◊ I.4

Point in Polygon Strategies

Eric Haines
3D/Eye Inc.
1050 Craft Road
Ithaca, NY 14850
erich@eye.com

Testing whether a point is inside a polygon is a basic operation in computer graphics. This Gem presents a variety of efficient algorithms. No single algorithm is the best in all categories, so the capabilities of the better algorithms are compared and contrasted. The variables examined are the different types of polygons, the amount of memory used, and the preprocessing costs. Code is included in this article for several of the best algorithms; the Gems IV distribution includes code for all the algorithms discussed.

◊ Introduction ◊

The motivation behind this Gem is to provide practical algorithms that are simple to implement and are fast for typical polygons. In applied computer graphics we usually want to check a point against a large number of triangles and quadrilaterals and occasionally test complex polygons. When dealing with floating-point operations on these polygons we do not care if a test point exactly on an edge is classified as being inside or outside, since these cases are normally extremely rare.

In contrast, the field of computational geometry has a strong focus on the order of complexity of an algorithm for all polygons, including pathological cases that are rarely encountered in real applications. The order of complexity for an algorithm in computational geometry may be low, but there is usually a large constant of proportionality or the algorithm itself is difficult to implement. Either of these conditions makes the algorithm unfit for use. Nonetheless, some insights from computational geometry can be applied to the testing of various sorts of polygons and can also shed light on connections among seemingly different algorithms.

Readers that are only interested in the results should skip to the "Conclusions" section.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

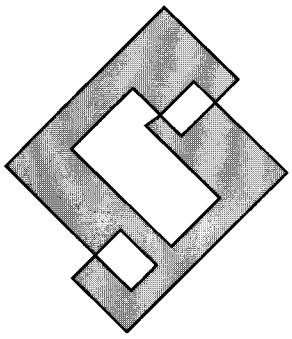

Figure 1. Jordan curve.

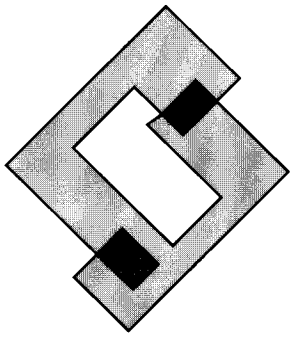

Figure 2. Winding number.

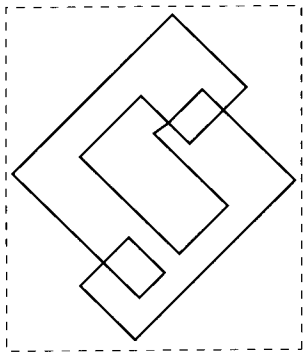

Figure 3. Bounding box.

◇ Definitions ◇

In this Gem a polygon is defined by an ordered set of vertices which form edges making a closed loop. The first and last vertices are connected by an edge, i.e., they are not the same. More complex objects, such as polygons with holes for font lettering, can be built from these polygons by applying the point in polygon test to each loop and concatenating the results.

There are two main types of polygons we will consider in this Gem: general and convex. If a number of points are to be tested against a polygon, it may be worthwhile determining whether the polygon is convex at the start so you are able to use a faster test. General polygons have no restrictions on the placement of vertices. Convex polygon determination is discussed in another Gem in this volume (Schorn and Fisher 1994). If you do not read this other Gem, at least note that a polygon with no concave angles is not necessarily convex; a good counterexample is a star formed by five vertices.

One definition of whether a point is inside a region is the *Jordan Curve Theorem*, also known as the parity or even-odd test. Essentially, it says that a point is inside a polygon if, for any ray from this point, there is an odd number of crossings of the ray with the polygon's edges. This definition means that some areas enclosed by a polygon are not considered inside (Figure 1).

If the entire area enclosed by the polygon is to be considered inside, then the *winding number* is used for testing. This value is the number of times the polygon goes around the point. In Figure 2 the darkly shaded areas have a winding number of two. Think of the polygon as a loop of string pulled tight around a pencil point; the number of loops around the point is the winding number. If a point is outside, the polygon does not wind around it and so the winding number is zero. Winding numbers also have a sign, which corresponds to the direction the edges wrap around the point. The winding

number test can be converted to the parity test; an odd winding number is equivalent to the parity test's inside condition.

