Geometry Concepts: Basic Concepts

By lbackstrom
TopCoder Member

Introduction

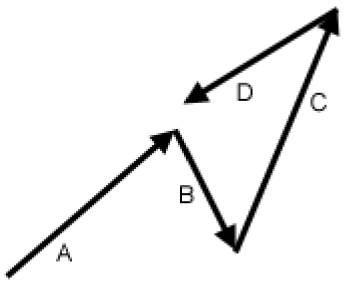
Many TopCoders seem to be mortally afraid of geometry problems. I think it's safe to say that the majority of them would be in favor of a ban on TopCoder geometry problems. However, geometry is a very important part of most graphics programs, especially computer games, and geometry problems are here to stay. In this article, I'll try to take a bit of the edge off of them, and introduce some concepts that should make geometry problems a little less frightening.

Vectors

Vectors are the basis of a lot of methods for solving geometry problems. Formally, a vector is defined by a direction and a magnitude. In the case of two-dimension geometry, a vector can be represented as pair of numbers, x and y, which gives both a direction and a magnitude. For example, the line segment from (1,3) to (5,1) can be represented by the vector (4,-2). It's important to understand, however, that the vector defines only the direction and magnitude of the segment in this case, and does not define the starting or ending locations of the vector.

Vector Addition

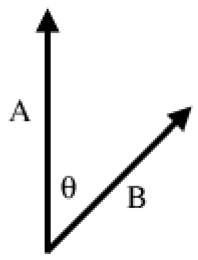
There are a number of mathematical operations that can be performed on vectors. The simplest of these is addition: you can add two vectors together and the result is a new vector. If you have two vectors (x_1, y_1) and (x_2, y_2), then the sum of the two vectors is simply (x_1+x_2, y_1+y_2). The image below shows the sum of four vectors. Note that it doesn't matter which order you add them up in - just like regular addition. Throughout these articles, we will use plus and minus signs to denote vector addition and subtraction, where each is simply the piecewise addition or subtraction of the components of the vector.



The sum of vectors A+B+C+D

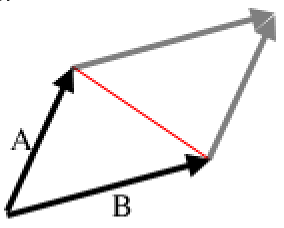
Dot Product

The addition of vectors is relatively intuitive; a couple of less obvious vector operations are dot and cross products. The dot product of two vectors is simply the sum of the products of the corresponding elements. For example, the dot product of (x_1, y_1) and (x_2, y_2) is $x_1*x_2 + y_1*y_2$. Note that this is not a vector, but is simply a single number (called a scalar). The reason this is useful is that the dot product, $A \cdot B = |A||B|Cos(\theta)$, where θ is the angle between the A and B. $|A|$ is called the norm of the vector, and in a 2-D geometry problem is simply the length of the vector, $sqrt(x^2+y^2)$. Therefore, we can calculate $Cos(\theta) = (A \cdot B)/(|A||B|)$. By using the `acos` function, we can then find θ. It is useful to recall that $Cos(90) = 0$ and $Cos(0) = 1$, as this tells you that a dot product of 0 indicates two perpendicular lines, and that the dot product is greatest when the lines are parallel. A final note about dot products is that they are not limited to 2-D geometry. We can take dot products of vectors with any number of elements, and the above equality still holds.



