Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 8 - Computational Geometry

OK, the topic this week is computational geometry. Before we go into the details, I'd like to give just some brief history on the geometry, because it is a fascinating subject. Geometry is by far the oldest form of mathematics - it is said that Plato's academy had an inscription over the entrance

$$\acute{\alpha}\gamma\epsilon\omega\mu\grave{\epsilon}\rho\eta\tau\varsigma\ \mu\eta\delta\epsilon\grave{\iota}\varsigma\ \epsilon\acute{\iota}\sigma\acute{\iota}\tau\omega, \tag{1}$$

or "ageometretos medeis eigito", meaning "Let no one ignorant of geometry enter here." However, prior to the 17th century geometry was almost exclusively dealt with synthetically - in terms of pure shapes and focused on the study of relationships between these forms, constructions of forms, and so on. Synthetic geometry is probably what you learnt in your high school geometry classes, it is in my opinion one of the most beautiful forms of mathematics, and is also not what we're going to be focused on today.

The work of Rene Decartes and Pierre de Fermat led to the development of *analytic* geometry, which was essentially the merging of synthetic geometry with the algebra that was then becoming common. Rather than dealing with Platonic forms of shapes and relationships between those forms, geometric shapes are embedded into a Cartesian grid and those shapes can be represented as algebraic functions. Then analysis of those shapes and their interactions can be done in terms of their algebraic representations rather than directly on some geometric plane. Computational geometry, in turn, is largely about the application of computing power to analytic problems. We'll be given some set of shapes or points with some implicit or explicit embedding in a particular Cartesian coordinate system and asked some question about the interactions between those shapes. I'll get into some specific questions later in the lecture.

So very commonly we'll be working in two dimensions (which is all I'll cover today), so we'll have two orthogonal axes, x and y, with positions of points within the plane represented as distances along each of those axes. So for example, we might have a point positioned at 2 units along the x axis and 3 along the y axis, meaning it can be represented by the pair (2,3). Shapes, in turn, can be represented as relationships between x and y variables - a circle around the point (0,0) of radius r is represented by the equation x^2 + y^2 = r^2.

I'm sure that everybody has seen a lot of this before, but I do like to impress upon you the importance of the progression from the very abstract and proof-oriented synthetic geometry to the more concrete and answer-oriented analytic geometry.

Computational geometry has a well-deserved reputation for being... not necessarily *difficult*, but certainly very *tricky*. This is particularly the case when you're solving problems numerically on a computer! There are a couple of reasons for this:

1) When solving computational geometry problems, there are often a number of important edge or corner cases (Are three points colinear? What happens if this line is vertical? What if this is a concave polygon?) that are tricky to get right.

2) Numerical stability is difficult to get right! Real numbers on a computer are generally dealt with using fixed-precision floating point numbers. But some real numbers can't be accurately represented by floats, giving rounding errors, and some algebraically correct solutions can give arbitrarily bad results due to these problems.

One of the more famous ones is the solutions to a quadratic equation, which you can probably remember as:

$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$
$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$

These are algebraically completely correct. But consider what happens in the computer when using them to evaluate solutions where b^2 is much much larger than $4ac$ and b is positive. In this case, the square root is approximately equal to b itself, and x_1 involves subtracting two nearly equal quantities from each other. This is guaranteed to magnify any rounding errors. Under these conditions, it's much better to use the algebraically equivalent but much more stable form

$x_1 = \frac{2c}{-b-\sqrt{b^2-4ac}}$

In general, some rules to improve numerical stability are: avoid floats and use integers instead whenever possible, perform equality tests on floats with some error (e.g., test whether their difference is less than some epsilon, rather than whether they are explicitly equal), and massage your algebra so as to avoid large numbers and subtracting nearly equal quantities.

Use Existing Implementations! More than any other topic I have or will cover in this class, using existing implementations for geometrical algorithms and primitives will save you a *lot* of trouble. It is distressingly easy to find a correct algorithm and solution and get incorrect answers because of these problems. If you can find them for your language of choice, use existing implementations where any numerical bugs have likely already been worked out.

Geometric Primitives

Points Points are the basic building block of higher dimensional objects. In 2D Euclidean space, we'll represent a point as a pair of x and y coordinates with respect to the origin. These coordinates can be either integers or floats depending on the problem itself.

Sorting is a common requirement - this can be done by overloading a comparison operator to compare first by x and second by y and calling a sort routine with this comparison operator.

Similarly, testing equality of points can be done by testing whether both the x and y coordinates of the two points are equal. *Don't forget for float coordinates to check equality using some epsilon accuracy rather than exactly!*

Distance between points can be calculated using the Pythagorean theorem: $D = \sqrt{dx^2 + dy^2}$. Be careful here, though - square roots are likely to cause numerical problems, and if exact distances aren't required, don't use them - to test distance equality, it might be better to use the square distance values.

Rotation of points around the origin can be done by multiplying them by a rotation matrix, giving:
$x' = x\ cos(\theta) - y\ sin(\theta)$
$y' = x\ sin(\theta) + y\ cos(\theta)$

Lines A line in 2D Euclidean space is the set of coordinates which satisfy some linear equation $ax+by+c = 0$, and so can be represented as a triplet of coefficients (a, b, c). As a warning: do *not* use the standard form you may remember $y = mx + b$ (slope/intercept form). This is undefined for vertical lines! Instead, the value of b in our triplet notation marks whether a line is vertical - it is 0 for vertical lines, and we will choose it to be 1 for non-vertical lines. (Other values of b would require us to scale the other coefficients, making comparisons more difficult.) Any pair of non-equal points defines a line going through those points, and so given a pair of points we can return the correct coefficients representing that line.

