МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.ШУХОВА»

(БГТУ им. В.Г. Шухова)

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Дисциплина: Компьютерная графика

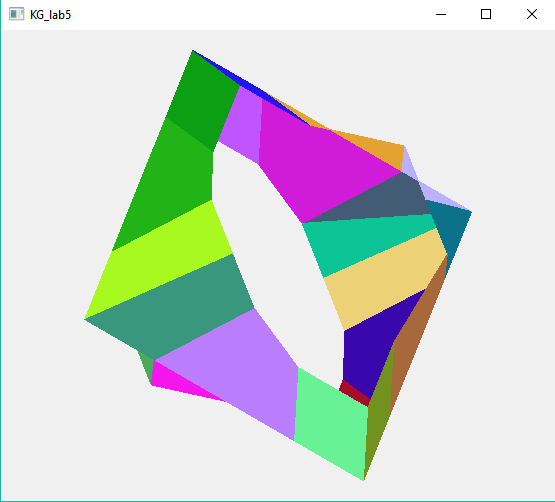
Лабораторная работа № 5

Алгоритмы сортировки

|  |  |
| --- | --- |
|  | Выполнила: ст. группы ПВ-31  Зановская А.И.  Проверил: Осипов О.В. |

Белгород

2018



mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QPainter>

#include <QMouseEvent>

#include "matrix.h"

#include "polygon.h"

#include <ctime>

class MainWindow : public QMainWindow

{

Q\_OBJECT

private:

int cntPol = 0; // Количество полигонов.

Polygon figure[28];

void *paintEvent* (QPaintEvent\*);

void *mousePressEvent*(QMouseEvent\* event);

void *mouseMoveEvent*(QMouseEvent\* event);

double angleX=0;

double angleY=0;

QPointF m\_mousePosition;

QColor randomColor();

public:

MainWindow(QWidget \*parent = 0);

~*MainWindow*();

};

#endif // MAINWINDOW\_H

mainwindow.cpp

#include "mainwindow.h"

QColor MainWindow::randomColor()

{

srand(rand() + time(0));

return QColor(rand() % 256, rand() % 256, rand() % 256, 255);

}

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

{

Vertex V[48];

V[0]=Vertex(0.5, 0, 0.5, 1);

V[1]=Vertex(0.5, -0.25, 0.5, 1);

V[2]=Vertex(1, 0, 0, 1);

V[3]=Vertex(0.75, -0.25, 0, 1);

V[4]=Vertex(0.25, -0.75, 0, 1);

V[5]=Vertex(0.25, -0.5, 0.25, 1);

V[6]=Vertex(0, -0.75, 0.25, 1);

V[7]=Vertex(0, -1, 0, 1);

V[8]=Vertex(-0.25, -0.75, 0, 1);

V[9]=Vertex(-0.25, -0.5, 0.25, 1);

V[10]=Vertex(-0.5, -0.25, 0.5, 1);

V[11]=Vertex(-0.75, -0.25, 0, 1);

V[12]=Vertex(-1, 0, 0, 1);

V[13]=Vertex(-0.5, 0, 0.5, 1);

V[14]=Vertex(-0.5, 0.25, 0.5, 1);

V[15]=Vertex(-0.75, 0.25, 0, 1);

V[16]=Vertex(-0.25, 0.75, 0, 1);

V[17]=Vertex(-0.25, 0.5, 0.25, 1);

V[18]=Vertex(0, 0.75, 0.25, 1);

V[19]=Vertex(0, 1, 0, 1);

V[20]=Vertex(0.25, 0.75, 0, 1);

V[21]=Vertex(0.25, 0.5, 0.25, 1);

V[22]=Vertex(0.5, 0.25, 0.5, 1);

V[23]=Vertex(0.75, 0.25, 0, 1);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

V[24]=Vertex(0.5, 0, -0.5, 1);

V[25]=Vertex(0.5, -0.25, -0.5, 1);

V[26]=Vertex(1, 0, 0, 1);

V[27]=Vertex(0.75, -0.25, 0, 1);

V[28]=Vertex(0.25, -0.75, 0, 1);

V[29]=Vertex(0.25, -0.5, -0.25, 1);