In ray tracing and other applications the original polygon is three-dimensional. To simplify computation it is worthwhile to project the polygon and test point into two dimensions. One way to do this is simply to ignore one coordinate. The best coordinate to drop is the one that yields the largest area for the 2D polygon formed. This is easily done by taking the absolute value of each coordinate of the polygon plane's normal and finding the largest; the corresponding coordinates are ignored (Glassner 1989). Precomputing some or all of this information once for a polygon uses more memory but increases the speed of the intersection test itself.

Point in polygon algorithms often benefit from having a bounding box around polygons with many edges. The point is first tested against this box before the full polygon test is performed; if the box is missed, so is the polygon (Figure 3). Most statistics generated in this Gem assume this bounding box test was already passed successfully.

In ray tracing, (Worley and Haines 1993) points out that the polygon's 3D bounding box can be treated like a 2D bounding box by throwing away one coordinate, as done above for polygons. By analysis of the operations involved, it can be shown to be more profitable in general to first intersect the polygon's plane and then test whether the point is inside the 2D bounding box, rather than first testing the 3D bounding box and then the plane. Other bounding box variants can be found in (Woo 1992).

◇ General Algorithms ◇

This section discusses the fastest algorithms for testing points against general polygons. Three classes of algorithms are compared: those which use the vertex list as their only data structure, those which do preprocessing and create an alternate form of the polygon, and those which create additional efficiency structures. The advantages of a vertex list algorithm is that no additional information or preprocessing is needed. However, the other two types of algorithms offer faster testing times in many cases.

Crossings Test

The fastest algorithm without any preprocessing is the crossings test. The earliest presentation of this algorithm is (Shimrat 1962), though it has a bug in it, corrected by (Hacker 1962). A ray is shot from the test point along an axis (+X is commonly used), and the number of crossings is computed for the even-odd test (Figure 4). One way to think about this algorithm is to consider the test point to be at the origin and to check the edges against this point. If the Y coordinates of a polygon edge differ in sign, then the edge can cross the test ray. In this case, if both X coordinates are positive, the edge

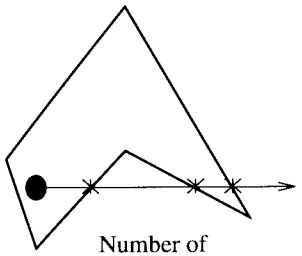

Figure 4. Crossings test.

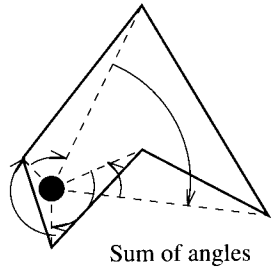

Figure 5. Angle summation.

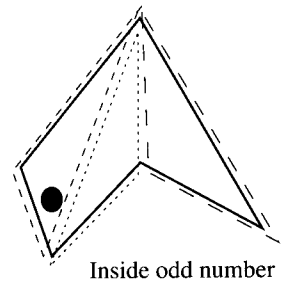

Figure 6. Triangle fan.

and ray must intersect and a crossing is recorded. Else, if the X signs differ, then the X intersection of the edge and the ray is computed and if positive a crossing is recorded.

What happens when the test ray intersects one or more vertices of the polygon? This problem can be ignored by considering the test ray to be a half-plane divider, with one of the half-planes including the ray's points (Preparata and Shamos 1985, Glassner 1989). In other words, whenever the ray would intersect a vertex, the vertex is always classified as being infinitesimally above the ray. In this way, no vertices are considered intersected and the code is both simpler and speedier.

MacMartin pointed out that for polygons with a large number of edges there are generally runs of edges that have Y coordinates with the same sign (Haines 1992). For example, a polygon representing Brazil might have a thousand edges, but only a few of these will straddle a given latitude line and there are long runs of contiguous edges on one side of this line. So a faster strategy is to loop through just the Y coordinates as fast as possible; when they differ then retrieve and check the X coordinates.

