2D Geometry Introduction (Part 2)

Point on Line

If we have the parametric equation of a line, it's not immediately obvious how to test if an arbitrary point is on the line or not. There is an easy way. Assume we have the line from P_0 to P_1

$$p = P_0 + t(P_1 - P_0)$$

Let's use the cross product to find the distance between a point P and the line. That formula is

$$d = \frac{\vec{v} \times \vec{l}}{|\vec{l}|}$$

where $\vec{v}$ is the vector from any point on the line to P, and $\vec{l}$ is a vector along the line. For our line, P_0 is on the line and $P_1 - P_0$ is a vector on the line, both of which we have handy from the line equation. If the point we're testing is on the line, then we'd expect $d = 0$. Putting this all together, we get

$$\frac{(P - P_0)\times(P_1 - P_0)}{|P_1 - P_0|} = 0$$

Since we're checking against zero, we can lose the denominator and just check

$$(P - P_0)\times(P_1 - P_0) = 0$$

We can also think of this as computing the area of a triangle (without the $\frac{1}{2}$ factor) with vertices P, P_0, and P_1. If the area is zero, then the triangle is degenerate and the three points are colinear; ergo P is on the same line as P_0 and P_1.

Now think about the case where $P = (0, 1), P_0 = (0, 0), P_1 = (1, 0)$; i.e. $\overline{P_0P_1}$ is along the x-axis and P is up the y-axis. In this case $(P - P_0)\times(P_1 - P_0) = (0, 1)\times(1, 0) = -1$. We've discussed that the sign of the cross product, when used to compute area, is dictated by the order of the vertices of the triangle we're looking at. Another interpretation is that the sign tells us what *side* of a line a point is on. Here, if we're looking down the x-axis, the point P will be on the left, and in general if we compute the cross product in this same form, a negative result will always mean the point is to the left of the line, and a positive result will mean the point is on the right side of the line.

Pick's Theorem

A *lattice point* is a point with integer coordinates. Pick's theorem quantifies the number of lattice points contained in a polygon. It works for any simple polygon (i.e. the polygon can be concave, but it can't have any holes in it.) Let A be the area of the polygon, B be the number of lattice points exactly on the boundary of the polygon, and I be the number of lattice points strictly on the interior of the polygon. Pick's theorem then states

$$A = \frac{B}{2} + I - 1$$

In the last lecture, we saw an easy way of computing the area of the polygon. Computing B, the number of points on the boundary, is also easy with a similar formula

$$B = \sum_{i=0}^{n} \gcd(x_{(i+1)\%n} - x_i, y_{(i+1)\%n} - y_i)$$

Example

Compute the number of lattice points interior to the polygon in figure 1 using Pick's theorem.

The area of the parallelogram is 24. (We can use the polygon area algorithm to get this, but the area of a parallelogram is just base times height. If we take the base to be the left vertical edge, it is 4, and the height from that to the right edge is 6). There are 12 boundary points.

If we solve Pick's theorem for I, we get

$$I = A - \frac{B}{2} + 1$$

which works out to $24 - \frac{12}{2} + 1 = 19$; we can verify this by counting in the diagram.

Point in Polygon

A problem that comes up often is to see if a given point is inside or outside of a polygon. There are many ways to accomplish this; we'll talk about one of the most popular, the edge intersection test.

The basic idea of the edge intersection test is that, if you draw a line from a point in any direction and count the number of edges of the polygon it intersects, that count will tell you if you started inside or outside the polygon. If the number of intersected edges is *odd*, then the point is inside the polygon; if it's *even*, then the point is outside. This is easy to see if you imagine a fully convex polygon; in that case, you will either intersect any one edge (inside), or no edges (outside). If the polygon has concavities, then then the line may intersect multiple

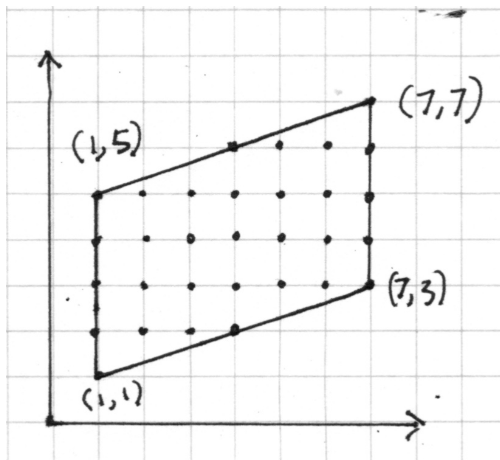

Figure 1: Pick's Theorem

edges, but the parity of the intersection count will still correctly identify if the point is in the interior.