Cross Product

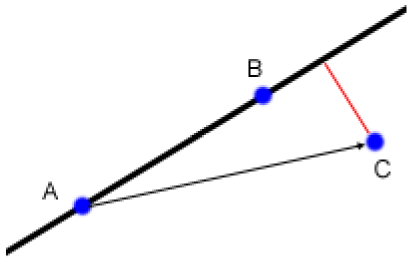
An even more useful operation is the cross product. The cross product of two 2-D vectors is `x1*y2 - y1*x2` Technically, the cross product is actually a vector, and has the magnitude given above, and is directed in the +z direction. Since we're only working with 2-D geometry for now, we'll ignore this fact, and use it like a scalar. Similar to the dot product, `A x B = |A||B|Sin(θ)`. However, θ has a slightly different meaning in this case: `|θ|` is the angle between the two vectors, but θ is negative or positive based on the right-hand rule. In 2-D geometry this means that if A is less than 180 degrees clockwise from B, the value is positive. Another useful fact related to the cross product is that the absolute value of `|A||B|Sin(θ)` is equal to the area of the parallelogram with two of its sides formed by A and B. Furthermore, the triangle formed by A, B and the red line in the diagram has half of the area of the parallelogram, so we can calculate its area from the cross product also.



Parallelogram from A and B

Line-Point Distance

Finding the distance from a point to a line is something that comes up often in geometry problems. Lets say that you are given 3 points, A, B, and C, and you want to find the distance from the point C to the line defined by A and B (recall that a line extends infinitely in either direction). The first step is to find the two vectors from A to B (AB) and from A to C (AC). Now, take the cross product `AB x AC`, and divide by `|AB|`. This gives you the distance (denoted by the red line) as `(AB x AC)/|AB|`. The reason this works comes from some basic high school level geometry. The area of a triangle is found as `base*height/2`. Now, the area of the triangle formed by A, B and C is given by `(AB x AC)/2`. The base of the triangle is formed by AB, and the height of the triangle is the distance from the line to C. Therefore, what we have done is to find twice the area of the triangle using the cross product, and then divided by the length of the base. As always with cross products, the value may be negative, in which case the distance is the absolute value.



Things get a little bit trickier if we want to find the distance from a line segment to a point. In this case, the nearest point might be one of the endpoints of the segment, rather than the closest point on the line. In the diagram above, for example, the closest point to C on the line defined by A and B is not on the segment AB, so the point closest to C is B. While there are a few different ways to check for this special case, one way is to apply the dot product. First, check to see if the nearest point on the line AB is beyond B (as in the example above) by taking AB · BC. If this value is greater than 0, it means that the angle between AB and BC is between -90 and 90, exclusive, and therefore the nearest point on the segment AB will be B. Similarly, if BA · AC is greater than 0, the nearest point is A. If both dot products are negative, then the nearest point to C is somewhere along the segment. (There is another way to do this, which I'll discuss here).

```
    //Compute the dot product AB · BC

    int dot(int[] A, int[] B, int[] C){

        AB = new int[2];

        BC = new int[2];
        AB[0] = B[0]-A[0];

        AB[1] = B[1]-A[1];

        BC[0] = C[0]-B[0];

        BC[1] = C[1]-B[1];

        int dot = AB[0] * BC[0] + AB[1] * BC[1];

        return dot;

    }

    //Compute the cross product AB x AC

    int cross(int[] A, int[] B, int[] C){

        AB = new int[2];

        AC = new int[2];

        AB[0] = B[0]-A[0];

        AB[1] = B[1]-A[1];

        AC[0] = C[0]-A[0];

        AC[1] = C[1]-A[1];

        int cross = AB[0] * AC[1] - AB[1] * AC[0];

        return cross;

    }

    //Compute the distance from A to B

    double distance(int[] A, int[] B){

        int d1 = A[0] - B[0];

        int d2 = A[1] - B[1];

        return sqrt(d1*d1+d2*d2);

    }

    //Compute the distance from AB to C

    //if isSegment is true, AB is a segment, not a line.

    double linePointDist(int[] A, int[] B, int[] C, boolean isSegment){

        double dist = cross(A,B,C) / distance(A,B);

        if(isSegment){

            int dot1 = dot(A,B,C);

            if(dot1 > 0)return distance(B,C);
            int dot2 = dot(B,A,C);

            if(dot2 > 0)return distance(A,C);

        }

        return abs(dist);

    }
```

