Владимир Огородников

9 декабря 2015 г.

Оглавление

Глава 1

circle.cpp

```
#include "circle.h"
#include "line.h"
#include "vector.h"
#include <cmath>
#include "math_ext.h"

using namespace geom2d;

Circle::Circle(const Vector &center, const Vector &A, const bool isValid_) :
    x(center.x),
    y(center.y),
    r(center.dist(A)),
    isValid(isValid_ && center.isValid && A.isValid)
{

}

Circle::Circle(const Vector &center, const double r, const bool isValid_) :
    x(center.x),
    y(center.y),
    r(r),
    isValid(isValid_ && center.isValid)
{

}

std::pair<Line, Line> Circle::tangents(const Vector &A, Vector *_X1, Vector *_X2) const
{
    if (!(isValid && A.isValid))
    {
        return std::pair<Line, Line>(Line(false), Line(false));
    }

    Vector O(x, y);

    if (equal(r, 0))
    {

        if (_X1 != NULL) {
            *_X1 = O;
        }

        if (_X2 != NULL) {
            *_X2 = O;
        }

        return std::pair<Line, Line>(Line(A, O), Line(false));
    }

    double oxProjection = r * r / O.dist(A);
    Vector OH = (A - O).normalize() * oxProjection;
    Vector H = O + OH;
    Vector HX = OH.getNormal() * sqrt(r * r - oxProjection * oxProjection);
    Vector X1 = H + HX, X2 = H - HX;

    if (_X1 != NULL) {
        *_X1 = X1;
    }

    if (_X2 != NULL) {
        *_X2 = X2;
    }

    switch (where(A))
    {
    case 1:
        return std::pair<Line, Line>(Line(A, X1), Line(A, X2));

    case 0:
        return std::pair<Line, Line>(Line(A, A + (O - A).getNormal()), Line(false));

    default:
        return std::pair<Line, Line>(Line(false), Line(false));
    }
}

std::pair<Vector, Vector> Circle::intersect(const Line &l) const
```

```
{
    if (!(isValid && l.isValid))
    {
        return std::pair<Vector, Vector>(Vector(false), Vector(false));
    }

    Vector O(x, y);
    Vector H = l.projection(O);

    Vector OH = H - O;
    double hlLength = sqrt(max(r * r - OH.length() * OH.length(), 0.0));
    Vector HL = OH.getNormal() * hlLength;

    switch (where(H))
    {
    case -1:
        return std::pair<Vector, Vector>(H + HL, H - HL);

    case 0:
        return std::pair<Vector, Vector>(H, Vector(false));

    default:
        return std::pair<Vector, Vector>(Vector(false), Vector(false));
    }
}

std::pair<Vector, Vector> Circle::intersect(const Circle &c) const
{
    return intersect(radixLine(c));
}

bool Circle::operator ==(const Circle &c) const
{
    return isValid && c.isValid && equal(x, c.x) && equal(y, c.y) && equal(r, c.r);
}

std::pair<Line, Line> Circle::commonTangents(const Circle &c,
        const bool outer) const
{
    return outer ? commonOuterTangents(c) : commonInnerTangents(c);
}

std::pair<Line, Line> Circle::commonOuterTangents(const Circle &c) const
{
    if (!(isValid && c.isValid) || (Vector(x, y).dist(Vector(c.x, c.y)) < fabs(r - c.r)))
    {
        return std::pair<Line, Line>(Line(false), Line(false));
    }

    Vector point;
    Circle pseudo;

    if (r < c.r)
    {
        point = Vector(x, y);
        pseudo = Circle(c.x, c.y, c.r - r);
    }
    else
    {
        point = Vector(c.x, c.y);
        pseudo = Circle(x, y, r - c.r);
    }

    Vector X1, X2;
    std::pair<Line, Line> ts = pseudo.tangents(point, &X1, &X2);
    if (!ts.second.isValid)
    {
        ts.first.moveToVector((point - pseudo.center()).getNormal() * r);
        ts.first.moveToVector(-(point - pseudo.center()).getNormal() * r);
    }
    else
    {
        ts.first .moveToVector((X1 - pseudo.center()).normalized() * min(r, c.r));
        ts.second.moveToVector((X2 - pseudo.center()).normalized() * min(r, c.r));
    }
    return ts;
}

std::pair<Line, Line> Circle::commonInnerTangents(const Circle &c) const
{
    if (!(isValid && c.isValid) || (Vector(x, y).dist(Vector(c.x, c.y)) < r + c.r))
    {
        return std::pair<Line, Line>(Line(false), Line(false));
    }

    Vector point(c.x, c.y);
    Circle pseudo(x, y, r + c.r);

    Vector X1, X2;
    std::pair<Line, Line> ts = pseudo.tangents(point, &X1, &X2);

    if (!ts.second.isValid)
    {
        ts.first .moveToVector((pseudo.center() - point).normalized() * c.r);
        ts.second = ts.first;
    }
    else
    {
        ts.first .moveToVector((pseudo.center() - X1).normalized() * c.r);
        ts.second.moveToVector((pseudo.center() - X2).normalized() * c.r);
    }
    return ts;
}

Vector Circle::center() const
{
    return Vector(x, y, isValid);
}

bool Circle::isTangent(const Line &l) const
{
    if (!(isValid && l.isValid))
    {
        return false;
    }
    Vector O(x, y);
    return equal(r, (O - l.projection(O)).length());
}

int Circle::where(const Vector &v) const
{
    if (!(isValid && v.isValid))
    {
        return -100;
    }
    Vector O(x, y);
    return signum((O - v).length() - r);
}

Line Circle::radixLine(const Circle &c) const
{
    // (x - x1) ^ 2 + (y - y1) ^ 2 - (x - x2) ^ 2 - (y - y2) ^ 2 = r1 ^ 2 - r2 ^ 2
    // x1^2 - 2*x*x1 + y1^2 - 2*y*y1 -x2^2 + 2*x*x2 - y2^2 + 2*y*y2 = r1^2 - r2^2
    // x * (-2 * x1 + 2 * x2) + y * (-2 * y1 + 2 * y2) + x1^2 + y1^2 - r1^2 - x2^2 -y2^2 +
    return Line(2 * (c.x - x), 2 * (c.y - y), x * x + y * y - r * r - c.x * c.x - c.y * c.y
}
```