V[30]=Vertex(0, -0.75, -0.25, 1);

V[31]=Vertex(0, -1, 0, 1);

V[32]=Vertex(-0.25, -0.75, 0, 1);

V[33]=Vertex(-0.25, -0.5, -0.25, 1);

V[34]=Vertex(-0.5, -0.25, -0.5, 1);

V[35]=Vertex(-0.75, -0.25, 0, 1);

V[36]=Vertex(-1, 0, 0, 1);

V[37]=Vertex(-0.5, 0, -0.5, 1);

V[38]=Vertex(-0.5, 0.25, -0.5, 1);

V[39]=Vertex(-0.75, 0.25, 0, 1);

V[40]=Vertex(-0.25, 0.75, 0, 1);

V[41]=Vertex(-0.25, 0.5, -0.25, 1);

V[42]=Vertex(0, 0.75, -0.25, 1);

V[43]=Vertex(0, 1, 0, 1);

V[44]=Vertex(0.25, 0.75, 0, 1);

V[45]=Vertex(0.25, 0.5, -0.25, 1);

V[46]=Vertex(0.5, 0.25, -0.5, 1);

V[47]=Vertex(0.75, 0.25, 0, 1);

QColor color = randomColor();

figure[cntPol++] = Polygon({V[0], V[1], V[2], V[0]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[1], V[2], V[3], V[1]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[1], V[3], V[4], V[5], V[1]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[4], V[5], V[6], V[7], V[4]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[6], V[7], V[8], V[9], V[6]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[8], V[9], V[10], V[11], V[8]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[10], V[11], V[12], V[13], V[10]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[12], V[13], V[14], V[15], V[12]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[14], V[15], V[16], V[17], V[14]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[16], V[17], V[18], V[19], V[16]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[20], V[21], V[18], V[19], V[20]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[20], V[21], V[22], V[23], V[20]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[2], V[0], V[22], V[23], V[2]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[24], V[25], V[27], V[26], V[24]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[25], V[27], V[28], V[29], V[25]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[28], V[29], V[30], V[31], V[28]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[30], V[31], V[32], V[33], V[30]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[32], V[33], V[34], V[35], V[321]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[34], V[35], V[36], V[37], V[34]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[36], V[37], V[38], V[39], V[36]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[38], V[39], V[40], V[41], V[38]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[40], V[41], V[42], V[43], V[40]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[42], V[43], V[44], V[45], V[42]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[44], V[45], V[46], V[47], V[44]}, color);

color = randomColor();

figure[cntPol++] = Polygon({V[46], V[47], V[26], V[24], V[46]}, color);

}

void MainWindow::*paintEvent*(QPaintEvent\*)

{

QPainter painter(this);

Matrix3D A = Matrix3D::Rotatey(angleX) \* Matrix3D::Rotatex(angleY);

for (int i = 0; i < cntPol; i++)

for (int j = 0; j < figure[i].lenght(); j++)

figure[i][j] = A \* figure[i][j];

Polygon::sort(figure, cntPol);

int scale;

if (this->height() < this->width())

scale = this->height() / 2;

else

scale = this->width() / 2;

A = Matrix3D::TranslationMatrix(this->width()/ 2.0, this->height()/2.0,0)

\* Matrix3D::ScalingMatrix(scale,scale,scale)\* Matrix3D::OrtographicPMatrix();

for (int i = 0; i < cntPol; i++)

{

QPolygonF pol;

for (int j = 0; j < figure[i].lenght()-1; j++)

pol << ( (A \* figure[i][j]).toQPointF() );

QColor color = figure[i].color;

painter.setBrush(QBrush(color));

painter.setPen(color);

painter.drawPolygon(pol);

}

angleX = 0;

angleY = 0;

}

void MainWindow::*mousePressEvent*(QMouseEvent\* event) {

m\_mousePosition.setX(event->x());

m\_mousePosition.setY(event->y());

}

void MainWindow::*mouseMoveEvent*(QMouseEvent\* event) {

double dx = event->x() - m\_mousePosition.x();

double dy = event->y() - m\_mousePosition.y();

m\_mousePosition.setX(event->x());

m\_mousePosition.setY(event->y());

angleX -= dx /3;

angleY += dy / 3;

repaint();