If vertical $(x_1 == x_2)$: $L = (a = 1, b = 0, c = -x_1)$

Otherwise: $L = (a = -(y_1 - y_2)/(x_1 - x_2), b = 1, c = -a * x_1 - y_1)$

Two lines then will be parallel if they have the same a and b. If c is also the same, the lines are identical. (Again, be careful about equality testing with floats!) If they are not parallel or identical, then they will intersect in exactly one point, which can be found by solving the system of two linear algebraic equations with two unknowns.

Common Tasks

Direction of Three Points Here's a common problem - take three ordered points A, B, and C. If you are traveling from A to B to C, which direction do you have to turn at B to end up at C? It might be left or right, or it could be straight, in which case the points are colinear. Let's define a function

$ccw(A, B, C) = (B - A) \times (C - A)$,

where $\times$ represents the (magnitude of the) cross product of the two points: $M \times N = x_M y_N - x_N y_M$. The magnitude of the cross product of two vectors is equivalent to the area of the parallelogram spanning those vectors. If the three points are colinear, then the parallellogram is empty and this value will be zero. If the value is positive, the path from A to B to C involves a leftward (or counterclockwise) turn, while if it's negative, it is a rightward turn.

Area of a Polygon That meaning of the cross product of two vectors as the area of the parallelogram also suggests a straightforward way to measure the area of the triangle ABC. As the cross product of (B-A) and (C-A) is the area of the spanning parallelogram, the triangle ABC has exactly half the area of that spanning parallelogram, so we can write the formula for the area of a triangle from three points as:

$$Area(A, B, C) = \frac{(B - A) \times (C - A)}{2}. \tag{2}$$

An alternative way to represent this is as one half the determinant of the matrix of vertex positions:

$$Area(A, B, C) = \frac{1}{2} \begin{vmatrix} x_A & y_A \\ x_B & y_B \\ x_C & y_C \end{vmatrix} = \frac{1}{2}(x_A y_B + x_B y_C + x_C y_A - x_A y_C - x_B y_A - x_C y_B) \tag{3}$$

The useful thing about this representation is that we can extend it to larger polygons with more than three vertices, so that the area of a polygon with N vertices - concave or convex, given in either clockwise or counterclockwise order - is half the determinant of the associated matrix:

$$Area = \frac{1}{2} \begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \cdots \\ x_{n-1} & y_{n-1} \end{vmatrix} = \frac{1}{2}(x_0 y_1 + x_1 y_2 + \cdots + x_{n-1} y_0 - x_0 y_{n-1} - x_1 y_0 - x_2 y_1 \cdots - x_{n-1} y_{n-2}) \tag{4}$$

Be careful about the order of the points for larger polygons! For triangles, every order you can give it is either clockwise or counterclockwise, so the absolute magnitude is guaranteed correct (though sometimes it will be negative). For larger polygons, giving unordered vertices can give you arbitrarily wrong answers. For the vertices of a square, for example, given in the wrong order, you can get an output of zero rather than the correct area.

Convex Hull The convex hull problem is: given n points in the 2D plane, find the smallest convex polygon that contains all of the given points. How can we do this? Well, we want to identify edges between points that are in the convex hull. For some edge AB, if AB is a part of the convex hull, then if we measure the direction of every other point in the set with respect to AB, they should all be the same. Every other point should be either to the left or to the right of AB. If there's discrepancies, then AB can't be a part of the convex hull. That suggests a simple $O(n^3)$ algorithm - take the edge between each pair of points and check whether all the other points are on the same side of that edge. If they are, add that edge to our list of edges in the convex hull. Then once we've gone through them all, connect those edges to form a complete polygon. This algorithm works, but it's quite slow and will be far too inefficient to run on any moderate or larger sized problem. We need something better, and the better algorithm I'll describe is called Graham's scan.

The first thing to note is that there are some points in any set that we can guarantee are a part of the convex hull. In particular, if we take the leftmost point, we can prove that it has to be a part of the convex hull. If it weren't there would have to be some edge between two other points that leaves the leftmost point inside the hull. But any edge between two other points will always leave it to the left of the edge, outside of the hull. So it has to be a part of the hull.

So we'll take the leftmost point as a starting position, and call it point 0. Then, let's sort all of the other points in the array in increasing order of the slope of the edge between themselves and point 0: $(y_i - y_0)/(x_i - x_0)$. And then we work through each of them like this:

First, we add the first two points to our convex hull, giving us three edges. And the direction from point 0 to point 1 to point 2 should be leftwards, or counterclockwise. Then we repeat the following:

Add the next point in our sorted list to our hull and check whether we turn left or right to reach it. If left, we keep that point and move on to the next point. If right, then we've just made a wrong turn and our polygon has a concavity in it. So we remove not the most recent vertex we added, but the one just before - the center of the right turn. And then we check again, seeing if that fixes the concavity, removing vertices until we're turning left again.

This is much faster than the naive algorithm - the sorting time actually dominates the time complexity. The actual scanning part only takes $O(n)$ time, so the $O(nlgn)$ of sorting is what limits the speed.

There is, however, a way to speed this up even more by preemptively identifying points that are necessarily interior! By a similar argument as above, we can tell that the rightmost, topmost, and bottommost points must be on the convex hull. This identifies at least 2 and potentially 4 points on the hull. If we have 3 or 4 identified points, we can draw a triangle or a quadrilateral connecting them, and any point inside that quadrilateral must be inside the convex hull as well. That means we can exclude a (hopefully) large number of points from the sorting and scanning procedures, potentially speeding up the runtime significantly.

Lab Section Again, we're going to work on this week's homework problem during lab section:
Google Code Jam Qualification Round 2008 - Fly Swatter