Глава 2

circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <utility>
#include <cstring>

namespace geom2d
{
class Vector;
class Line;

class Circle
{
public:
    double x, y, r;
    bool isValid;
    Circle(const double x, const double y, const double r, const bool isValid = true) :
        x(x),
        y(y),
        r(r),
        isValid(isValid)
    {

    }

    Circle(const bool isValid = true) :
        x(0),
        y(0),
        r(0),
        isValid(isValid)
    {
    }

    Circle(const Vector &center, const Vector &A, const bool isValid = true);
    Circle(const Vector &center, const double r, const bool isValid = true);
    std::pair<Line, Line> tangents(const Vector &v, Vector *X1 = NULL, Vector *X2 = NULL) co
    std::pair<Vector, Vector> intersect(const Line &l) const;
    std::pair<Vector, Vector> intersect(const Circle &c) const;
    bool operator ==(const Circle &c) const;
    std::pair<Line, Line> commonTangents(const Circle &c, const bool outer) const;
    std::pair<Line, Line> commonOuterTangents(const Circle &c) const;
    std::pair<Line, Line> commonInnerTangents(const Circle &c) const;
    bool isTangent(const Line &l) const;
    int where(const Vector &v) const;
    Line radixLine(const Circle &c) const;
    Vector center() const;
};
}
#endif // CIRCLE_H
```

Глава 3

line.cpp