}

matrix.h

#ifndef Matrix4\_H

#define Matrix4\_H

#include "vector.h"

class Matrix3D

{

private:

double m[4][4];

public:

Matrix3D();

Matrix3D(const Matrix3D& matr);

Matrix3D(double a11, double a12, double a13, double a14,

double a21, double a22, double a23, double a24,

double a31, double a32, double a33, double a34,

double a41, double a42, double a43, double a44);

Matrix3D& operator=(const Matrix3D& matr);

Matrix3D operator \*(const Matrix3D& A);

// Матрицы афинных преобразований.

static Matrix3D ScalingMatrix(double kx, double ky, double kz);

static Matrix3D ReflectionMatrix(Qt::Axis axis);

static Matrix3D TranslationMatrix(double dx, double dy, double dz);

void ZeroMatr();

static Matrix3D Rotatex(double angle);

static Matrix3D Rotatey(double angle);

static Matrix3D Rotatez(double angle);

static Matrix3D OrtographicPMatrix();

static Matrix3D CabinetPMatrix();

static Matrix3D FreePMatrix();

static Matrix3D CentralPMatrix(double dotCoord, Qt::Axis axis=Qt::ZAxis);

};

#endif // Matrix\_H

matrix.cpp

#include "Matrix.h"

#include <cmath>

Matrix3D::Matrix3D()

: m{{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 1 }} {}

Matrix3D::Matrix3D(double a11, double a12, double a13, double a14,

double a21, double a22, double a23, double a24,

double a31, double a32, double a33, double a34,

double a41, double a42, double a43, double a44) {

m[0][0] = a11; m[0][1] = a12; m[0][2] = a13; m[0][3] = a14;

m[1][0] = a21; m[1][1] = a22; m[1][2] = a23; m[1][3] = a24;

m[2][0] = a31; m[2][1] = a32; m[2][2] = a33; m[2][3] = a34;

m[3][0] = a41; m[3][1] = a42; m[3][2] = a43; m[3][3] = a44;

}

Matrix3D::Matrix3D(const Matrix3D& A)

{

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

m[i][j] = A.m[i][j];

}

Matrix3D& Matrix3D::operator=(const Matrix3D& A)

{

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

m[i][j] = A.m[i][j];

}

Matrix3D Matrix3D::operator \*(const Matrix3D& A) {

return Matrix3D (m[0][0] \* A.m[0][0] + m[0][1] \* A.m[1][0] + m[0][2] \* A.m[2][0] + m[0][3] \* A.m[3][0],

m[0][0] \* A.m[0][1] + m[0][1] \* A.m[1][1] + m[0][2] \* A.m[2][1] + m[0][3] \* A.m[3][1],

m[0][0] \* A.m[0][2] + m[0][1] \* A.m[1][2] + m[0][2] \* A.m[2][2] + m[0][3] \* A.m[3][2],

m[0][0] \* A.m[0][3] + m[0][1] \* A.m[1][3] + m[0][2] \* A.m[2][3] + m[0][3] \* A.m[3][3],

m[1][0] \* A.m[0][0] + m[1][1] \* A.m[1][0] + m[1][2] \* A.m[2][0] + m[1][3] \* A.m[3][0],

m[1][0] \* A.m[0][1] + m[1][1] \* A.m[1][1] + m[1][2] \* A.m[2][1] + m[1][3] \* A.m[3][1],

m[1][0] \* A.m[0][2] + m[1][1] \* A.m[1][2] + m[1][2] \* A.m[2][2] + m[1][3] \* A.m[3][2],

m[1][0] \* A.m[0][3] + m[1][1] \* A.m[1][3] + m[1][2] \* A.m[2][3] + m[1][3] \* A.m[3][3],

m[2][0] \* A.m[0][0] + m[2][1] \* A.m[1][0] + m[2][2] \* A.m[2][0] + m[2][3] \* A.m[3][0],

m[2][0] \* A.m[0][1] + m[2][1] \* A.m[1][1] + m[2][2] \* A.m[2][1] + m[2][3] \* A.m[3][1],

