Computational Geometry

Jonathan Wang

December 1, 2006

Ocassionally we run upon problems that require doing some math – specifically geometry. Since we can hardly draw angles and find similar triangles in programs, such problems are solved using computational geometry.

1 Vectors

Computational geometry deals with finding relations between points on the cartesian coordinate plane. However, with the use of doubles, divide by zero slopes, and other ugliness, we will refrain from using equations of lines. If you do find yourself trying to solve problems by solving systems of equations and lines or something similarly complicated, try to think again – the answer will usually be vectors.

A vector, simply put, is an arrow. It points in some direction, and has a length, which we call the magnitude. Being an arrow, it does not have a definite position, it is only relative. We denote vectors by putting an arrow on top ($\vec{r}$), or when we are typing by bolding the letter ($\mathbf{r}$). Vectors have an x, y, and z component, describing where it points. A unit vector is a vector in some direction with magnitude 1. We write vectors in multiple forms: two of the most common are $\langle x, y, z\rangle$ and $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$. $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are unit vectors in the positive x, y, z directions. Although the second form takes longer to write out, it is more useful, so we will use it. We denote the magnitude of a vector $\mathbf{r}$ as r or $|\mathbf{r}|$. Magnitude is equal to $\sqrt{x^2 + y^2 + z^2}$ by Pythagorean theorem. $-\mathbf{r}$ is a vector directly opposite to $\mathbf{r}$.

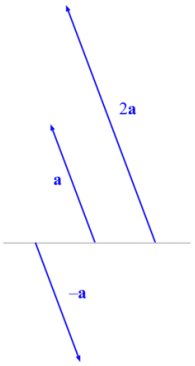
We can multiply a vector by a real number, called a scalar. The resulting vector is:

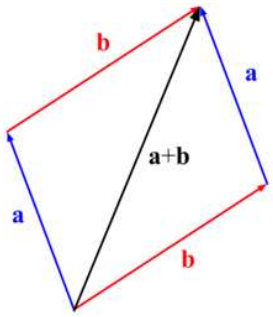
$$k\mathbf{r} = (kr_x)\mathbf{i} + (kr_y)\mathbf{j} + (kr_z)\mathbf{k}$$

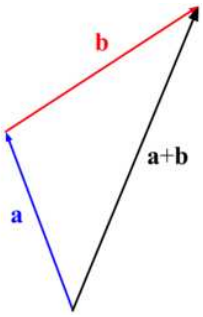
The magnitude of $k\mathbf{r}$ is kr.

The unit vector of a vector is written as $\hat{r}$:

$$\hat{r} = \frac{\mathbf{r}}{r}$$







1.1 Addition and Subtraction

When we add two vectors, we put the tail (the end without the arrow) of one vector to the head of the other. The resulting vector goes from the tail of the second vector to the head of the first. Vector subtraction works in the same way.

$$\mathbf{r} = \mathbf{u} + \mathbf{v}$$

$$r_x\mathbf{i} + r_y\mathbf{j} + r_z\mathbf{k} = (u_x + v_x)\mathbf{i} + (u_y + v_y)\mathbf{j} + (u_z + v_z)\mathbf{k}$$

We can add the components of the vector in each direction separately; the same goes for subtraction.

There are two ways to "multiply" two vectors. Let's start with the easier one:

1.2 Dot product

Strangely enough, the dot product of two vectors is not a vector, but a scalar. The dot product of vectors $\mathbf{u}$ and $\mathbf{v}$ is written as $\mathbf{u} \cdot \mathbf{v}$.

$$\mathbf{r} = \mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z + v_z$$

The magnitude $r = |\mathbf{u} \cdot \mathbf{v}| = uv \cos\theta$, where θ is the angle between $\mathbf{u}$ and $\mathbf{v}$ ($\theta \leq 180°$). From this, we can tell that, if $u, v \neq 0$, $|\mathbf{u} \cdot \mathbf{v}| < 0$ if $\theta > 90°$ and $|\mathbf{u} \cdot \mathbf{v}| = 0$ if $\theta = 90°$ and $|\mathbf{u} \cdot \mathbf{v}| > 0$ if $\theta < 90°$. We use the dot product most often to check if two lines are perpendicular.

The dot product **is** commutative: $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$.