```
#include "line.h"
#include "vector.h"
#include <cmath>
#include "math_ext.h"

using namespace geom2d;

Line::Line(const Vector &A, const Vector &B, const bool isValid_)
{
        // Vector normal = (A - B).getNormal();
        Vector direction = (A - B);

        a = direction.y;
        b = - direction.x;
        c = -(a * A.x + b * A.y);
        isValid = isValid_ && A.isValid && B.isValid;
}

double Line::dist(const Vector &A) const
{
        if (!(isValid && A.isValid))
        {
                return -100;
        }
        Line l = normalized();
        return fabs(l.a * A.x + l.b * A.y + l.c);
}

Vector Line::intersect(const Line &l) const
{
        if (!(isValid && l.isValid))
```

```
	{
		return Vector(false);
	}

	double det = a * l.b - l.a * b;
	return Vector(c * l.b - l.c * b, a * l.c - l.a * c) / det;
}

Line Line::normalize()
{
	double k = sqrt(a * a + b * b) * signum(a);
	if (!equal(k, 0))
	{
		a /= k;
		b /= k;
		c /= k;
	}
	return *this;
}

Line Line::normalized() const
{
	Line l(*this);
	return l.normalize();
}

Vector Line::getNormal() const
{
	return Vector(a, b, isValid).normalize();
}

bool Line::operator ||(const Line &l) const
{
	return isValid && l.isValid && (getNormal() || l.getNormal());
}

int Line::where(const Vector &v) const
{
	if (!(isValid && v.isValid))
	{
		return -100;
	}
	return signum(a * v.x + b * v.y + c);
}

bool Line::operator ==(const Line &l) const
```

```
{
        return isValid && l.isValid && equal(a * l.b, b * l.a) && equal(a * l.c, c * l.a);
}

Vector Line::projection(const Vector &v) const
{
        if (!(isValid && v.isValid))
        {
                return Vector(false);
        }
        Vector normal = getNormal();
        return intersect(Line(v, v + normal));
}

void Line::moveToVector(const Vector &v)
{
        if (!(isValid && v.isValid))
        {
                return;
        }
        normalize();
        Vector normal(a, b);
        c -= v * normal;
}
```

Глава 4

line.h

```
#ifndef LINE_H
#define LINE_H

namespace geom2d {

class Vector;

class Line
{
public:
    double a, b, c;
    bool isValid;
    Line(const double a, const double b, const double c, const bool isValid = true) :
        a(a),
        b(b),
        c(c),
        isValid(isValid)
    {

    }

    Line(const bool isValid = true) :
        a(0),
        b(1),
        c(0),
        isValid(isValid)
    {

