w0);

swap\_y1\_y0\_second = true;

}

// Определяем номера строк пикселей, в которых располагаются точки треугольника

int Y0 = (int) (y0 + 0.5f);

int Y1 = (int) (y1 + 0.5f);

int Y2 = (int) (y2 + 0.5f);

// Отсечение невидимой части треугольника

if (Y0 < 0) Y0 = 0;

else if (Y0 > height) Y0 = height;

if (Y1 < 0) Y1 = 0;

else if (Y1 > height) Y1 = height;

if (Y2 < 0) Y2 = 0;

else if (Y2 > height) Y2 = height;

double S = (y1 - y2) \* (x0 - x2) + (x2 - x1) \* (y0 - y2); // Площадь треугольника

// Рисование верхней части треугольника

for (int Y = Y0; Y < Y1; Y++)

{

double y = Y + 0.5; // Координата y середины пикселя

// Вычисление координат граничных пикселей

int X0 = (int) ((y - y0) / (y1 - y0) \* (x1 - x0) + x0 + 0.5f);

int X1 = (int) ((y - y0) / (y2 - y0) \* (x2 - x0) + x0 + 0.5f);

if (X0 > X1) std::swap(X0, X1); // Сортировка пикселей

if (X0 < 0) X0 = 0; // Отсечение невидимых пикселей в строке y

if (X1 > width) X1 = width;

for (int X = X0; X < X1; X++)

{

double x = X + 0.5; // Координата x середины пикселя

// Середина пикселя имеет координаты (x, y)

// Вычислим барицентрические координаты этого пикселя

double h0 = ((y1 - y2) \* (x - x2) + (x2 - x1) \* (y - y2)) / S;

double h1 = ((y2 - y0) \* (x - x2) + (x0 - x2) \* (y - y2)) / S;

double h2 = ((y0 - y1) \* (x - x1) + (x1 - x0) \* (y - y1)) / S;

double ih0 = h0 / w0;

double ih1 = h1 / w1;

double ih2 = h2 / w2;

double NZ = 1.0 / (ih0 + ih1 + ih2);

h0 = ih0 \* NZ;

h1 = ih1 \* NZ;

h2 = ih2 \* NZ;

float Z = h0 \* z0 + h1 \* z1 + h2 \* z2; // Глубина пикселя

// Определение глубины точки в экранной системе координат

PIXEL\* pixel = pixels + (size\_t)Y \* width + X; // Вычислим адрес пикселя (Y, X) в матрице пикселей pixels

// pixel->Z - глубина пикселя, которая уже записана в буфер кадра

// Если текущий пиксель находится ближе того пикселя, который уже записан в буфере кадра

if (Z > -1 && Z < 1/\* && Z < pixel->Z\*/)

{ // то обновляем пиксель в буфере кадра

//float zmax = 0.8, zmin = 0.1;

//color = COLOR(255 - (Z - zmin) / (zmax - zmin) \* 255, 100, 100);

//color = COLOR((2.5 - Z) / 4 \* 255, 0, 0);

if (swap\_y1\_y0\_second) {

std::swap(h0, h1);

}

if (swap\_y2\_y1) {

std::swap(h1, h2);

}

if (swap\_y1\_y0\_first) {

std::swap(h0, h1);

}

auto color = shader->main(shader->getVertexData(h0, h1, h2));

pixel->RED += (color.ALPHA / 255.0) \* (color.RED - pixel->RED);

pixel->GREEN += (color.ALPHA / 255.0) \* (color.GREEN - pixel->GREEN);

pixel->BLUE += (color.ALPHA / 255.0) \* (color.BLUE - pixel->BLUE);

pixel->Z = Z;

}

}

}

// Рисование нижней части треугольника

for (int Y = Y1; Y < Y2; Y++)

{

double y = Y + 0.5; // Координата y середины пикселя

// Вычисление координат граничных пикселей

int X0 = (int)((y - y1) / (y2 - y1) \* (x2 - x1) + x1 + 0.5f);

int X1 = (int)((y - y0) / (y2 - y0) \* (x2 - x0) + x0 + 0.5f);

if (X0 > X1) std::swap(X0, X1); // Сортировка пикселей

if (X0 < 0) X0 = 0; // Отсечение невидимых пикселей в строке y

if (X1 > width) X1 = width;

for (int X = X0; X < X1; X++)

{

double x = X + 0.5; // Координата x середины пикселя

// Середина пикселя имеет координаты (x, y)