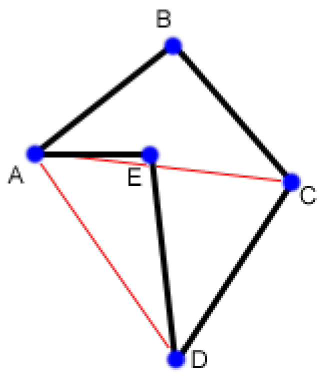
That probably seems like a lot of code, but lets see the same thing with a point class and some operator overloading in C++ or C#. The * operator is the dot product, while ^ is cross product, while + and - do what you would expect.

```
    //Compute the distance from AB to C

    //if isSegment is true, AB is a segment, not a line.

    double linePointDist(point A, point B, point C, bool isSegment){

        double dist = ((B-A)^(C-A)) / sqrt((B-A)*(B-A));

        if(isSegment){

            int dot1 = (C-B)*(B-A);

            if(dot1 > 0)return sqrt((B-C)*(B-C));

            int dot2 = (C-A)*(A-B);

            if(dot2 > 0)return sqrt((A-C)*(A-C));

        }

        return abs(dist);

    }
```

Operator overloading is beyond the scope of this article, but I suggest that you look up how to do it if you are a C# or C++ coder, and write your own 2-D point class with some handy operator overloading. It will make a lot of geometry problems a lot simpler.

Polygon Area

Another common task is to find the area of a polygon, given the points around its perimeter. Consider the non-convex polygon below, with 5 points. To find its area we are going to start by triangulating it. That is, we are going to divide it up into a number of triangles. In this polygon, the triangles are ABC, ACD, and ADE. But wait, you protest, not all of those triangles are part of the polygon! We are going to take advantage of the signed area given by the cross product, which will make everything work out nicely. First, we'll take the cross product of `AB x AC` to find the area of ABC. This will give us a negative value, because of the way in which A, B and C are oriented. However, we're still going to add this to our sum, as a negative number. Similarly, we will take the cross product `AC x AD` to find the area of triangle ACD, and we will again get a negative number. Finally, we will take the cross product `AD x AE` and since these three points are oriented in the opposite direction, we will get a positive number. Adding these three numbers (two negatives and a positive) we will end up with a negative number, so will take the absolute value, and that will be area of the polygon.



The reason this works is that the positive and negative number cancel each other out by exactly the right amount. The area of ABC and ACD ended up contributing positively to the final area, while the area of ADE contributed negatively. Looking at the original polygon, it is obvious that the area of the polygon is the area of ABCD (which is the same as ABC + ABD) minus the area of ADE. One final note, if the total area we end up with is negative, it means that the points in the polygon were given to us in clockwise order. Now, just to make this a little more concrete, lets write a little bit of code to find the area of a polygon, given the coordinates as a 2-D array, p.

```
int area = 0;

int N = lengthof(p);

//We will triangulate the polygon

//into triangles with points p[0],p[i],p[i+1]



for(int i = 1; i+1<N; i++){

    int x1 = p[i][0] - p[0][0];

    int y1 = p[i][1] - p[0][1];

    int x2 = p[i+1][0] - p[0][0];

    int y2 = p[i+1][1] - p[0][1];

    int cross = x1*y2 - x2*y1;

    area += cross;

}

return abs(cross/2.0);
```

Notice that if the coordinates are all integers, then the final area of the polygon is one half of an integer.

Geometry Concepts: Line Intersection and its Applications

In the previous section we saw how to use vectors to solve geometry problems. Now we are going to learn how to use some basic linear algebra to do line intersection, and then apply line intersection to a couple of other problems.

Line-Line Intersection

One of the most common tasks you will find in geometry problems is line intersection. Despite the fact that it is so common, a lot of

coders still have trouble with it. The first question is, what form are we given our lines in, and what form would we like them in? Ideally, each of our lines will be in the form $Ax+By=C$, where A, B and C are the numbers which define the line. However, we are rarely given lines in this format, but we can easily generate such an equation from two points. Say we are given two different points, (x_1, y_1) and (x_2, y_2), and want to find A, B and C for the equation above. We can do so by setting
$A = y_2-y_1$
$B = x_1-x_2$
$C = A*x_1+B*y_1$
Regardless of how the lines are specified, you should be able to generate two different points along the line, and then generate A, B and C. Now, lets say that you have lines, given by the equations:
$A_1x + B_1y = C_1$
$A_2x + B_2y = C_2$
To find the point at which the two lines intersect, we simply need to solve the two equations for the two unknowns, x and y.