    }

    Line(const Vector &A, const Vector &B, const bool isValid = true);
    double dist(const Vector &A) const;
    Vector intersect(const Line &l) const;
    Line normalize();
    Line normalized() const;
    Vector getNormal() const;
    bool operator ||(const Line &l) const;
    int where(const Vector &v) const;
    bool operator ==(const Line &l) const;
    Vector projection(const Vector &v) const;
    void moveToVector(const Vector &v);
};
}
#endif // LINE_H
```

Глава 5

main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "vector.h"
#include "line.h"
#include "polygon.h"

using namespace std;
using namespace geom2d;

Polygon readPolygon(istream &in)
{
    int n;
    in >> n;
    vector<Vector> v;

    for (int i = 0; i < n; i++)
    {
        Vector A;
        in >> A;
        v.push_back(A);
    }

    return Polygon(v);
}

void writePolygon(ostream &out, const Polygon &p)
{
    if(!p.isValid)
    {
        out << "This polygon doesn’t exist.\n";
        return;
    }

    vector<Vector> v = p.points;

    for(int i = 0; i < (int) v.size(); i++)
    {
        out <<  v[i] << '\n';
    }
}

bool color = false;
bool convexHull = false;
bool help = false;

int genConvexHull();

int main()
{
    Vector A(0, 0), B(1, 1);
    Line l(A, B);
    l.moveToVector(Vector(2, 1));
    cout << l.a << ' ' << l.b << ' ' << l.c << '\n';
    return 0;
}

int hrenmain(int argc, char *argv[])
{
    for(int i = 0; i < argc; i++)
    {
        if(string(argv[i]) == "-c")
        {
            color = true;
        }

        if(string(argv[i]) == "-h")
        {
            convexHull = true;
        }

        if(string(argv[i]) == "-?")
        {
            help = true;
        }
    }
    if(help)
    {
        cout << "Usage: " << string(argv[0]) << " [-?] [-c] [-h]\n"
                "Use the key -? to see this help.\n"
                "Use the key -c to enable color IO (not available on some terminals).\n"
                "Use the key -h to run another example.\n";
        return 0;
    }

    if(convexHull)
    {
        genConvexHull();
        return 0;
    }

    fixed(cout);
    if(color)    cout << "\x1b[33m";  // Yellow foreground
    cout << "This is a demo program. \n"
            "It searches an intersection of two convex polygons. ";
    if(color)    cout << "\x1b[37m";  // White foreground
    cout << "O(mn)";
    if(color)    cout << "\x1b[33m";  // Yellow foreground
    cout << "\n";

    cout << "Write the number of first polygon's vertices and their coordinates:\n";
    if(color)    cout << "\x1b[32m"; // Green foreground
    Polygon p1 = readPolygon(cin);
    if(color)    cout << "\x1b[33m"; // Yellow foreground

    cout << "Write the number of second polygon's vertices and their coordinates:\n";
    if(color)    cout << "\x1b[32m"; // Green foreground
    Polygon p2 = readPolygon(cin);
    if(color)    cout << "\x1b[33m"; // Yellow foreground

    cout << "Their intersection:\n";
    if(color)    cout << "\x1b[32m"; // Green foreground
    writePolygon(cout, p1.intersection(p2));
    if(color)    cout << "\x1b[0m"; // Restore default
    return 0;
}

int genConvexHull()
{
    fixed(cout);

    if(color)    cout << "\x1b[33m";  // Yellow foreground
    cout << "This is a demo program. \n"
            "It searches a convex hull for a set of points. ";
    if(color)    cout << "\x1b[37m";  // White foreground
    cout << "O(n log n)";
    if(color)    cout << "\x1b[33m";  // Yellow foreground
    cout << "\n";

    int n;
    cout << "Write the number of points and their coordinates:\n";
    if(color)    cout << "\x1b[32m"; // Green foreground
    cin >> n;
    vector<Vector> v;

    for (int i = 0; i < n; i++)
    {
        Vector A;
        cin >> A;
        v.push_back(A);
    }

    Polygon p(v);
    p = p.convexHull();
    v = p.points;

    if(color)    cout << "\x1b[33m"; // Yellow foreground
    cout << "Convex hull's vertices:\n";

    if(color)    cout << "\x1b[32m"; // Green foreground
    for(int i = 0; i < (int) v.size(); i++)
    {
        cout << v[i] << '\n';
    }

    if(color)    cout << "\x1b[0m"; // Restore default
    return 0;
}
```

Глава 6

math_ext.cpp

```
#include <cmath>
#include "math_ext.h"

const double EPS = 1e-7;

double geom2d::eps()
{
    return EPS;
}

double geom2d::toRadians(const double deg)
{
    return deg * M_PI / 180.0;
}

double geom2d::toDegrees(const double rad)
{
    return rad * 180.0 / M_PI;
}

bool geom2d::equal(const double a, const double b)
{
    return std::abs(a - b) < EPS;
}

int geom2d::signum(double a)
{
    return equal(a, 0.0) ? 0 : a > 0 ? 1 : -1;
}

double geom2d::max(double a, double b)
```

```
{
    return a < b ? b : a;
}

double geom2d::min(double a, double b)
{
    return a < b ? a : b;
}
```