m[2][0] \* A.m[0][2] + m[2][1] \* A.m[1][2] + m[2][2] \* A.m[2][2] + m[2][3] \* A.m[3][2],

m[2][0] \* A.m[0][3] + m[2][1] \* A.m[1][3] + m[2][2] \* A.m[2][3] + m[2][3] \* A.m[3][3],

m[3][0] \* A.m[0][0] + m[3][1] \* A.m[1][0] + m[3][2] \* A.m[2][0] + m[3][3] \* A.m[3][0],

m[3][0] \* A.m[0][1] + m[3][1] \* A.m[1][1] + m[3][2] \* A.m[2][1] + m[3][3] \* A.m[3][1],

m[3][0] \* A.m[0][2] + m[3][1] \* A.m[1][2] + m[3][2] \* A.m[2][2] + m[3][3] \* A.m[3][2],

m[3][0] \* A.m[0][3] + m[3][1] \* A.m[1][3] + m[3][2] \* A.m[2][3] + m[3][3] \* A.m[3][3]);

}

Vector3D operator\*(const Matrix3D& A, const Vector3D& V)

{

return Vector3D(A.m[0][0] \* V.vect[0] + A.m[0][1] \* V.vect[1] + A.m[0][2] \* V.vect[2] + A.m[0][3] \* V.vect[3],

A.m[1][0] \* V.vect[0] + A.m[1][1] \* V.vect[1] + A.m[1][2] \* V.vect[2] + A.m[1][3] \* V.vect[3],

A.m[2][0] \* V.vect[0] + A.m[2][1] \* V.vect[1] + A.m[2][2] \* V.vect[2] + A.m[2][3] \* V.vect[3],

A.m[3][0] \* V.vect[0] + A.m[3][1] \* V.vect[1] + A.m[3][2] \* V.vect[2] + A.m[3][3] \* V.vect[3]);

}

Matrix3D Matrix3D::ScalingMatrix(double kx, double ky, double kz){

Matrix3D result;

result.m[0][0] = kx;

result.m[1][1] = ky;

result.m[2][2] = kz;

return result;

}

Matrix3D Matrix3D::ReflectionMatrix(Qt::Axis axis)

{

Matrix3D result;

result.m[0][0] = 1;

result.m[1][1] = 1;

result.m[2][2] = 1;

if (axis == Qt::XAxis)

result.m[0][0] \*= -1;

else if (axis == Qt::YAxis)

result.m[1][1] \*= -1;

else if (axis == Qt::ZAxis)

result.m[2][2] \*= -1;

return result;

}

Matrix3D Matrix3D::TranslationMatrix(double dx, double dy, double dz)

{

Matrix3D result;

result.m[0][0] = 1;

result.m[1][1] = 1;

result.m[2][2] = 1;

result.m[0][3] = dx;

result.m[1][3] = dy;

result.m[2][3] = dz;

return result;

}

void Matrix3D::ZeroMatr()

{

for (int i=0;i<4;i++)

for (int j=0;j<4;j++)

m[i][j]=0;

}

Matrix3D Matrix3D::Rotatex(double angle){

Matrix3D R;

R.ZeroMatr();

R.m[0][0]=1;

R.m[1][1]=cos(angle\*M\_PI/180);

R.m[2][1]=-sin(angle\*M\_PI/180);

R.m[1][2]=sin(angle\*M\_PI/180);

R.m[2][2]=cos(angle\*M\_PI/180);

R.m[3][3]=1;

return R;

}

Matrix3D Matrix3D::Rotatey(double angle){

Matrix3D R;

R.ZeroMatr();

R.m[0][0]=cos(angle\*M\_PI/180);

R.m[2][0]=sin(angle\*M\_PI/180);

R.m[1][1]=1;

R.m[0][2]=-sin(angle\*M\_PI/180);

R.m[2][2]=cos(angle\*M\_PI/180);

R.m[3][3]=1;

return R;

}

Matrix3D Matrix3D::Rotatez(double angle){

Matrix3D R;

R.ZeroMatr();

R.m[0][0]=cos(angle\*M\_PI/180);

R.m[1][0]=-sin(angle\*M\_PI/180);

R.m[0][1]=sin(angle\*M\_PI/180);