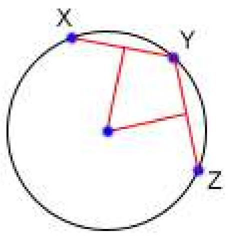
```
    double det = A1*B2 - A2*B1

    if(det == 0){

        //Lines are parallel

    }else{

        double x = (B2*C1 - B1*C2)/det

        double y = (A1*C2 - A2*C1)/det

    }
```

To see where this comes from, consider multiplying the top equation by B_2, and the bottom equation by B_1. This gives you
$A_1B_2x + B_1B_2y = B_2C_1$
$A_2B_1x + B_1B_2y = B_1C_2$
Now, subtract the bottom equation from the top equation to get
$A_1B_2x - A_2B_1x = B_2C_1 - B_1C_2$
Finally, divide both sides by $A_1B_2 - A_2B_1$, and you get the equation for x. The equation for y can be derived similarly.

This gives you the location of the intersection of two lines, but what if you have line segments, not lines. In this case, you need to make sure that the point you found is on both of the line segments. If your line segment goes from (x_1,y_1) to (x_2,y_2), then to check if (x,y) is on that segment, you just need to check that $\min(x_1,x_2) \le x \le \max(x_1,x_2)$, and do the same thing for y. You must be careful about double precision issues though. If your point is right on the edge of the segment, or if the segment is horizontal or vertical, a simple comparison might be problematic. In these cases, you can either do your comparisons with some tolerance, or else use a fraction class.

Finding a Circle From 3 Points

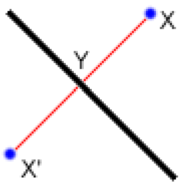
Given 3 points which are not colinear (all on the same line) those three points uniquely define a circle. But, how do you find the center and radius of that circle? This task turns out to be a simple application of line intersection. We want to find the perpendicular bisectors of XY and YZ, and then find the intersection of those two bisectors. This gives us the center of the circle.



To find the perpendicular bisector of XY, find the line from X to Y, in the form $Ax+By=C$. A line perpendicular to this line will be given by the equation $-Bx+Ay=D$, for some D. To find D for the particular line we are interested in, find the midpoint between X and Y by taking the midpoint of the x and y components independently. Then, substitute those values into the equation to find D. If we do the same thing for Y and Z, we end up with two equations for two lines, and we can find their intersections as described above.

Reflection

Reflecting a point across a line requires the same techniques as finding a circle from 3 points. First, notice that the distance from X to the line of reflection is the same as the distance from X' to the line of reflection. Also note that the line between X and X' is perpendicular to the line of reflection. Now, if the line of reflection is given as $Ax+By=C$, then we already know how to find a line perpendicular to it: `-Bx+Ay=D`. To find D, we simply plug in the coordinates for X. Now, we can find the intersection of the two lines at Y, and then find `X' = Y - (X - Y)`.



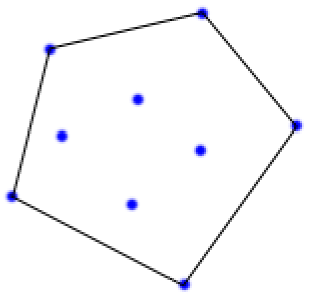
Rotation

Rotation doesn't really fit in with line intersection, but I felt that it would be good to group it with reflection. In fact, another way to find the reflected point is to rotate the original point 180 degrees about Y.