Глава 7

math_ext.h

```
#ifndef MATH_EXT_H
#define MATH_EXT_H

namespace geom2d
{

double eps();
double toRadians(const double deg);
double toDegrees(const double rad);
bool equal(const double a, const double b);
int signum(double a);
double max(double a, double b);
double min(double a, double b);

}

#endif // MATH_EXT_H
```

Глава 8

polygon.cpp

```
#include "polygon.h"
#include <cmath>
#include "math_ext.h"
#include <algorithm>
#include "line.h"
#include "vector.h"

using namespace std;
using namespace geom2d;

Polygon::Polygon(const std::vector<Vector> &points, const bool isValid_) :
    points(points)
{
    isValid = isValid_;
    for (int i = 0; i < (int) points.size(); i++)
    {
        isValid &= points[i].isValid;
    }
}

double Polygon::area() const
{
    if (!isValid)
    {
        return -100;
    }

    double ans = 0;
    int n = points.size();

    for (int i = 0; i < n - 1; i++)
    {
        ans += points[i] / (points[i + 1]);
    }

    ans += points[n - 1] / (points[0]);
    return fabs(ans / 2.0);
}

bool Polygon::isConvex() const
{
    if (!isValid)
    {
        return false;
    }
    int n = points.size();
    int sign = signum((points[1] - points[0]) / (points[n - 1] - points[0]));

    for (int i = 1; i < n - 1; i++)
    {
        if (signum((points[i + 1] - points[i]) / (points[i - 1] - points[i])) !=
                sign)
        {
            return false;
        }
    }

    return true;
}

bool Polygon::hasPoint(const Vector &v) const
{
    if (!(isValid && v.isValid))
    {
        return false;
    }

    double ans = 0;
    int n = points.size();

    for (int i = 0; i < n; i++)
        if (points[i] == v)
        {
            return true;
        }

    for (int i = 0; i < n - 1; i++)
    {
        ans += (points[i + 1] - v) ^ (points[i] - v);
    }

    ans += (points[0] - v) ^ (points[n - 1] - v);
    return !equal(ans, 0);
}

class HullComparator
{
public:
    Vector base;
    HullComparator(const Vector &base) :
        base(base)
    {

    }
    bool operator ()(const Vector &a, const Vector &b) const
    {
        Vector v1 = a - base, v2 = b - base;
        double c = v1 / v2;
        return c > 0 || (equal(c, 0) && v1.length() < v2.length());
    }
};

Polygon Polygon::convexHull() const
{
    if(!isValid)
    {
        return Polygon(false);
    }

    if (this->points.size() < 3)
    {
        return Polygon(this->points);
    }

    std::vector<Vector> points = this->points;
    int idx = 0;
    int n = points.size();

    for (int i = 1; i < n; i++)
        if (equal(points[i].y, points[idx].y) ? points[i].x < points[idx].x :
                points[i].y < points[idx].y)
        {
            idx = i;
        }

    std::swap(points[idx], points[0]);

    std::sort(points.begin() + 1, points.end(), HullComparator(points[0]));
    std::vector<Vector> stack;
    stack.push_back(points[0]);
    stack.push_back(points[1]);

    for (int i = 2; i < n; i++)
    {
        Vector A = points[i];
        Vector O = stack.back();
        Vector B = stack[stack.size() - 2];

        while (stack.size() > 1 && (A - O) / (B - O) < eps())
        {
            stack.pop_back();
            O = stack.back();
            B = stack[stack.size() - 2];
        }

        stack.push_back(A);
    }

    return Polygon(stack);
}

std::pair<Polygon, Polygon> Polygon::split(const Line &l) const
// l.where(): First -> -1; Second -> 1
{
    if (!(isValid && l.isValid))
    {
        return std::pair<Polygon, Polygon>(Polygon(false), Polygon(false));
    }

    vector<Vector> first, second;
    int lastWhere = l.where(points[0]);
    for (int i = 0; i < (int) points.size(); i++)
    {
        int where = l.where(points[i]);
        if (where * lastWhere == -1)
        {
            Vector intersection = l.intersect(Line(points[i], points[i-1]));
            first.push_back(intersection);
            second.push_back(intersection);
        }

        if (where != 1)
            first.push_back(points[i]);
        if (where != -1)
            second.push_back(points[i]);

        lastWhere = where;
    }

    int where = l.where(points[0]);
    if (where * lastWhere == -1)
    {
        Vector intersection = l.intersect(Line(points[0], points.back()));
        first.push_back(intersection);
        second.push_back(intersection);
    }

    pair<Polygon, Polygon> ans;
    ans.first.points = first;
    ans.first.isValid = first.size() >= 3;
    ans.second.points = second;
    ans.second.isValid = second.size() >= 3;
    return ans;
}

Polygon Polygon::intersection(const Polygon &p) const
{
    if (!(isValid && p.isValid))
    {
        return Polygon(false);
    }

    Polygon ans = p;
    for (int i = 0, n = points.size(); i < n; i++)
    {
        Vector A = points[i], B = points[(i + 1) % n];
        Line l(A, B);
        std::pair<Polygon, Polygon> pp = ans.split(l);
        if (l.where(points[(i + 2) % n]) == 1)
        {
            ans = pp.second;
        }
        else
        {
            ans = pp.first;
        }
    }

    ans.isValid &= ans.points.size() >= 3;

    return ans;
}
```