R.m[1][1]=cos(angle\*M\_PI/180);

R.m[2][2]=1;

R.m[3][3]=1;

return R;

}

Matrix3D Matrix3D::OrtographicPMatrix()

{

Matrix3D result;

result.m[0][0] = 1;

result.m[1][1] = 1;

return result;

}

Matrix3D Matrix3D::CabinetPMatrix()

{

Matrix3D result;

result.m[0][0] = 1;

result.m[1][1] = 1;

result.m[0][2] = 1;

result.m[1][2] = cos(M\_PI / 4) / 2;

return result;

}

Matrix3D Matrix3D::FreePMatrix()

{

Matrix3D result;

result.m[0][0] = 1;

result.m[1][1] = 1;

result.m[0][2] = 1;

result.m[1][2] = cos(M\_PI / 4);

return result;

}

Matrix3D Matrix3D::CentralPMatrix(double dotCoord, Qt::Axis axis)

{

Matrix3D result;

double coord = -1 / dotCoord;

if (axis == Qt::XAxis)

{

result.m[1][1] = 1;

result.m[2][2] = 1;

result.m[3][0] = coord;

}

else if (axis == Qt::YAxis)

{

result.m[0][0] = 1;

result.m[2][2] = 1;

result.m[3][1] = coord;

}

else if (axis == Qt::ZAxis)

{

result.m[0][0] = 1;

result.m[1][1] = 1;

result.m[3][2] = coord;

}

return result;

}

polygon.h

#ifndef POLYGON\_H

#define POLYGON\_H

#include "vector.h"

#include <QPolygonF>

#include <QColor>

#include <qmath.h>

using Vertex = Vector3D;

class Polygon

{

private:

Vertex ver[5];

int len;

public:

Polygon(int n = 0);

Polygon(std::initializer\_list<Vertex>); //список инициализации

Polygon(std::initializer\_list<Vertex>, QColor);

Polygon(QVector <Vertex> vect);

QColor color;

int lenght() const;

const Vertex& operator[](int i) const;

Vertex& operator[](int i);

QPolygonF toQPolygonF();

bool inside(const Vertex& point) const;

//нахождение точки пересечения

// отрезков ab и cd, а также глубин этих отрезков.

static bool intersects(const Vertex& a, const Vertex& b, const Vertex& c, const Vertex& d, Vertex& o, double& z1, double& z2);

static int cover(const Polygon& A, const Polygon& B, int na, int nb);

static int cover2(const Polygon& A, const Polygon& B, int na, int nb);

static void sort(Polygon\* mas, int n);

};

#endif // POLYGON\_H

polygon.cpp

#include "polygon.h"

Polygon::Polygon(int n) : len(n)

{

for (int i = 0; i < len; i++)

this->ver[i] = Vertex();

}

Polygon::Polygon(std::initializer\_list<Vertex> list)

{

this->len = list.size();

int i = 0;

for (auto it = list.begin(); it != list.end(); it++)

this->ver[i++] = \*it;

}

Polygon::Polygon(std::initializer\_list<Vertex> list, QColor color)

{

this->len = list.size();

int i = 0;

for (auto it = list.begin(); it != list.end(); it++)

this->ver[i++] = \*it;

this->color = color;

}

Polygon::Polygon(QVector <Vertex> list){

this->len = list.size();

int i = 0;

for (i = 0; i < len; i++)

this->ver[i] = list[i];

}

int Polygon::lenght() const

{

return this->len;

}

const Vertex& Polygon::operator[](int i) const

{

return this->ver[i];

}

Vertex& Polygon::operator[](int i)

{

return this->ver[i];

}

QPolygonF Polygon::toQPolygonF()

{

QPolygonF result(this->len);

for (int i = 0; i < this->len; i++)

result[i] = this->ver[i].toQPointF();

return result;

}

inline static int sign(double x)

{

return (x > 0) - (x < 0);

}

inline static double crossProduct(const Vertex& a1, const Vertex& a2,

const Vertex& b1, const Vertex& b2)

{

Vertex a(a2.x() - a1.x(), a2.y() - a1.y()); // Вектор a.