// Вычислим барицентрические координаты этого пикселя

double h0 = ((y1 - y2) \* (x - x2) + (x2 - x1) \* (y - y2)) / S;

double h1 = ((y2 - y0) \* (x - x2) + (x0 - x2) \* (y - y2)) / S;

double h2 = ((y0 - y1) \* (x - x1) + (x1 - x0) \* (y - y1)) / S;

double ih0 = h0 / w0;

double ih1 = h1 / w1;

double ih2 = h2 / w2;

double NZ = 1.0 / (ih0 + ih1 + ih2);

h0 = ih0 \* NZ;

h1 = ih1 \* NZ;

h2 = ih2 \* NZ;

float Z = h0 \* z0 + h1 \* z1 + h2 \* z2; // Глубина пикселя

// Определение глубины точки в экранной системе координат

// Если текущий пиксель находится ближе того пикселя, который уже записан в буфере кадра

PIXEL\* pixel = pixels + (size\_t)Y \* width + X;

if (Z > -1 && Z < 1/\* && Z < pixel->Z\*/)

{

if (swap\_y1\_y0\_second) {

std::swap(h0, h1);

}

if (swap\_y2\_y1) {

std::swap(h1, h2);

}

if (swap\_y1\_y0\_first) {

std::swap(h0, h1);

}

//float zmax = 0.8, zmin = 0.1;

//color = COLOR(255 - (Z - zmin) / (zmax - zmin) \* 255, 100, 100);

//color = COLOR((2.5 - Z) / 4 \* 255, 0, 0);

auto color = shader->main(shader->getVertexData(h0, h1, h2));

pixel->RED += (color.ALPHA / 255.0) \* (color.RED - pixel->RED);

pixel->GREEN += (color.ALPHA / 255.0) \* (color.GREEN - pixel->GREEN);

pixel->BLUE += (color.ALPHA / 255.0) \* (color.BLUE - pixel->BLUE);

pixel->Z = Z;

}

}

}

}

void Quad(float x0, float y0, float z0, float w0, float x1, float y1, float z1, float w1, float x2, float y2, float z2, float w2, float x3, float y3, float z3, float w3, BaseShader\* shader)

{

Triangle(x0, y0, z0, w0, x1, y1, z1, w1, x2, y2, z2, w2, shader);

Triangle(x2, y2, z2, w2, x3, y3, z3, w3, x0, y0, z0, w0, shader);

}

~Frame(void)

{

delete []pixels;

}

};

#endif // FRAME\_H

**Model.h**

#pragma once

#include <vector>

#include <tuple>

#include "Vector.h"

#include "Shaders.h"

// Возвращает точки объекта

const std::vector<Vector> get\_points(float scale\_factor = 1.) {

std::vector<Vector> result = {

// Вершина рукоятки

{-0.05, 1, 0.05}, // 0

{0.05, 1, 0.05}, // 1

{-0.05, 1, -0.05},// 2

{0.05, 1, -0.05}, // 3

// Нижняя часть рукоятки

{-0.05, 0.25, 0.05}, // 4

{0.05, 0.25, 0.05}, // 5

{-0.05, 0.25, -0.05},// 6

{0.05, 0.25, -0.05}, // 7

// Нижняя часть молота

{-0.4, 0.25, 0.2}, // 8

{0.4, 0.25, 0.2}, // 9

{-0.4, 0.25, -0.2},// 10

{0.4, 0.25, -0.2}, // 11

// Боковые шипы молота

{-(0.4f + scale\_factor), 0.17, 0}, // 12

{(0.4f + scale\_factor), 0.17, 0}, // 13

{0, 0.17, -(0.2f + scale\_factor)}, // 14

{0, 0.17, (0.2f + scale\_factor)}, // 15

// Соединители боковых шипов

{-0.4, 0.09, 0.2}, // 16

{0.4, 0.09, 0.2}, // 17

{-0.4, 0.09, -0.2}, // 18

{0.4, 0.09, -0.2}, // 19

// Нижний шип

{0, 0.09f - scale\_factor, 0} // 20

};

return result;