Глава 9

polygon.h

```
#ifndef POLYGON_H
#define POLYGON_H

#include <vector>
#include "vector.h"
#include "line.h"

namespace geom2d {

class Polygon
{
public:
    std::vector<Vector> points;
    bool isValid;
    Polygon(const std::vector<Vector> &points, const bool isValid = true);
    Polygon(const bool isValid = true) :
        isValid(isValid)
    {
        points.resize(1);
    }

    double area() const;
    bool hasPoint(const Vector &v) const;
    bool isConvex() const;
    Polygon convexHull() const;
    std::pair<Polygon, Polygon> split(const Line &l) const;
    Polygon intersection(const Polygon &p) const;
};
}
#endif // POLYGON_H
```

Глава 10

ray.cpp

```
#include "ray.h"

using namespace geom2d;

double Ray::dist(Vector &v) const
{
    if (!(isValid && v.isValid))
    {
        return -100;
    }
    Vector H = projection(v);
    if ((H - end) * direction < 0)
    {
        return v.dist(end);
    }
    return v.dist(H);
}
```

Глава 11

ray.h

```
#ifndef RAY_H
#define RAY_H

#include "vector.h"
#include "line.h"

namespace geom2d {

class Ray : public Line
{
public:
    Vector end, direction;
    Ray(const Vector &end, const Vector &direction, const bool isValid = true) :
        Line(end, end + direction, isValid && end.isValid && direction.isValid),
        end(end),
        direction(direction)
    {

    }

    double dist(Vector &v) const;
};
}
#endif // RAY_H
```