Figure 2 shows this concept and some of the inevitable edge cases that arise. Here we are using horizontal lines in the positive x direction from our test points. A line that starts at a point and goes to infinity on just one side of the point is called a *ray*. For each consecutive pair of vertices, we intersect the ray with the line segment between those vertices, giving us two parametetric line equations:

$$P = P_n + t_1 x, 0 \leq t_1 \leq \infty$$

$$P = V_i + t_2(V_{(i+1)\%n} - V_i), 0 \leq t_2 \leq 1$$

(Note how the parametric line equation form trivially gives us tests that we're intersecting in the region of both lines that we care about). In the figure, we can see P_0's ray intersects one edge: $\overline{V_1 V_2}$; thus P_0 is inside the polygon. P_1's ray, however, intersects two edges: $\overline{V_2 V_3}$ and $\overline{V_3 V_4}$, so P_1 is outside the polygon.

There are three edge cases to think about: the ray intersects a vertex exactly, the ray and and edge happen to be along the same line, and the point we're testing is on the polygon boundary (technically this is two cases, the point could be on an edge or on a vertex).

If the ray intersects a vertex, it is unforunately not clear without more data if the intersection should be considered in the count or not. In the figure, assume we have a ray parallel to the others that intersects V_3. That ray is on the outside of the polygon, because V_3's neighbors V_2 and V_4 are both on one side of the ray. However, if we have a ray that goes through V_2, that ray *is* leaving the polygon, because V_2's neighboring vertices are on either side of the ray. The best way to deal with this situation is to nudge the direction of our ray slightly so that it doesn't intersect a vertex. The ray's direction is totally arbitrary, and this is typically much easier than going down the rabbit hole of more tests of the vertices against the ray.

If a ray overlays an edge, such as P_2 in the figure, then the intersection should not be counted. P_2 intersects, otherwise, only $\overline{V_2V_3}$, which properly indentifies it as inside the polygon. If we imagine P_2 to be farther down so that it intersects $\overline{V_0V_1}$, then it otherwise intersects no other edge, properly classifying it as an exterior point. The only case of interest is if the origin of the ray is inside the bounds of an edge (e.g. if we had V_5, P_2, V_4 all in that order on the line), in which case the point under test is actually on the polygon's boundary, neither interior nor exterior.

Finally, if t_1 in our equations above is zero for any edge we test against, then the point we're testing is on the polygon boundary. If t_2 is either zero or one, then the point is actually a vertex of the polygon; else it's on top of one of the edges.

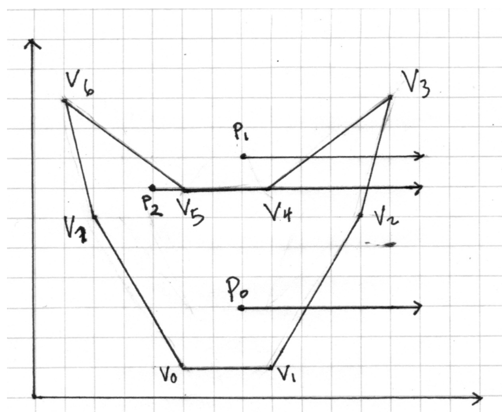

Figure 2: Point in Polygon Test: Edge Intersection

Convex Hull

Given a set of points in the plane, a *convex hull* of the set is a convex polygon, whose vertices are part of the set, with the property that no vertex in the set is exterior to the polygon. More intuitively, imagine stretching a rubber band large enough to contain all the points in the set. The rubber band traces out the convex hull of the set. Figure 3 shows a simple convex hull. Note that point P_1 is not part of the hull. If it were included, the hull would still contain all of the points, but it would no longer be a convex polygon.

The convex hull comes up in many applications and is well studied. There are many algorithms to compute it; we will look at the Graham Scan, which is efficient yet still fairly simple to implement.