}

// Задаёт индексы точек для формирования полигонов

std::vector<std::tuple<int, int, int>> polygons = {

// Вершина рукоятки

{0, 1, 2}, // 0

{3, 1, 2}, // 1

// Стенки рукоятки

{0, 1, 5}, // 2

{1, 5, 7}, // 3

{1, 3, 7}, // 4

{3, 7, 6}, // 5

{3, 2, 6}, // 6

{2, 6, 4}, // 7

{2, 0, 4}, // 8

{0, 4, 5}, // 9

// Нижняя часть рукоятки

{5, 4, 9},

{4, 9, 8},

{6, 4, 8},

{6, 8, 10},

{6, 7, 10},

{7, 10, 11},

{7, 11, 5},

{11, 5, 9},

// Шипы молота

{8, 10, 12},

{8, 9, 15},

{9, 11, 13},

{11, 10, 14},

{16, 18, 12},

{16, 17, 15},

{17, 19, 13},

{19, 18, 14},

{11, 19, 13},

{11, 19, 14},

{10, 18, 14},

{10, 18, 12},

{8, 16, 12},

{8, 16, 15},

{9, 17, 15},

{9, 17, 13},

// Нижний шип

{19, 18, 20},

{18, 16, 20},

{16, 17, 20},

{17, 19, 20}

};

std::vector<BaseShader\*> materials = {

new DottedShader({0, 0, 0}, {1, 0, 0}, {0, 0, 1}),

new DottedShader({1, 0, 1}, {1, 0, 0}, {0, 0, 1}),

new ColorShader({98, 82, 154, 120}),

new ColorShader({142, 132, 211, 120}),

new ColorShader({97, 103, 222, 120}),

new ColorShader({70, 93, 36, 120}),

new ColorShader({62, 122, 190, 120}),

new ColorShader({83, 57, 98, 120}),

new ColorShader({148, 167, 154, 120}),

new ColorShader({111, 115, 224, 120}),

new ColorShader({175, 218, 20}),

new ColorShader({168, 178, 84}),

new ColorShader({40, 168, 153}),

new ColorShader({132, 46, 214}),

new ColorShader({105, 215, 86}),

new ColorShader({109, 183, 95}),

new ColorShader({64, 121, 201}),

new ColorShader({141, 185, 103}),

new ColorShader({240, 240, 0, 255}),

new ColorShader({240, 0, 240, 255}),

new ColorShader({0, 240, 240, 255}),

new ColorShader({240, 240, 240, 255}),

new ColorShader({200, 200, 0, 200}),

new ColorShader({200, 0, 200, 200}),

new ColorShader({0, 200, 200, 200}),

new ColorShader({200, 200, 200, 200}),

new ColorShader({125, 240, 0, 255}),

new ColorShader({240, 0, 125, 255}),

new ColorShader({0, 125, 240, 255}),

new ColorShader({240, 125, 125, 255}),

new ColorShader({240, 64, 0, 200}),

new ColorShader({64, 0, 240, 200}),

new ColorShader({0, 240, 64, 200}),

new ColorShader({64, 240, 240, 200}),

new ColorShader({40, 40, 40, 240}),

new ColorShader({40, 40, 40, 240}),

new ColorShader({40, 40, 40, 240}),

new ColorShader({40, 40, 40, 240})

};

void regen\_transparencies() {

for (int i = 2; i < materials.size(); i++) {

((ColorShader\*)materials[i])->base.RED = rand() % 256;

((ColorShader\*)materials[i])->base.GREEN = rand() % 256;

((ColorShader\*)materials[i])->base.BLUE = rand() % 256;

if (rand() & 1) {

((ColorShader\*)materials[i])->base.ALPHA = 20 + rand() % 120;

}

else {

((ColorShader\*)materials[i])->base.ALPHA = 255;

}

}

}

**Painter.h**

#ifndef PAINTER\_H

#define PAINTER\_H

#include "Frame.h"

#include "Vector.h"

#include "Model.h"

#include <optional>

#include <tuple>

// Время от начала запуска программы

float time = 0;

float scale = .1;

float x\_offset = 0;

float y\_offset = 0;

float z\_offset = -2;

float x\_rot = 0;

float y\_rot = 0;

float z\_rot = 0;

float fig\_scale = 2.5;

int draw\_mode = 1;

Perspective currentPerspective = static\_cast<Perspective>(0);

#define EPS 0.0000000001

std::tuple<bool, Vector> intersect\_points\_2d(Vector A, Vector B, Vector C, Vector D) {

float dABx = std::abs(B.x - A.x);

float dCDx = std::abs(C.x - D.x);

float dABy = std::abs(B.y - A.y);

float dCDy = std::abs(C.y - D.y);

Vector O = {};

if (dABx < EPS && dCDy < EPS) {

O.x = A.x;

O.y = C.y;

return { true, O };

}

else if (dABy < EPS && dCDx < EPS) {

O.x = C.x;

O.y = A.y;

return { true, O };

}

if (dABx < EPS && dCDx < EPS || dABy < EPS && dCDy < EPS) {

// Прямые параллельны

return { false, O };