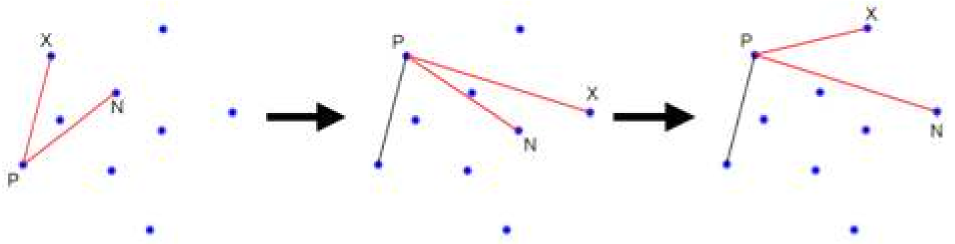
Imagine that we want to rotate one point around another, counterclockwise by θ degrees. For simplicity, lets assume that we are rotating about the origin. In this case, we can find that `x' = x Cos(θ) - y Sin(θ)` and `y' = x Sin(θ) + y Cos(θ)`. If we are rotating about a point other than the origin, we can account for this by shifting our coordinate system so that the origin is at the point of rotation, doing the rotation with the above formulas, and then shifting the coordinate system back to where it started.

Convex Hull

A convex hull of a set of points is the smallest convex polygon that contains every one of the points. It is defined by a subset of all the points in the original set. One way to think about a convex hull is to imagine that each of the points is a peg sticking up out of a board. Take a rubber band and stretch it around all of the points. The polygon formed by the rubber band is a convex hull. There are many different algorithms that can be used to find the convex hull of a set of points. In this article, I'm just going to describe one of them, which is fast enough for most purposes, but is quite slow compared to some of the other algorithms.



First, loop through all of your points and find the leftmost point. If there is a tie, pick the highest point. You know for certain that this point will be on the convex hull, so we'll start with it. From here, we are going to move clockwise around the edge of the hull, picking the points on the hull, one at a time. Eventually, we will get back to the start point. In order to find the next point around the hull, we will make use of cross products. First, we will pick an unused point, and set the next point, N, to that point. Next, we will iterate through each unused points, X, and if `(X-P) x (N-P)` (where P is the previous point) is negative, we will set N to X. After we have iterated through each point, we will end up with the next point on the convex hull. See the diagram below for an illustration of how the algorithm works. We start with P as the leftmost point. Now, say that we have N and X as shown in the leftmost frame. In this case the cross product will be negative, so we will set N = X, and there will be no other unused points that make the cross product negative, and hence we will advance, setting P = N. Now, in the next frame, we will end up setting N = X again, since the cross product here will be negative. However, we aren't done yet because there is still another point that will make the cross product negative, as shown in the final frame.



The basic idea here is that we are using the cross product to find the point which is furthest counterclockwise from our current position at P. While this may seem fairly straightforward, it becomes a little bit tricky when dealing with colinear points. If you have no colinear points on the hull, then the code is very straightforward.

```
    convexHull(point[] X){

        int N = lengthof(X);
        int p = 0;
        //First find the leftmost point
        for(int i = 1; i<N; i++){
            if(X[i] < X[p])
                p = i;
        }
        int start = p;
        do{
            int n = -1;
            for(int i = 0; i<N; i++){

                //Don't go back to the same point you came from
                if(i == p)continue;

                //If there is no N yet, set it to i
                if(n == -1)n = i;
                int cross = (X[i] - X[p]) x (X[n] - X[p]);

                if(cross < 0){
                    //As described above, set N=X
                    n = i;
                }
            }
            p = n;
        }while(start!=p);
    }
```