Either the even-odd or winding number test can be used to classify the point. The even-odd test is done by simply counting the number of crossings. The winding number test is computed by keeping track of whether the crossed edge passes from the Y- to the Y+ half-plane (add 1) or vice versa (subtract 1). The final value is then the number of counterclockwise windings about the point.

The slowest algorithm for testing points is by far the pure angle summation method. It's simple to describe: sum the signed angles formed at the point by each edge's endpoints (Figure 5). The winding number can then be computed by finding the nearest multiple of 360 degrees. The problem with this pure scheme is that it involves a large number of costly math function calls.

However, the idea of angle summation can be used to formulate a fast algorithm for testing points; see Weiler's Gem in this volume (Weiler 1994). There is a strong

connection between Weiler's algorithm and the crossings test. Weiler avoids expensive trigonometry computations by adding or subtracting one or more increments of 90 degrees as the loop vertices move from quadrant to quadrant (with the test point at the origin). The crossings test is similar in that it can be thought of as counting movements of 360 degrees when an edge crosses the test ray. The crossings test tends to be faster because it does not have to categorize and record all quadrant-to-quadrant movements but only those which cross the test ray. Weiler's formulation is significant for the way it adds to the understanding of underlying principles.

Triangle Fan Tests

In *Graphics Gems*, (Badouel 1990) presents a method of testing points against convex polygons. The polygon is treated as a fan of triangles emanating from one vertex and the point is tested against each triangle by computing its barycentric coordinates. As (Berlin 1985) points out, this test can also be used for non-convex polygons by keeping a count of the number of triangles that overlap the point; if odd, the point is inside the polygon (Figure 6). Unlike the convex test, where an intersection means that the test is done, all the triangles must be tested against the point for the non-convex test. Also, for the non-convex test there may be multiple barycentric coordinates for a given point, since triangles can overlap.

The barycentric test is faster than the crossings test for triangles but becomes quite slow for polygons with more edges. However, (Spackman 1993) notes that pre-normalizing the barycentric equations and storing a set of precomputed values gives better performance. This version of the algorithm is twice as fast as the crossings test for triangles and is in general faster for polygons with few edges. The barycentric coordinates (which are useful for interpolation and texture mapping) are also computed.

A faster triangle fan tester, proposed by (Green 1993), is to store a set of half-plane equations for each triangle and test each in turn. If the point is outside any of the three edges, it is outside the triangle. The half-plane test is an old idea, but storing the half-planes instead of deriving them on the fly from the vertices gives this scheme its speed at the cost of some additional storage space. For triangles this scheme is the fastest of all of the algorithms discussed so far. It is also very simple to code and so lends itself to assembly language translation. Theoretically the Spackman test should usually have a smaller average number of operations per test, but in practice the optimized code for the half-plane test is faster.

Both the half-plane and Spackman triangle testers can be sped up further by sorting the order of the edge tests. Worley and Haines (Spackman 1993) note that the half-plane triangle test is more efficient if the longer edges are tested first. Larger edges tend to cut off more exterior area of the polygon's bounding box and so can result in earlier

exit from testing a given triangle. Sorting in this way makes the test up to 1.7 times faster, rising quickly with the number of edges in the polygon. However, polygons with a large number of edges tend to bog down the sorted edge triangle algorithm, with the crossings test being faster above around 10 edges.

A problem occurs in general triangle fan algorithms when the code assumes that a point that lies on a triangle edge is always inside that triangle. For example, a quadrilateral is treated as two triangles. If a point is exactly on the edge between the two triangles it will be classified as being inside both triangles and so will be classified as being outside the polygon (this problem does not happen with the convex test).

The code presented for these algorithms does not fully address this problem. In reality, a random point tested against a polygon has an infinitesimal chance of landing exactly on any edge. For rendering purposes this problem can be ignored, with the result being one misshaded pixel once in a great while. A more robust solution (which will slow down the test) is to note whether an edge is to include the points exactly on it or not. Also, an option which has not been explored is to test shared interior edges only once against the point and share the results between the adjacent triangles.