Physically, the dot product is the scalar projection of $\mathbf{u}$ onto $\mathbf{v}$ times v, or vice versa.

1.3 Cross product

The cross product is probably more useful but it's also harder to find and remember. The cross product is written as $\mathbf{u} \times \mathbf{v}$. The best way to remember how to find it is to write the two vectors in a matrix:

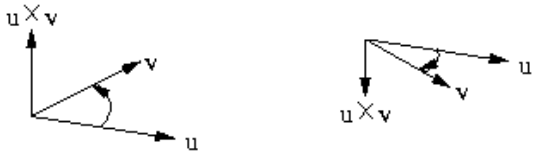
$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

The cross product is then the determinant of this matrix:

$$\mathbf{u} \times \mathbf{v} = (u_y v_z - u_z v_y)\mathbf{i} - (u_x v_z - u_z v_x)\mathbf{j} + (u_x v_y - u_y v_x)\mathbf{k}$$

For two dimensional arrays, set the z component to 0.

The cross product is perpendicular to the two original vectors. There are two directions that the cross product can point in. The direction is determined by whether $\mathbf{u}$ is to the right of $\mathbf{v}$ or to the left. We use the right hand rule to determine this direction (this may be more difficult for left-handed people).



Due to this directional difference, the cross product **is not** commutative: $\mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u})$.

The magnitude of the cross product $|\mathbf{u} \times \mathbf{v}| = uv \sin\theta$, which means that the cross product represents the area of the quadrilateral with adjacent sides $\mathbf{u}$ and $\mathbf{v}$. The cross product has magnitude 0 when $u = 0$, $v = 0$, or $\mathbf{u}$ and $\mathbf{v}$ are parallel.

Phew! We're done with our general introduction to vectors, here are just a few specific coding tips for computational geometry problems:

2 Geometric Algorithms

2.1 Arctangent

OK, this isn't really an algorithm, but when you want to find an angle between a vector and the **positive** x axis. Naturally you want to take the arctangent, but passing a double or float of value $\frac{y}{x}$ may have division by zero or negative x values. There is a special function in C++ that takes the y and x values and gives you an angle between $-\pi$ and π, which is just what you want. The function is `atan2(y,x)`.

2.2 Debugging

This is really annoying for computational geometry problems, so hopefully your code works the first time. Sadly, that rarely happens. Computational geometry problems are also annoying because of the many special cases - make sure you cover all of these, especially cases that might make your program crash (division by zero). Also remember that when checking for double equality, check for $|a - b| < \epsilon$, where $\epsilon = 10^{-9}$.

2.3 Area of a Triangle

If you know the three vertices, which we can represent as position vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, we can pick a vertex (let's say $\mathbf{a}$) and create vectors of the adjacent sides ($\mathbf{u} = \mathbf{b} - \mathbf{a}$ and $\mathbf{v} = \mathbf{c} - \mathbf{a}$). If you remember, the cross product's magnitude was the area of the parallelogram which the vectors as adjacent sides. The area of the triangle is therefore half of the magnitude of the cross product $\frac{1}{2}|\mathbf{u} \times \mathbf{v}|$.

You can also use Heron's formula to find the area of a triangle with **side lengths** of a, b, and c. We find the semiperimeter $s = \frac{a+b+c}{2}$. The area is then $\sqrt{s(s-a)(s-b)(s-c)}$.

2.4 Area of a Polygon

The area of a polygon with consecutive vertices $(x_1, y_1), \ldots, (x_n, y_n)$ is equal to the pseudo-determinant of the matrix:

$$\frac{1}{2}\begin{vmatrix} x_1 & x_2 & \ldots & x_n \\ y_1 & y_2 & \ldots & y_n \end{vmatrix}$$

which is equal to $|x_1y_2 + x_2y_3 + \ldots + x_ny_1 - y_1x_2 - y_2x_3 - \ldots - y_nx_1|$.

2.5 Check if two lines are parallel

As we have mentioned before, just make two vectors representative of the two lines, and take the cross product. If the magnitude is zero, the lines are parallel. Since we deal with floats, check for almost zero-ness instead.

2.6 Distance from a point to a line

We want to find the perpendicular distance from a point P to a line AB. We use position vectors to represent $\mathbf{P}$, $\mathbf{A}$, $\mathbf{B}$. The distance

$$d(P, AB) = \frac{|(\mathbf{P} - \mathbf{A}) \times (\mathbf{B} - \mathbf{A})|}{|\mathbf{B} - \mathbf{A}|}$$

which can be proved with a little trigonometry.