}

else if (dABx < EPS && dABy < EPS || dCDx < EPS && dCDy < EPS) {

// Передана точка

return { false, O };

}

else if (dABx < EPS) {

// AB параллельна абсциссе

O.x = A.x;

O.y = (O.x - C.x) \* (D.y - C.y) / (D.x - C.x) + C.y;

return {

O.x < max(C.x, D.x) && O.x > min(C.x, D.x) &&

O.y < max(A.y, B.y) && O.y > min(A.y, B.y) &&

O.y < max(C.y, D.y) && O.y > min(C.y, D.y), O };

}

else if (dABy < EPS) {

// AB параллельна ординате

O.y = A.y;

O.x = (O.y - C.y) \* (D.x - C.x) / (D.y - C.y) + C.x;

return { O.x < max(A.x, B.x) && O.x > min(A.x, B.x) &&

O.x < max(C.x, D.x) && O.x > min(C.x, D.x) &&

O.y < max(C.y, D.y) && O.y > min(C.y, D.y), O };

}

else if (dCDy < EPS) {

O.y = C.y;

O.x = (O.y - A.y) \* (B.x - A.x) / (B.y - A.y) + A.x;

return { O.x < max(A.x, B.x) && O.x > min(A.x, B.x) &&

O.x < max(C.x, D.x) && O.x > min(C.x, D.x) &&

O.y < max(A.y, B.y) && O.y > min(A.y, B.y), O };

}

else if (dCDx < EPS) {

O.x = C.x;

O.y = (O.x - A.x) \* (B.y - A.y) / (B.x - A.x) + A.y;

return { O.x < max(A.x, B.x) && O.x > min(A.x, B.x) &&

O.y < max(A.y, B.y) && O.y > min(A.y, B.y) &&

O.y < max(C.y, D.y) && O.y > min(C.y, D.y), O };

}

else {

O.y = ((D.y - C.y) \* (A.x - A.y \* (B.x - A.x) / (B.y - A.y) - C.x) / (D.x - C.x) + C.y) / (1 - (D.y - C.y) \* (B.x - A.x) / ((D.x - C.x) \* (B.y - A.y)));

O.x = (O.y - C.y) \* (D.x - C.x) / (D.y - C.y) + C.x;

return { O.x < max(A.x, B.x) && O.x > min(A.x, B.x) &&

O.x < max(C.x, D.x) && O.x > min(C.x, D.x) &&

O.y < max(A.y, B.y) && O.y > min(A.y, B.y) &&

O.y < max(C.y, D.y) && O.y > min(C.y, D.y), O };

}

}

// Возвращает 1, если AC ближе CD.

// Возвращает -1, еслир AC дальше CD.

// Возвращает 0, если отрезки не пересекаются.

int cmp\_sides(Vector A, Vector B, Vector C, Vector D) {

auto inters\_res = intersect\_points\_2d({

A.x, A.y, 0, 0

}, {

B.x, B.y, 0, 0

}, {

C.x, C.y, 0, 0

}, {

D.x, D.y, 0, 0

});

// Если точки совпадают, нужно вернуть 0, иначе всё ломается (очень плохо ломается)

if (std::abs(A.x - C.x) < EPS && std::abs(A.y - C.y) < EPS ||

std::abs(A.x - D.x) < EPS && std::abs(A.y - D.y) < EPS ||

std::abs(B.x - C.x) < EPS && std::abs(B.y - C.y) < EPS ||

std::abs(B.x - D.x) < EPS && std::abs(B.y - D.y) < EPS) return 0;

if (!std::get<0>(inters\_res)) return 0;

auto O = std::get<1>(inters\_res);

float zAB, zCD;

if (std::abs(B.x - A.x) > std::abs(B.y - A.y)) {

zAB = (O.x - A.x) \* (B.z - A.z) / (B.x - A.x) + A.z;

}

else {

zAB = (O.y - A.y) \* (B.z - A.z) / (B.y - A.y) + A.z;

}

if (std::abs(D.x - C.x) > std::abs(D.y - C.y)) {

zCD = (O.x - C.x) \* (D.z - C.z) / (D.x - C.x) + C.z;

}

else {

zCD = (O.y - C.y) \* (D.z - C.z) / (D.y - C.y) + C.z;

}

if (zAB < zCD) {

return 1;

}

else {

return -1;

}

}

bool is\_point\_inside\_polygon(Vector O, std::vector<Vector> polygon) {

if (std::abs(O.x - polygon[0].x) < EPS && std::abs(O.x - polygon[0].x) < EPS ||

std::abs(O.x - polygon[1].x) < EPS && std::abs(O.x - polygon[1].x) < EPS ||

std::abs(O.x - polygon[2].x) < EPS && std::abs(O.x - polygon[2].x) < EPS) return true;

int neg\_count = 0, pos\_count = 0;

std::vector<Vector> vectors;

