Inclusion of a Point in a Polygon

by Dan Sunday

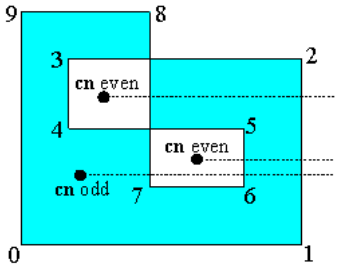
Determining the inclusion of a point P in a 2D planar polygon is a geometric problem that results in interesting algorithms. Two commonly used methods are:

1. The ***Crossing Number*** **(cn)** method
   - which counts the number of times a ray starting from the point P crosses the polygon boundary edges. The point is outside when this "crossing number" is even; otherwise, when it is odd, the point is inside. This method is sometimes referred to as the "even-odd" test.

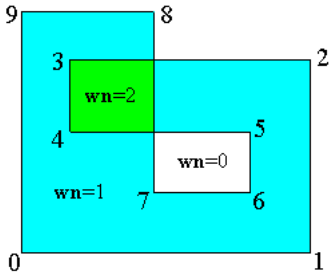
2. The ***Winding Number*** **(wn)** method
   - which counts the number of times the polygon winds around the point P. The point is outside only when this "winding number" wn = 0; otherwise, the point is inside.

If a polygon is simple (i.e., it has no self intersections), then both methods give the same result for all points. But for non-simple polygons, the two methods can give different answers for some points. For example, when a polygon overlaps with itself, then points in the region of overlap are found to be outside using the crossing number, but are inside using the winding number, as shown in the diagrams:

Crossing Number Method



Winding Number Method



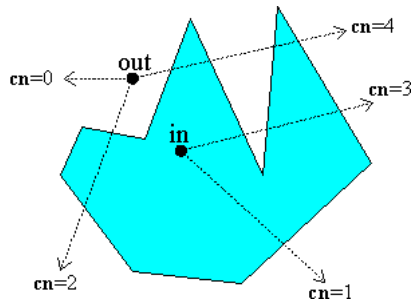
Vertices are numbered: 0 1 2 3 4 5 6 7 8 9

In this example, points inside the overlap region have **wn** = 2, implying that they are inside the polygon twice. Clearly, the winding number gives a better intuitive answer than the crossing number does.

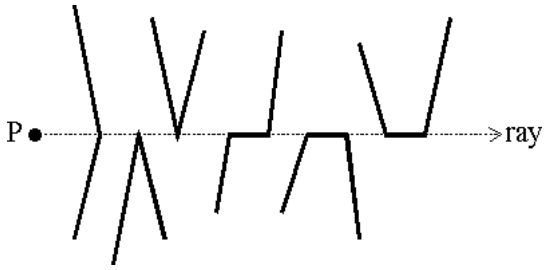
Despite this, the crossing number method is more commonly used since **cn** is erroneously thought to be significantly (up to 20 times!) more efficient to compute than **wn** [O'Rourke, 1998]. But this is not the case, and **wn** can be computed with the same efficiency as **cn** by counting signed crossings. In fact, our implementation for wn_PnPoly() is faster than the code for **cn** given by [Franklin, 2000], [Haines, 1994] or [O'Rourke, 1998], although the **cn** "PointInPolygon()" routine of [Moller & Haines, 1999] is comparable to our **wn** algorithm. But the bottom line is that for both geometric correctness and efficiency reasons, the **wn** algorithm should always be preferred for determining the inclusion of a point in a polygon.

The Crossing Number

This method counts the number of times a ray starting from a point P crosses a polygon boundary edge separating it's inside and outside. If this number is even, then the point is outside; otherwise, when the crossing number is odd, the point is inside. This is easy to understand intuitively. Each time the ray crosses a polygon edge, its in-out parity changes (since a boundary always separates inside from outside, right?). Eventually, any ray must end up beyond and outside the bounded polygon. So, if the point is inside, the sequence of crossings ">" must be: in > out > ... > in > out, and there are an odd number of them. Similarly, if the point is outside, there are an even number of crossings in the sequence: out > in > ... > in > out.