Глава 12

vector.cpp

```
#include <cmath>
#include <iostream>
#include "vector.h"
#include "math_ext.h"

using namespace geom2d;

Vector Vector::operator -() const
{
    return Vector(-x, -y, isValid);
}

Vector Vector::operator +(const Vector &v) const
{
    return Vector(x + v.x, y + v.y, isValid && v.isValid);
}

Vector Vector::operator -(const Vector &v) const
{
    return Vector(x - v.x, y - v.y, isValid && v.isValid);
}

Vector Vector::operator *(const double d) const
{
    return Vector(x * d, y * d, isValid);
}

Vector Vector::operator /(const double d) const
{
    return Vector(x / d, y / d, isValid && !equal(d, 0));
}
```

```
double Vector::operator *(const Vector &v) const
{
    return x * v.x + y * v.y;
}

double Vector::operator /(const Vector &v) const
{
    return x * v.y - y * v.x;
}

double Vector::operator ^(const Vector &v) const
{
    if (!(isValid && v.isValid))
    {
        return -100;
    }
    return atan2((*this) / (v), (*this)*(v));
}

Vector geom2d::operator *(const double d, const Vector &v)
{
    return v * d;
}

double Vector::length() const
{
    if (!isValid)
    {
        return -100;
    }
    return sqrt(x * x + y * y);
}

double Vector::dist(const Vector &v) const
{
    return (*this - v).length();
}

bool Vector::operator ==(const Vector &v) const
{
    return isValid && v.isValid && equal(x, v.x) && equal(y, v.y);
}

Vector Vector::operator +=(const Vector &v)
{
    x += v.x;
    y += v.y;
    isValid &= v.isValid;
    return *this;
}

Vector Vector::operator -=(const Vector &v)
{
    x -= v.x;
    y -= v.y;
    isValid &= v.isValid;
    return *this;
}

Vector Vector::operator *=(double d)
{
    x *= d;
    y *= d;
    return *this;
}

Vector Vector::normalize()
{
    double l = length();
    if (!equal(l, 0))
    {
        x /= l;
        y /= l;
    }
    return *this;
}

Vector Vector::normalized() const
{
    Vector v(*this);
    v.normalize();
    return v;
}

Vector Vector::getNormal() const
{
    return Vector(y, -x, isValid).normalize();
}

bool Vector::operator ||(const Vector &v) const
{
    return isValid && v.isValid && equal(x * v.y, y * v.x);
}

Vector Vector::rotated(const double phi) const
{
    Vector v(*this);
    double cosa = v.x, sina = v.y;
    double cosphi = cos(phi), sinphi = sin(phi);
    v.x = cosa * cosphi - sina * sinphi;
    v.y = sina * cosphi + cosa * sinphi;
    return v;
}

Vector Vector::rotate(const double phi)
{
    return *this = rotated(phi);
}

std::istream &geom2d::operator >>(std::istream &in, Vector &v)
{
    in >> v.x >> v.y;
    return in;
}

std::ostream &geom2d::operator <<(std::ostream &out, const Vector &v)
{
    out << v.x << ' ' << v.y;
    return out;
}
```

Глава 13

vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>

namespace geom2d {

class Vector
{
public:
    double x, y;
    bool isValid;
    Vector(const double x, const double y, const bool isValid = true) :
        x(x),
        y(y),
        isValid(isValid)
    {

    }

    Vector(const Vector &v) :
        x(v.x),
        y(v.y),
        isValid(v.isValid)
    {

    }

    Vector(const bool isValid = true) :
        x(0),
        y(0),
        isValid(isValid)
    {

    }
    Vector operator -() const;
    Vector operator +(const Vector &v) const;
    Vector operator -(const Vector &v) const;
    double operator *(const Vector &v) const;
    Vector operator *(const double d) const;
    Vector operator /(const double d) const;
    double operator /(const Vector &v) const;
    double operator ^(const Vector &v) const;
    double length() const;
    double dist(const Vector &v) const;
    bool operator ==(const Vector &v) const;
    Vector operator +=(const Vector &v);
    Vector operator -=(const Vector &v);
    Vector operator *=(double d);
    Vector normalize();
    Vector normalized() const;
    Vector getNormal() const;
    bool operator ||(const Vector &v) const;
    Vector rotate(const double phi);
    Vector rotated(const double phi) const;
};

Vector operator *(const double d, const Vector &v);
std::istream &operator >>(std::istream &in, Vector &v);
std::ostream &operator <<(std::ostream &out, const Vector &v);

}


#endif // VECTOR_H
```