2.7 Check if a point is on a line

A point is on a line if the distance from the point to the line is 0.

2.8 Distance from a point to a line segment

Basically, check if the triangle PAB is obtuse. If it is, the distance will be the minimum of $d(P, A)$ and $d(P, B)$. Otherwise, use the distance from a point to a line formula.

2.9 Check if a point is on a line segment

Check if distance from a point to the line segment is 0. Alternatively check if distance from point to line is zero and $AB = AP + PB$.

2.10 Check if points on the same side of line

This only works for two dimensions. If we want to check if points C and D are on the same side of line AB, calculate the z component of $(\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A})$ and $(\mathbf{B} - \mathbf{A}) \times (\mathbf{D} - \mathbf{A})$. If the z components have the same sign (you can check if their product is positive), then C and D are on the same side of the line AB.

2.11 Check if point inside a triangle

A triangle is bounded by three lines. We know that the average of the vertices of the triangle is inside the triangle. So we check if this average point and the point in question are on the same side of each of the three sides of the triangle. If they are, the point we are concerned with is inside the triangle. If you like, you can pick some other point that is inside the triangle aside from the average point as well.

2.12 Check if point inside convex polygon

You can do the same thing you did with the triangle, but now with n sides.

2.13 Check if four (or more) points are coplanar

To determine if a collection of points are coplanar, we take three points, A, B, and C. If for some other point D, $(\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A}) \cdot (\mathbf{D} - \mathbf{A}) = 0$, then the collection of points resides in the same plane.

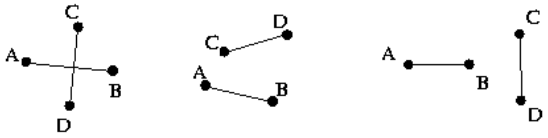
2.14 Line intersection

In 2D, two lines intersect if they are not parallel.

In 3D, two lines intersect if they are not parallel and the four endpoints of the lines are coplanar (or just the lines are coplanar).

2.15 Line segment intersection

In 2D, two line segments AB and CD intersect if and only if A and B are on opposite sides of line CD **and** C and D are on opposite sides of line AB.



In 3D, you must solve a system of equations for the two variables i and j:

$$A_x + (B_x - A_x)i = C_x + (D_x - C_x)j$$

$$A_y + (B_y - A_y)i = C_y + (D_y - C_y)j$$

$$A_z + (B_z - A_z)i = C_z + (D_z - C_z)j$$

This is just parameterizing the two 3D line segments and solving. If the system has solution (i, j) where $0 \leq i, j \leq 1$ (meaning the point is between the endpoints), then the line segments intersect at: $\langle A_x + (B_x - A_x)i, A_y + (B_y - A_y)i, A_z + (B_z - A_z)i \rangle$.

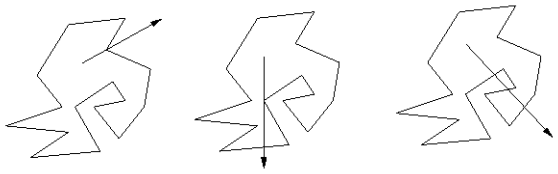
For 2D, you can solve the same system of equations without the z component to find the point of intersection.

2.16 Checking convexity of 2-dimensional polygon

To check the convexity of a 2-dimensional polygon, traverse the vertices in clock-wise order. For every three vertices A, B, C, calculate the cross product $(\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A})$. If the z component of all of the cross products are positive, the polygon is convex.

2.17 Checking point in non-convex polygon

To determine if a point is inside a non-convex polygon, make a ray in a random direction from the point and count how many times it intersects the polygon. If the ray intersects at a vertex or along an edge, pick a new direction. Otherwise, the point is inside the polygon if and only if the ray intersects the polygon an odd number of times. You can do this for convex polygons as well.



3 Geometry Methodologies

There are some techniques you can use to optimize programs or approximate solutions.