Once we start to deal with colinear points, things get trickier. Right away we have to change our method signature to take a boolean specifying whether to include all of the colinear points, or only the necessary ones.

```
    //If onEdge is true, use as many points as possible for
    //the convex hull, otherwise as few as possible.
    convexHull(point[] X, boolean onEdge){
        int N = lengthof(X);

        int p = 0;

        boolean[] used = new boolean[N];

        //First find the leftmost point

        for(int i = 1; i<N; i++){

            if(X[i] < X[p])

                p = i;

        }

        int start = p;

        do{

            int n = -1;

            int dist = onEdge?INF:0;

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

                //X[i] is the X in the discussion



                //Don't go back to the same point you came from

                if(i==p)continue;



                //Don't go to a visited point

                if(used[i])continue;



                //If there is no N yet, set it to X

                if(n == -1)n = i;

                int cross = (X[i] - X[p]) x (X[n] - X[p]);



                //d is the distance from P to X

                int d = (X[i] - X[p]) · (X[i] - X[p]);

                if(cross < 0){

                    //As described above, set N=X

                    n = i;

                    dist = d;
                }else if(cross == 0){

                    //In this case, both N and X are in the

                    //same direction.  If onEdge is true, pick the

                    //closest one, otherwise pick the farthest one.

                    if(onEdge && d < dist){

                        dist = d;

                        n = i;

                    }else if(!onEdge && d > dist){

                        dist = d;

                        n = i;

                    }

                }

            }

            p = n;

            used[p] = true;

        }while(start!=p);

    }
```

Geometry Concepts: Using Geometry in TopCoder Problems

PointInPolygon (SRM 187)

Requires: Line-Line Intersection, Line-Point Distance

First off, we can use our Line-Point Distance code to test for the "BOUNDARY" case. If the distance from any segment to the test point is 0, then return "BOUNDARY". If you didn't have that code pre-written, however, it would probably be easier to just check and see if the test point is between the minimum and maximum x and y values of the segment. Since all of the segments are vertical or horizontal, this is sufficient, and the more general code is not necessary.

Next we have to check if a point is in the interior or the exterior. Imagine picking a point in the interior and then drawing a ray from that point out to infinity in some direction. Each time the ray crossed the boundary of the polygon, it would cross from the interior to the exterior, or vice versa. Therefore, the test point is on the interior if, and only if, the ray crosses the boundary an odd number of times. In practice, we do not have to draw a raw all the way to infinity. Instead, we can just use a very long line segment from the test point to a point that is sufficiently far away. If you pick the far away point poorly, you will end up having to deal with cases where the long segment touches the boundary of the polygon where two edges meet, or runs parallel to an edge of a polygon — both of which are tricky cases to deal with. The quick and dirty way around this is to pick two large random numbers for the endpoint of the segment. While this might not be the most elegant solution to the problem, it works very well in practice. The chance of this segment intersecting anything but the interior of an edge are so small that you are almost guaranteed to get the right answer. If you are really concerned, you could pick a few different random points, and take the most common answer.

```
String testPoint(verts, x, y){

    int N = lengthof(verts);

    int cnt = 0;

    double x2 = random()*1000+1000;

    double y2 = random()*1000+1000;

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

        if(distPointToSegment(verts[i],verts[(i+1)%N],x,y) == 0)

            return "BOUNDARY";

        if(segmentsIntersect((verts[i],verts[(i+1)%N],{x,y},{x2,y2}))

            cnt++;

    }

    if(cnt%2 == 0)return "EXTERIOR";

    else return "INTERIOR";

}
```

TVTower(SRM 183)

Requires: Finding a Circle From 3 Points

In problems like this, the first thing to figure out is what sort of solutions might work. In this case, we want to know what sort of circles we should consider. If a circle only has two points on it, then, in most cases, we can make a slightly smaller circle, that still has those two points on it. The only exception to this is when the two points are exactly opposite each other on the circle. Three points, on the other hand, uniquely define a circle, so if there are three points on the edge of a circle, we cannot make it slightly smaller, and still have all three of them on the circle. Therefore, we want to consider two different types of circles: those with two points exactly opposite each other, and those with three points on the circle. Finding the center of the first type of circle is trivial — it is simply halfway between the two points. For the other case, we can use the method for Finding a Circle From 3 Points. Once we find the center of a potential circle, it is then trivial to find the minimum radius.

```
int[] x, y;

int N;

double best = 1e9;

void check(double cx, double cy){

    double max = 0;

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

        max = max(max,dist(cx,cy,x[i],y[i]));

    }
    best = min(best,max);
}
double minRadius(int[] x, int[] y){
    this.x = x;
    this.y = y;
    N = lengthof(x);
    if(N==1)return 0;
    for(int i = 0; i<N; i++){
        for(int j = i+1; j<N; j++){
            double cx = (x[i]+x[j])/2.0;
            double cy = (y[i]+y[j])/2.0;
            check(cx,cy);
            for(int k = j+1; k<N; k++){
                //center gives the center of the circle with
                //(x[i],y[i]), (x[j],y[j]), and (x[k],y[k]) on
                //the edge of the circle.
                double[] c = center(i,j,k);
                check(c[0],c[1]);
            }
        }
    }
    return best;
}
```