In implementing an algorithm for the **cn** method, one must insure that only crossings that change the in-out parity are counted. In particular, special cases where the ray passes through a vertex must be handled properly. These include the following types of ray crossings:



Further, one must decide whether a point on the polygon's boundary is inside or outside. A standard convention is to say that a point on a left or bottom edge is inside, and a point on a right or top edge is outside. This way, if two distinct polygons share a common boundary segment, then a point on that segment will be in one polygon or the other, but not both at the same time. This avoids a number of problems that might occur, especially in computer graphics displays.

A straightforward "crossing number" algorithm selects a horizontal ray extending to the right of P and parallel to the positive x-axis. Using this specific ray, it is easy to compute the intersection of a polygon edge with it. It is even easier to determine when no such intersection is possible. To count the total crossings, **cn**, the algorithm simply loops through all edges of the polygon, tests each for a crossing, and increments **cn** when one occurs. Additionally, the crossing tests must handle the special cases and points on an edge. This is accomplished by the

Edge Crossing Rules

1. an upward edge includes its starting endpoint, and excludes its final endpoint;
2. a downward edge excludes its starting endpoint, and includes its final endpoint;
3. horizontal edges are excluded
4. the edge-ray intersection point must be strictly right of the point P.

One can apply these rules to the preceding special cases, and see that they correctly determine valid crossings. Note that Rule #4 results in points on a right-side boundary edge being outside, and ones on a left-side edge being inside.

Pseudo-Code: Crossing # Inclusion

Code for this algorithm is well-known, and the edge crossing rules are easily expressed. For a polygon represented as an array $V[n+1]$ of vertex points with $V[n]=V[0]$, popular implementation logic ([Franklin, 2000], [O'Rourke, 1998]) is as follows:

```
typedef struct {int x, y;} Point;

cn_PnPoly( Point P, Point V[], int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (each edge E[i]:V[i]V[i+1] of the polygon) {
        if (E[i] crosses upward ala Rule #1
         || E[i] crosses downward ala  Rule #2) {
            if (P.x <  x_intersect of E[i] with y=P.y)   // Rule #4
                 ++cn;   // a valid crossing to the right of P.x
        }
    }
```

```
        return (cn&1);    // 0 if even (out), and 1 if  odd (in)

    }
```

Note that the tests for upward and downward crossing satisfying Rules #1 and #2 also exclude horizontal edges (Rule #3). All-in-all, a lot of work is done by just a few tests which makes this an elegant algorithm.

However, the validity of the crossing number method is based on the "Jordan Curve Theorem" which says that a simple closed curve divides the 2D plane into exactly 2 connected components: a bounded "inside" one and an unbounded "outside" one. The catch is that the curve must be simple (without self intersections), otherwise there can be more than 2 components and then there is no guarantee that crossing a boundary changes the in-out parity. For a closed (and thus bounded) curve, there is exactly one unbounded "outside" component; but bounded components can be either inside or outside. And two of the bounded components that have a shared boundary may both be inside, and crossing over their shared boundary would not change the in-out parity.

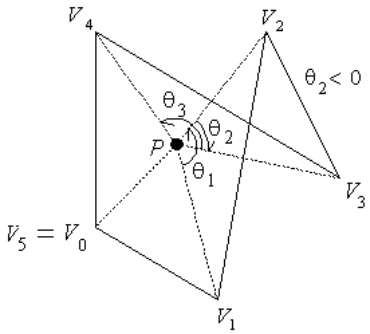
The Winding Number

On the other hand, the winding number accurately determines if a point is inside a nonsimple closed polygon. It does this by computing how many times the polygon winds around the point. A point is outside only when the polygon doesn't wind around the point at all which is when the winding number **wn** = 0. More generally, one can define the winding number $\mathbf{wn}(P,\mathbf{C})$ of any closed continuous curve **C** around a point P in the 2D plane. Let the continuous 2D curve **C** be defined by the points $C(u)=C(x(u),y(u))$, for $0 \le u \le 1$ with $C(0)=C(1)$. And let P be a point not on **C**. Then, define the vector $\mathbf{c}(P,u) = C(u) - P$ from P to $C(u)$, and the unit vector $\mathbf{w}(P,u) = \mathbf{c}(P,u) / |\mathbf{c}(P,u)|$ which gives a continuous function $W(P):\mathbf{C}\rightarrow\mathbf{S}^1$ mapping the point $C(u)$ on **C** to the point $\mathbf{w}(P,u)$ on the unit circle $\mathbf{S}^1=\{(x,y)|\,x^2+y^2=1\}$. This map can be represented in polar coordinates as $W(P)(u)=(\cos\theta(u),\sin\theta(u))$ where $\theta(u)$ is a positive counterclockwise angle in radians. The winding number $\mathbf{wn}(P,\mathbf{C})$ is then equal to the integer number of times that $W(P)$ wraps **C** around $\mathbf{S}^1$. This corresponds to a homotopy class of $\mathbf{S}^1$, and can be computed by the integral:

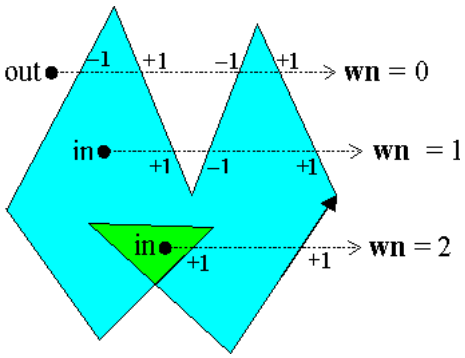
$$\mathbf{wn}(P,\mathbf{C})=\frac{1}{2\pi}\oint_{W(P)} d\theta=\frac{1}{2\pi}\int_{u=0}^{1}\theta(u)du$$

When the curve **C** is a polygon with vertices $V_0,V_1,...,V_n=V_0$, this integral reduces to the sum of the (signed) angles that each edge V_iV_{i+1} subtends with the point **P**. So, if $\theta_i=\angle(PV_i,PV_{i+1})$, we have:

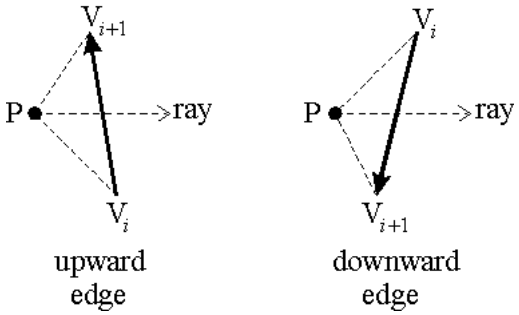
$$\begin{aligned}\mathbf{wn}(P,\mathbf{C}) &= \frac{1}{2\pi}\sum_{i=0}^{n-1}\theta_i \\ &= \frac{1}{2\pi}\sum_{i=0}^{n-1}\arccos\left(\frac{(V_i-P)\bullet(V_{i+1}-P)}{|(V_i-P)||(V_{i+1}-P)|}\right)\end{aligned}$$



This formula is clearly not very efficient since it uses a computationally expensive arccos() trig function. But, a simple observation lets us replace this formula by a more efficient one. Pick any point Q on $\mathbf{S}^1$. Then, as the curve *W(P)* wraps around $\mathbf{S}^1$, it passes Q a certain number of times. If we count (+1) when it passes Q counterclockwise, and (–1) when it passes clockwise, then the accumulated sum is exactly the total number of times that *W(P)* wraps around $\mathbf{S}^1$, and is equal to the winding number $\mathbf{wn}(P,\mathbf{C})$. Further, if we take an infinite ray **R** starting at P and extending in the direction of the vector Q, then intersections where **R** crosses the curve **C** correspond to the points where *W(P)* passes Q. To do the math, we have to distinguish between positive and negative crossings where **C** crosses **R** from right-to-left or left-to-right. This can be determined by the sign of the dot product between a normal vector to **C** and the direction vector $\mathbf{q} = Q$ [Foley et al, 1996, p-965], and when the curve **C** is a polygon, one just needs to make this determination once for each edge. For a horizontal ray **R** from P, testing whether an edge's endpoints are above and below the ray suffices. If the edge crosses the positive ray from below to above, the crossing is positive (+1); but if it crosses from above to below, the crossing is negative (–1). One then simply adds all crossing values to get $\mathbf{wn}(P,\mathbf{C})$. For example:



Additionally, one can avoid computing the actual edge-ray intersection point by using the isLeft() attribute; however, it needs to be applied differently for ascending and descending edges. If an upward edge crosses the ray to the right of P, then P is on the left side of the edge since the triangle $V_iV_{i+1}P$ is oriented counterclockwise. On the other hand, a downward edge crossing the positive ray would have P on the right side since the triangle $V_iV_{i+1}P$ would then be oriented clockwise.