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

vectors.push\_back({

polygon[i].x - O.x,

polygon[i].y - O.y,

polygon[i].z - O.z

});

}

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

Vector a = vectors[i], b = vectors[(i + 1) % polygon.size()];

float cz = a.x \* b.y - b.x \* a.y;

if (cz < 0) {

neg\_count++;

}

else {

pos\_count++;

}

}

return neg\_count == polygon.size() || pos\_count == polygon.size();

}

// Возвращает 1, если A ближе B

// Возвращает -1, если B ближе A

// Возвращает 0, если A и B не пересекаются.

int cmp\_triangles(std::vector<Vector> polygonA, std::vector<Vector> polygonB) {

Vector Ea;

Vector Eb;

bool Eb\_first = true;

bool a\_in\_b = true;

bool b\_in\_a = true;

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

Ea.x += polygonA[i].x / polygonA.size();

Ea.y += polygonA[i].y / polygonA.size();

Ea.z += polygonA[i].z / polygonA.size();

a\_in\_b &= is\_point\_inside\_polygon(polygonA[i], polygonB);

for (int j = 0; j < polygonB.size(); j++) {

auto compare\_sides = cmp\_sides(

polygonA[i], polygonA[(i + 1) % polygonA.size()],

polygonB[j], polygonB[(j + 1) % polygonB.size()]

);

if (compare\_sides) return compare\_sides;

if (!Eb\_first) continue;

Eb.x += polygonB[j].x / polygonB.size();

Eb.y += polygonB[j].y / polygonB.size();

Eb.z += polygonB[j].z / polygonB.size();

b\_in\_a &= is\_point\_inside\_polygon(polygonB[j], polygonA);

}

Eb\_first = false;

}

bool swapped = false;

if (!a\_in\_b && !b\_in\_a) return 0;

// Делаем так, чтобы b был внутри a

if (a\_in\_b) {

std::swap(Ea, Eb);

std::swap(polygonA, polygonB);

swapped = true;

}

auto P0 = polygonA[0];

auto P1 = polygonA[1];

auto P2 = polygonA[2];

float z = ((Eb.y - P0.y) \* ((P1.x - P0.x) \* (P2.z - P0.z) - (P2.x - P0.x) \* (P1.z - P0.z)) -

(Eb.x - P0.x) \* ((P1.y - P0.y) \* (P2.z - P0.z) - (P2.y - P0.y) \* (P1.z - P0.z)) ) /

((P1.x - P0.x) \* (P2.y - P0.y) - (P2.x - P0.x) \* (P1.y - P0.y)) + P0.z;

if (Eb.z < z) {

if (swapped) {

return 1;

}

else {

return -1;

}

}

else {

if (swapped) {

return -1;

}

else {

return 1;

}

}

}

// Тип проекции (перспективная или ортографическая)

class Painter

{

public:

void Draw(Frame& frame)

{

float angle = time; // Угол поворота объекта

auto A = get\_points(scale);

Matrix projection\_matrix; // Матрица проектирования

// Выбор матрицы проектирования

if (frame.perspective == Perspective::ORTHO) //Ортографическое проектирование

{

projection\_matrix = Matrix::Ortho(-2.0 \* frame.width / frame.height, 2.0 \* frame.width / frame.height, -2.0, 2.0, 1, 140.0f);

}

else if (frame.perspective == Perspective::FRUSTUM) // Перспективное проектирование

{

projection\_matrix = Matrix::Frustum(-0.5 \* frame.width / frame.height, 0.5 \* frame.width / frame.height, -0.5, 0.5, 1, 140);

}

else if (frame.perspective == Perspective::TRIMETRIC) {

projection\_matrix = Matrix::Axonometric(3.14 / 4., 3.14 / 6., -2.0 \* frame.width / frame.height, 2.0 \* frame.width / frame.height, -2.0, 2.0, 0, 140.0f) \* Matrix::Translation(0, 0, 1);

}

else if (frame.perspective == Perspective::DIMETRIC) {

projection\_matrix = Matrix::Axonometric(0.5152212, 0.45779986, -2.0 \* frame.width / frame.height, 2.0 \* frame.width / frame.height, -2.0, 2.0, 0, 140.0f) \* Matrix::Translation(0, 0, 1);