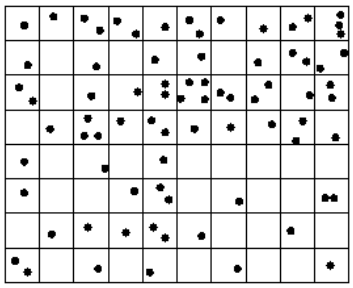
3.1 Monte Carlo

Instead of calculated something rigorously, you can use randomness to your advantage. If you simulate an event over and over and calculate the probability that something occurs, you will get an approximate answer very close to the actual (if you do it enough).

For example, if you want to find the area of a figure, throw darts at a bounding box, and find the probability of hitting the figure inside the box. Multiplying the probability by the area of the box will give a rough estimate of the area of the figure.

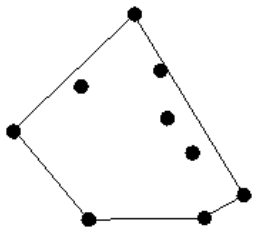
3.2 Partitioning

You can cut a plane into many sections (usually a grid). If you want to calculate something concerning a figure in the plane, you can only look at the sections that contain parts of the figure (i.e., if the section is blank, skip it). This can improve the speed of your algorithm.



4 Convex Hull

Consider the problem where we have a whole bunch of points and we want to find the convex polygon with the smallest area such that each point is contained within or on the boundary of the polygon. In other words, if each point is a fixed peg, determine the convex polygon that is formed by we stretch a rubber band enclosing all the pegs.



We will teach the Graham Scan algorithm for convex hulls, which runs in $N \log N$ time. There is also the Gift Wrap algorithm, which runs in N^2 time. Since it is slower and not actually easier to code, we will not describe that algorithm. Feel free to look it up on your own.

We find the average of all the points as the center point that we are certain is inside the convex hull. Then, we set this point as the "origin" and find the angles all the potential vertices make to the positive x axis. This lends itself to using the `atan2` function. We sort the angles, to get the points in counterclockwise order. Now we traverse the points one by one. For every new point, we check the previous two points and see if the angle from $P_i P_{i-1} P_{i-2}$ is concave, using cross products. If it isn't concave, we delete point P_{i-1} and check until it is concave. After we go through all the points, we have to check points P_n, P_0, and P_1, make close the polygon and still ensure concavity.

The pseudocode for this algorithm is as follows:

```
# x(i), y(i) is the x,y position
     #    of the i-th point
     # zcrossprod(v1,v2) -> z component
     #             of the vectors v1, v2
     #  if zcrossprod(v1,v2) < 0,
     #     then v2 is "right" of v1
     #   since we add counter-clockwise
     #     <0 ->  angle > 180 deg
 1   (midx, midy) = (0, 0)
```

```
 2   For all points i
 3     (midx, midy) = (midx, midy) +
           (x(i)/npoints, y(i)/npoints)
 4   For all points i
 5     angle(i) = atan2(y(i) - midy,
                        x(i) - midx)
 6     perm(i) = i

 7   sort perm based on the angle() values
     # i.e., angle(perm(0)) <=
              angle(perm(i)) for all i

     # start making hull
 8   hull(0) = perm(0)
 9   hull(1) = perm(1)
10   hullpos = 2
11   for all points p, perm() order,
                except perm(npoints - 1)
12     while (hullpos > 1 and
          zcrossprod(hull(hullpos-2) -
13        hull(hullpos-1),
          hull(hullpos-1) - p) < 0)
14           hullpos = hullpos - 1
15       hull(hullpos) = p
16       hullpos = hullpos + 1

     # add last point
17   p = perm(npoints - 1)
18   while (hullpos > 1 and
          zcrossprod(hull(hullpos-2) -
19        hull(hullpos-1),
          hull(hullpos-1) - p) < 0)
20     hullpos = hullpos - 1

21   hullstart = 0
22   do
23     flag = false
24     if (hullpos - hullstart >= 2 and
            zcrossprod(p -
25          hull(hullpos-1),
            hull(hullstart) - p) < 0)
26       p = hull(hullpos-1)
27       hullpos = hullpos - 1
28       flag = true
29     if (hullpos - hullstart >= 2 and
           zcrossprod(hull(hullstart) - p,
           hull(hullstart+1) -
           hull(hullstart)) < 0)
30         hullstart = hullstart + 1
31         flag = true
32   while flag
33   hull(hullpos) = p
34   hullpos = hullpos + 1
```

Information and graphics taken from USACO training pages and Wikipedia.