Pseudo-Code: Winding Number Inclusion

This results in the following **wn** algorithm which is an adaptation of the **cn** algorithm and uses the same edge crossing rules as before to handle special cases.

```
typedef struct {int x, y;} Point;

wn_PnPoly( Point P, Point V[], int n )
{
    int    wn = 0;    // the  winding number counter

    // loop through all edges of the polygon
    for (each edge E[i]:V[i]V[i+1] of the polygon) {
        if (E[i] crosses upward ala Rule #1)  {
            if (P is  strictly left of E[i])    // Rule #4
                ++wn;   // a valid up intersect right of P.x
        }
        else
        if (E[i] crosses downward ala Rule  #2) {
            if (P is  strictly right of E[i])   // Rule #4
                --wn;   // a valid down intersect right of P.x
        }
    }
    return wn;    // =0 <=> P is outside the polygon

}
```

Clearly, this winding number algorithm has the same efficiency as the analogous crossing number algorithm. Thus, since it is more accurate in general, the winding number algorithm should always be the preferred method to determine inclusion of a point in an arbitrary polygon.

The **wn** algorithm's efficiency can be improved further by rearranging the crossing comparison tests. This is shown in the detailed implementation of wn_PnPoly() given below. In that code, all edges that are totally above or totally below P get rejected after only two (2) inequality tests. However, currently popular implementations of the **cn** algorithm ([Franklin, 2000], [Haines, 1994], [O'Rourke, 1998]) use *at least* three (3) inequality tests for each rejected edge. Since most of the edges in a large polygon get rejected in practical applications, there is about a 33% (or more) reduction in the number of comparisons done. In runtime tests using very large (1,000,000 edge) random polygons (with edge length < 1/10 the polygon diameter) and 1000 random test points (inside the polygon's bounding box), we measured a 20% increase in efficiency overall.

Enhancements

There are some enhancements to point in polygon algorithms [Haines, 1994] that software developers should be aware of. We mention a few that pertain to ray crossing algorithms. However, there are other techniques that give better performance in special cases such as testing inclusion in small convex polygons like triangles. These are discussed in [Haines, 1994].

Bounding Box or Ball

It is efficient to first test that a point P is inside the bounding box or ball of a large polygon before testing all edges for ray crossings. If a point is outside the bounding box or ball, it is also outside the polygon, and no further testing is needed. But, one must precompute the bounding box (the max and min for vertex x and y coordinates) or the bounding ball (center and minimum radius) and store it for future use. This is worth doing if more than a few points are going to be tested for inclusion, which is generally the case. Further information about computing bounding containers can be found in Algorithm 8.

3D Planar Polygons

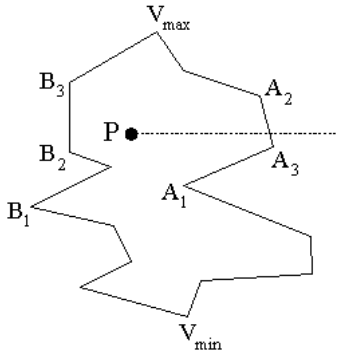
In 3D applications, one sometimes wants to test a point and polygon that are in the same plane. For example, one may have the intersection point of a ray with the plane of a polyhedron's face, and want to test if it is inside the face. Or one may want to know if the base of a 3D perpendicular dropped from a point is inside a planar polygon.

3D inclusion is easily determined by projecting the point and polygon into 2D. To do this, one simply ignores one of the 3D coordinates and uses the other two. To optimally select the coordinate to ignore, compute a normal vector to the plane, and select the coordinate with the largest absolute value [Snyder & Barr, 1987]. This gives the projection of the polygon with maximum area, and results in robust computations.