}

else if (frame.perspective == Perspective::ISOMETRIC) {

projection\_matrix = Matrix::Axonometric(0.615479708, 3.14 / 4., -2.0 \* frame.width / frame.height, 2.0 \* frame.width / frame.height, -2.0, 2.0, 0, 140.0f) \* Matrix::Translation(0, 0, 1);

}

Matrix proj\_viewport = projection\_matrix \* // Проектирование

Matrix::Viewport(0, 0, frame.width, frame.height);

Matrix general\_matrix =

frame.transform \*

proj\_viewport; // Преобразование нормализованных координат в оконные

std::vector<Vector> B(A.size());

for (int i = 0; i < A.size(); i++)

{

B[i] = A[i] \* general\_matrix;

// Преобразование однородных координат в обычные

B[i].x /= B[i].w;

B[i].y /= B[i].w;

B[i].z /= B[i].w;

B[i].w = 1;

}

std::vector<std::vector<Vector>> polygons\_vals;

std::vector<int> polygons\_indices;

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

polygons\_vals.push\_back({

B[std::get<0>(polygons[i])],

B[std::get<1>(polygons[i])],

B[std::get<2>(polygons[i])]

});

polygons\_indices.push\_back(i);

}

auto H = std::vector<std::vector<int>>(polygons.size(), std::vector<int>(polygons.size(), 0));

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

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

auto res = cmp\_triangles(polygons\_vals[i], polygons\_vals[j]);

H[i][j] = res;

H[j][i] = -res;

}

}

while (H.size() != 0) {

int index\_to\_del = 0;

int min\_ones = H.size();

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

int ones = 0;

for (int j = 0; j < H[i].size(); j++) {

if (H[i][j] == 1) {

ones++;

}

}

if (ones < min\_ones) {

min\_ones = ones;

index\_to\_del = i;

}

}

frame.Triangle(

polygons\_vals[index\_to\_del][0].x, polygons\_vals[index\_to\_del][0].y, polygons\_vals[index\_to\_del][0].z, polygons\_vals[index\_to\_del][0].w,

polygons\_vals[index\_to\_del][1].x, polygons\_vals[index\_to\_del][1].y, polygons\_vals[index\_to\_del][1].z, polygons\_vals[index\_to\_del][1].w,

polygons\_vals[index\_to\_del][2].x, polygons\_vals[index\_to\_del][2].y, polygons\_vals[index\_to\_del][2].z, polygons\_vals[index\_to\_del][2].w,

materials[polygons\_indices[index\_to\_del]], SHOW\_POLYGON);

H.erase(H.begin() + index\_to\_del);

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

H[i].erase(H[i].begin() + index\_to\_del);

}

polygons\_vals.erase(polygons\_vals.begin() + index\_to\_del);

polygons\_indices.erase(polygons\_indices.begin() + index\_to\_del);

}

/\*for (int i = 0; i < polygons.size(); i++) {

auto pointA = B[std::get<0>(polygons[i])];

auto pointB = B[std::get<1>(polygons[i])];

auto pointC = B[std::get<2>(polygons[i])];

auto polygonColor = materials[i];

frame.Triangle(pointA.x, pointA.y, pointA.z, pointA.w, pointB.x, pointB.y, pointB.z, pointB.w, pointC.x, pointC.y, pointC.z, pointC.w, polygonColor, SHOW\_GRID);

}\*/

}

};

#endif // PAINTER\_H

**Shaders.h**

#pragma once

#include "Vector.h"

#include "Color.h"

class BaseShader {

public:

BaseShader() {}

virtual COLOR main(Vector data) = 0;

virtual Vector getVertexData(double h0, double h1, double h2) = 0;

};

class CheckmateShader : public BaseShader {

Vector A, B, C;

public:

CheckmateShader(Vector A, Vector B, Vector C) : A(A), B(B), C(C) {}

COLOR main(Vector data) {

bool x\_factor = (int)(data.x \* 5) % 2;

bool z\_factor = (int)(data.z \* 5) % 2;

if ((x\_factor + z\_factor) % 2) return { 10, 10, 10 };

return { 255, 255, 255, 120 };

}

Vector getVertexData(double h0, double h1, double h2) {

Vector result = {

A.x \* h0 + B.x \* h1 + C.x \* h2,

A.y \* h0 + B.y \* h1 + C.y \* h2,

A.z \* h0 + B.z \* h1 + C.z \* h2

};

return result;

}

};

class ColorShader : public BaseShader {

public:

COLOR base;

ColorShader(COLOR base) : base(base) {}

COLOR main(Vector data) {

return base;