Vertex b(b2.x() - b1.x(), b2.y() - b1.y()); // Вектор b.

return a.x() \* b.y() - a.y() \* b.x();

}

bool Polygon::inside(const Vertex& point) const

{

int currSign=sign(crossProduct(point, this->ver[1],point,this->ver[0]));

int prevSign;

for (int i = 1; i < this->lenght()-1; i++)

{

prevSign = currSign;

currSign=sign(crossProduct(point,this->ver[i+1],point,this->ver[i]));

if (currSign != prevSign)

return false;

}

return true;

}

bool Polygon::intersects(const Vertex &a, const Vertex &b, const Vertex &c, const Vertex &d, Vertex &o, double &za, double &zb)

{

double noPar = (d.y()-c.y())\*(b.x()-a.x())-(d.x()-c.x())\*(b.y()-a.y());

if (fabs(noPar) < 0.001) return false; // Параллельны.

double z1 = ((d.x() - c.x())\*(a.y() - c.y()) - (d.y() - c.y())\*(a.x() - c.x())) / ((d.y() - c.y())\*(b.x() - a.x()) - (d.x() - c.x())\*(b.y() - a.y()));

double z2 = ((b.x() - a.x())\*(a.y() - c.y()) - (b.y() - a.y())\*(a.x() - c.x())) / ((d.y() - c.y())\*(b.x()-a.x()) - (d.x() - c.x())\*(b.y() - a.y()));

if ((z1 < 0 || z1 > 1) && (z2 < 0 || z2 > 1))

return false; // Пересекаются прямые, но не отрезки.

// Найдём точку пересечения, подставив параметры в уравнение.

o = Vertex(a.x() + z1\*(b.x() - a.x()), a.y() + z2\*(b.y() - a.y()));

// Нахождение глубины точки пересечения.

if (fabs(b.x() - a.x()) > fabs(b.y() - a.y()))

zb = (b.z() - a.z()) \* (o.x() - a.x()) / (b.x() - a.x()) + a.z();

else

zb = (b.z() - a.z()) \* (o.y() - a.y()) / (b.y() - a.y()) + a.z();

if (fabs(d.x() - c.x()) > fabs(d.y() - c.y()))

za = (d.z() - c.z()) \* (o.x() - c.x()) / (d.x() - c.x()) + c.z();

else

za = (d.z() - c.z()) \* (o.y() - c.y()) / (d.y() - c.y()) + c.z();

return true;

}

int Polygon::cover(const Polygon &A, const Polygon &B, int na, int nb)

{

for (int i = 0; i < na; i++)

{

for (int j = 0; j < nb; j++)

{

if (!(A[i] == B[j+1] || A[i] == B[j] || A[i+1] == B[j] || A[i+1] == B[j+1]))

{

Vertex o;

double z1;//расстояние от точки пересечения до второго трезка

double z2;//расстояние от точки пересечения до первого трезка

// Пересекаются ли проекции отрезков

bool r = Polygon::intersects(A[i], A[i+1], B[j], B[j+1], o, z1, z2);

if (r)

return z2 < z1 ? 1 : -1;

}

}

}

int pa = Polygon::cover2(A, B, na, nb);

int pb = Polygon::cover2(B, A, nb, na);

return pa - pb;

}

int Polygon::cover2(const Polygon &A, const Polygon &B, int na, int nb)

{

for (int i = 0; i < nb; i++)

{

int j = 0;

while (j < na && B[i] != A[j])

j++;

// Находится ли проекция точки B[i] на плоскость z = 0 внутри проекции многоугольника A.

bool s = A.inside(B[i]);

if (j == na && s)

{

Vertex r;

double za;

double zr;

Polygon::intersects(A[0], A[1], A[2], B[i], r, zr, za);

return zr < B[i].z() ? 1 : -1;

}

}

return 0;

}

void Polygon::sort(Polygon \*mas, int n)

{

for (int i = 1; i < n; i++)

{

Polygon tmp = mas[i];

int j = i - 1;

while (j >= 0 && Polygon::cover(mas[j], tmp, mas[j].lenght()-1, tmp.lenght()-1) >= 0)

{

mas[j + 1] = mas[j];

j--;

}

mas[j + 1] = tmp;

}

}