Convex or Monotone Polygons

When a polygon is known to be convex, many computations can be speeded up. For example, the bounding box can be computed in O(log n) time [O'Rourke, 1998] instead of O(n) as for nonconvex polygons. Also, point inclusion can be tested in O(log n) time. More generally, the following algorithm works for convex or monotone polygons.

A convex or y-monotone polygon can be split into two polylines, one with edges increasing in y, and one with edges decreasing in y. The split occurs at two vertices with max y and min y coordinates. Note that these vertices are at the top and bottom of the polygon's bounding box, and if they are both above or both below P's y-coordinate, then the test point is outside the polygon. Otherwise, each of these two polylines intersects a ray paralell to the x-axis once, and each potential crossing edge can be found by doing a binary search on the vertices of each polyline. This results in a practical O(log n) (preprocessing and runtime) algorithm for convex and montone polygon inclusion testing. For example, in the following diagram, only three binary search vertices have to be tested on each polyline ($A1$, $A2$, $A3$ ascending; and $B1$, $B2$, $B3$ descending):



This method also works for polygons that are monotone in an arbitrary direction. One then uses a ray from P that is perpendicular to the direction of monotonicity, and the algorithm is easily adapted. A little more computation is needed to determine which side of the ray a vertex is on, but for a large enough polygon, the O(log n) performance more than makes up for the overhead.

Implementations

Here is a "C++" implementation of the winding number algorithm for the inclusion of a point in polygon. We just give the 2D case, and use the simplest structures for a point and a polygon which may differ in your application.

```
// Copyright 2000 softSurfer, 2012 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.


// a Point is defined by its coordinates {int x, y;}
//===================================================================


// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2  on the line
//            <0 for P2  right of the line
//    See: Algorithm 1 "Area of Triangles and Polygons"
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x -  P0.x) * (P1.y - P0.y) );
}
//===================================================================


// cn_PnPoly(): crossing number test for a point in a polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1] with V[n]=V[0]
//      Return:  0 = outside, 1 = inside
// This code is patterned after [Franklin, 2000]
int
cn_PnPoly( Point P, Point* V, int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i]  to V[i+1]
       if (((V[i].y <= P.y) && (V[i+1].y > P.y))     // an upward crossing
        || ((V[i].y > P.y) && (V[i+1].y <=  P.y))) { // a downward crossing
            // compute  the actual edge-ray intersect x-coordinate
            float vt = (float)(P.y  - V[i].y) / (V[i+1].y - V[i].y);
            if (P.x <  V[i].x + vt * (V[i+1].x - V[i].x)) // P.x < intersect
                 ++cn;   // a valid crossing of y=P.y right of P.x
        }
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}
//===================================================================


// wn_PnPoly(): winding number test for a point in a polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1] with V[n]=V[0]
//      Return:  wn = the winding number (=0 only when P is outside)
int
wn_PnPoly( Point P, Point* V, int n )
{
    int    wn = 0;    // the  winding number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {   // edge from V[i] to  V[i+1]
        if (V[i].y <= P.y) {          // start y <= P.y
            if (V[i+1].y  > P.y)      // an upward crossing
                 if (isLeft( V[i], V[i+1], P) > 0)  // P left of  edge
                     ++wn;            // have  a valid up intersect
        }
        else {                        // start y > P.y (no test needed)
            if (V[i+1].y  <= P.y)     // a downward crossing
                 if (isLeft( V[i], V[i+1], P) < 0)  // P right of  edge
                     --wn;            // have  a valid down intersect
        }
    }
    return wn;
```

```
}
//===================================================================
```

References

James Foley, Andries van Dam, Steven Feiner & John Hughes, "Filled Primitives" in Computer Graphics (3rd Edition) (2013)

Wm. Randolph Franklin, "PNPOLY - Point Inclusion in Polygon Test" Web Page (2000)

Eric Haines, "Point in Polygon Strategies" in Graphics Gems IV (1994)

Tomas Moller & Eric Haines, "Ray/Polygon Intersection" in Real-Time Rendering (3rd Edition) (2008)

Joseph O'Rourke, "Point in Polygon" in Computational Geometry in C (2nd Edition) (1998)

John M. Snyder & Alan H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", Computer Graphics 21(4), 119-126 (1987) [also in the Proceedings of SIGGRAPH 1987]