}

Vector getVertexData(double h0, double h1, double h2) {

return Vector();

}

};

class DottedShader : public BaseShader {

Vector A, B, C;

public:

DottedShader(Vector A, Vector B, Vector C) : A(A), B(B), C(C) {}

COLOR main(Vector data) {

double x\_factor = fmod(data.x, 0.2) \* 5;

double z\_factor = fmod(data.z, 0.2) \* 5;

if (pow(x\_factor - 0.5, 2) + pow(z\_factor - 0.5, 2) < 0.25 \* 0.25) {

return { 255, 255, 255 };

}

else {

return { 255, 0, 0, 255};

}

}

Vector getVertexData(double h0, double h1, double h2) {

Vector result = {

A.x \* h0 + B.x \* h1 + C.x \* h2,

A.y \* h0 + B.y \* h1 + C.y \* h2,

A.z \* h0 + B.z \* h1 + C.z \* h2

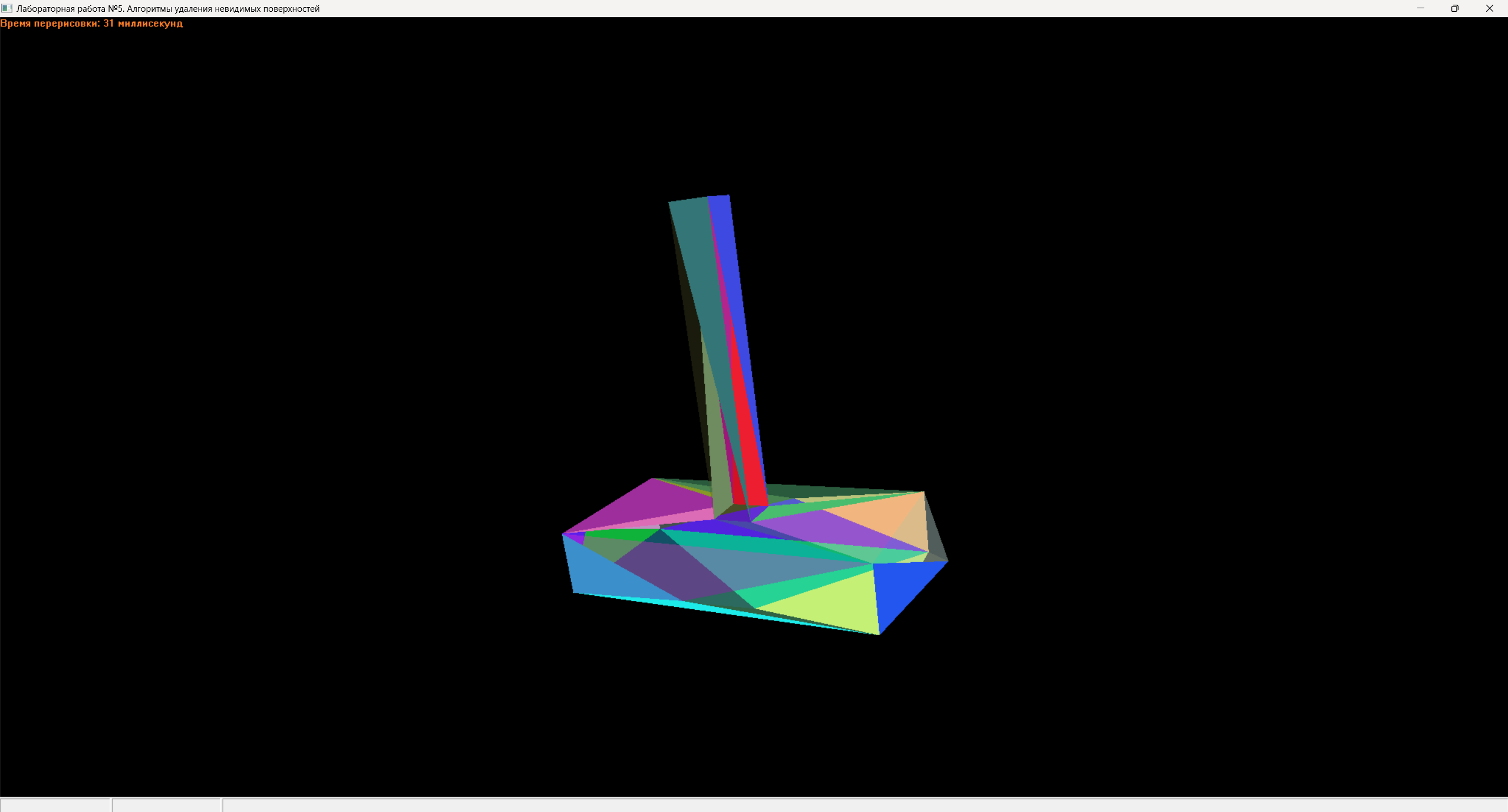
};