Satellites (SRM 180)

Requires: Line-Point Distance

This problem actually requires an extension of the Line-Point Distance method discussed previously. It is the same basic principle, but the formula for the cross product is a bit different in three dimensions.

The first step here is to convert from spherical coordinates into (x,y,z) triples, where the center of the earth is at the origin.

```
    double x = sin(lng/180*PI)*cos(lat/180*PI)*alt;
    double y = cos(lng/180*PI)*cos(lat/180*PI)*alt;
    double z = sin(lat/180*PI)*alt;
```

Now, we want to take the cross product of two 3-D vectors. As I mentioned earlier, the cross product of two vectors is actually a vector, and this comes into play when working in three dimensions. Given vectors (x_1,y_1,z_1) and (x_2,y_2,z_2) the cross product is defined as the vector `(i,j,k)` where

```
    i = y₁z₂ - y₂z₁;

    j = x₂z₁ - x₁z₂;

    k = x₁y₂ - x₂y₁;
```

Notice that if $z_1 = z_2 = 0$, then i and j are 0, and k is equal to the cross product we used earlier. In three dimensions, the cross product is still related to the area of the parallelogram with two sides from the two vectors. In this case, the area of the parallelogram is the norm of the vector: `sqrt(i*i+j*j+k*k)`.

Hence, as before, we can determine the distance from a point (the center of the earth) to a line (the line from a satellite to a rocket). However, the closest point on the line segment between a satellite and a rocket may be one of the end points of the segment, not the closest point on the line. As before, we can use the dot product to check this. However, there is another way which is somewhat simpler to code. Say that you have two vectors originating at the origin, S and R, going to the satellite and the rocket, and that |X| represents the norm of a vector X.
Then, the closest point to the origin is R if $|R|^2 + |R-S|^2 \le |S|^2$ and it is S if $|S|^2 + |R-S|^2 \le |R|^2$
Naturally, this trick works in two dimensions also.

Further Problems

Once you think you've got a handle on the three problems above, you can give these ones a shot. You should be able to solve all of them with the methods I've outlined, and a little bit of cleverness. I've arranged them in what I believe to ascending order of difficulty.

ConvexPolygon (SRM 166)
Requires: Polygon Area

Surveyor (TCCC '04 Qual 1)
Requires: Polygon Area

Travel (TCI '02)
Requires: Dot Product

Parachuter (TCI '01 Round 3)
Requires: Point In Polygon, Line-Line Intersection

PuckShot (SRM 186)
Requires: Point In Polygon, Line-Line Intersection

ElectronicScarecrows (SRM 173)
Requires: Convex Hull, Dynamic Programming

Mirrors (TCI '02 Finals)
Requires: Reflection, Line-Line Intersection

Symmetry (TCI '02 Round 4)
Requires: Reflection, Line-Line Intersection

Warehouse (SRM 177)
Requires: Line-Point Distance, Line-Line Intersection

The following problems all require geometry, and the topics discussed in this article will be useful. However, they all require some additional skills. If you got stuck on them, the editorials are a good place to look for a bit of help. If you are still stuck, there has yet to be a problem related question on the round tables that went unanswered.

DogWoods (SRM 201)
ShortCut (SRM 215)
SquarePoints (SRM 192)

Tether (TCCC '03 W/MW Regional)
TurfRoller (SRM 203)
Watchtower (SRM 176)