The basic of the Graham scan is to add the points in the set, one at a time, to the hull; if adding a point creates a piece of hull that violates the hull constraints, we remove previously

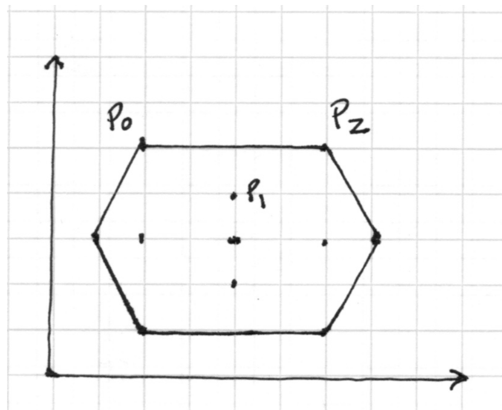

Figure 3: A Convex Hull

added points until the hull is correct again. Key to this idea is visiting the points in a reasonable order. The algorithm has three steps:

1. Find a point P_0 on the hull.
2. Sort the points radially about P_0.
3. Visit the sorted points, adding them to the hull as appropriate.

Figure 4 shows the first few steps of the algorithm.

Finding P_0

Any point that has an extreme x or y value will be on the hull. One reasonable choice is the leftmost point with the lowest y value.

Sorting the points

We want to visit the points counterclockwise around P_0. There are multiple ways to accomplish this, including computing angles between P_0 and each other point. But there's an easier way. If we think about the candidate point P we want to choose to follow P_0, all other points in the set will be to the left of $\overline{PP_0}$. If we think about rotating a line counterclockwise about P_0 through the points as we add them, the next point to add will always be the point that is left of that line but right of all other lines. We've already seen how to test for left-ness of a point relative to a line – the sign of the cross product.

There's one edge case: if the cross product is zero. That means that there are two points colinear with P_0. In this case, we want to choose the point closer (by standard Euclidean distance) to P_0 first.

2D Geometry Introduction (Part 2)

Scanning the hull

We would like to always have at least 2 points on the stack. The algorithm's invariant is that all points on the stack are on the hull, going counterclockwise. We know P_0 is on the hull. We also know that P_{n-1} is the point preceding it on the hull, as it's at the most extreme angle going counterclockwise.

The scan proceeds from P_1 to P_{n-1}. We will call the point we're looking at P_{next}, the point on the top of the stack P_{curr}, and the point before that on the stack P_{prev}. If P_{curr} is left of the line $\overline{P_{prev}P_{next}}$, then we pop it off the stack. Such a point will cause a concavity. In figure 4, we can see this with P_2 as we try to add P_3; if we don't discard P_2 then it will cause a concavity in the hull. We may need to pop off multiple points. When we discover a correct configuration, we push P_{next} and continue.

At the end of the scan, we will end up pushing and leaving P_{n-1} on the stack. We already pushed it when we started, so we pop the stack once at the very end to discard it.

Runtime

Figure 5 shows the pseudocode. We can analyze each of the parts of the algorithm in turn. Finding P_0 is just a linear scan of the points; hence it is $O(n)$. Sorting the points radially can be done with any sorting algorithm, which is $O(n\,lg\,n)$. The final scan examines every point once. There are at most n pops off of the stack, since once a point is popped off the stack, it will never be pushed again; the scan is $O(n)$.

The algorithm is dominated by the time for the sort, and is $O(n\,lg\,n)$ overall.

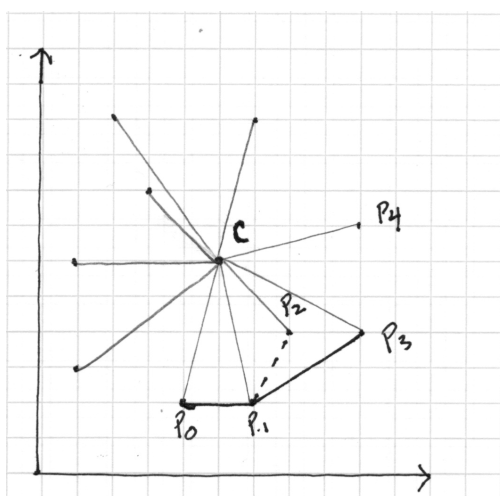

Figure 4: The Start of Graham Scan

```
                                                         ▷ Tests if P left of P0P1
function LeftOrOn(P, P0, P1)
    return (P − P0)×(P1 − P0) ≤ 0
end function
                                                         ▷ Computes the convex hull of S
function GrahamScan(S)
    P0 ← right-most lowest point
    Sort S counterclockwise around P0
    H ← newstack()
    H.push(P_{n−1})
    H.push(P0)
    i ← 1
    while i < n do
        if LeftOrOn(H.top(), H.top() − 1, S[i]) then
            H.pop()
        else
            H.push(S[i])
            i ← i + 1
        end if
    end while
    H.pop()
    return H
end function
```

Figure 5: Graham Scan