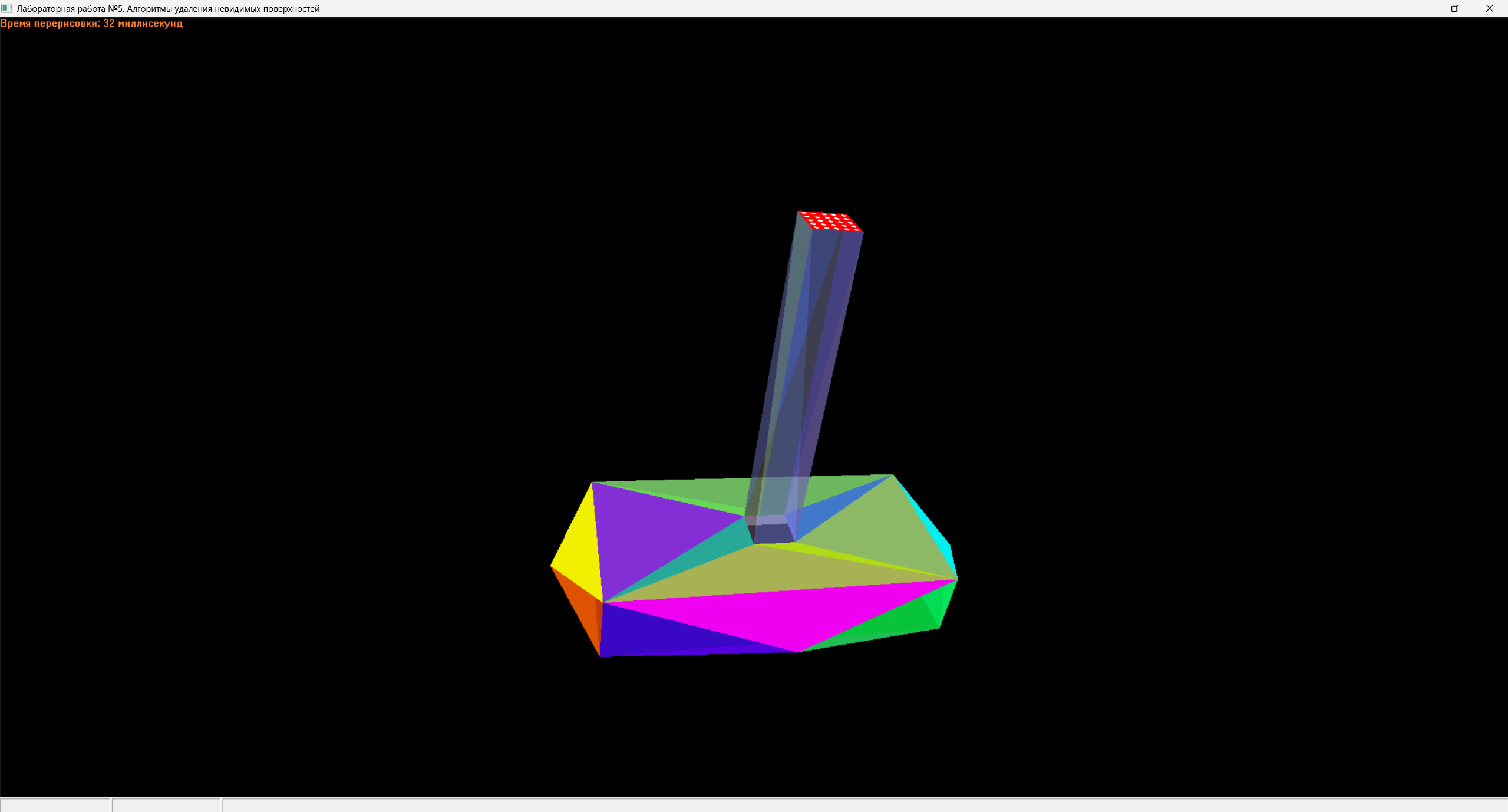
return result;

}

};

Ссылка на репозиторий: <https://github.com/IAmProgrammist/comp_graphics/tree/lab_5_painter_algorithm>





**Вывод:** в ходе лабораторной работы изучили алгоритмы удаления невидимых поверхностей и создать программу для визуализации объёмной трёхмерной модели с закрашенными гранями.