Grid Method

An even faster, and more memory intensive, method of testing for points inside a polygon is lookup grids. The idea is to impose a grid inside the bounding box containing the polygon. Each grid cell is categorized as being fully inside, fully outside, or indeterminate. The indeterminate cells also have a list of edges that overlap the cell, and also one corner (or more) is determined to be inside or outside.

To test a point against this structure is extremely quick in most cases. For a reasonable polygon many of the cells are either inside or outside, so testing consists of a simple look-up. If the cell contains edges, then a line segment is formed from the test point to the cell corner and is tested against all edges in the list (Antonio 1992). Since the state of the corner is known, the state of the test point can be found from the number of intersections (Figure 7). Salesin and Stolfi suggest an algorithm similar to this as part of their ray tracing acceleration technique (Salesin and Stolfi 1989).

Care must be taken when a polygon edge exactly (or even nearly exactly) crosses a grid corner, as this corner is then unclassifiable. Rather than coping with the topological and numerical precision problems involved, one simple solution is to just start generating the grid from scratch again, giving slightly different dimensions to the bounding box. Also, when testing the line segment against the edges in a list, exact intersections of an edge endpoint must be counted only once.

One additional enhancement partially solves this problem. Each grid cell has four sides. If no polygon edges cross a side, then that side will be fully inside or outside the polygon. A horizontal or vertical test line segment can then be generated from the test

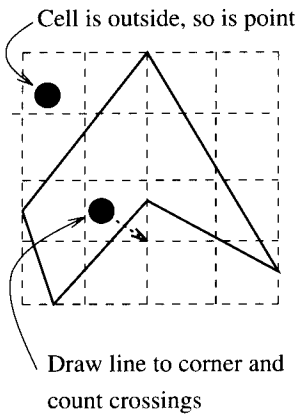

Figure 7. Grid crossings test.

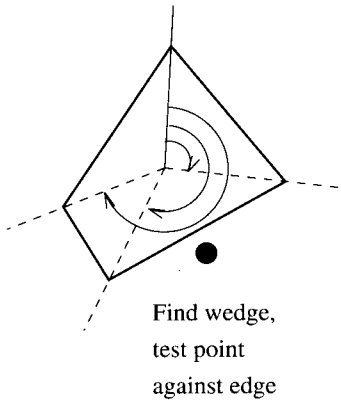

Figure 8. Inclusion test.

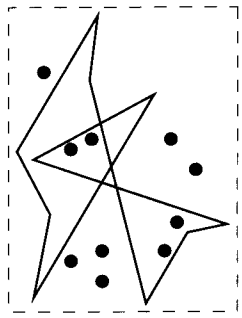

Figure 9. Random polygon.

point to this cell side and the faster crossings test can be used against the edges in the cell. In addition, this crossings test deals with endpoint intersection more robustly.

Note: the grid test code is in the Gems IV code distribution, but has been left out of the book because of its length.

Pixel Based Testing

One interesting case that is related to gridding is that of pixel-limited picking. When a dataset is displayed on the screen and a large amount of picking is to be done on a still image, a specialized test is worthwhile. Hanrahan and Haeberli note that the image can be generated once into a separate buffer, filling in each polygon's area with an identifying index (Hanrahan and Haeberli 1990). When a pixel is picked on this fixed image, it is looked up in this buffer and the polygon selected is known immediately.

◇ Convex Polygons ◇

Convex polygons can be intersected faster due to their geometric properties. For example, the crossings test can quit as soon as two Y-sign difference edges are found, since this is the maximum that a convex polygon can have. Also, note that more polygons can use this faster crossings test by checking only the change in the Y direction (and not X and Y as for the full convexity test); see (Schorn and Fisher 1994). For example, a block letter "E" has at most two Y intersections for any test point's horizontal line (and so is called *monotone* in Y), so it can use the faster crossings test.

The triangle fan tests can exit as soon as any triangle is found to contain the point. These algorithms can be enhanced by both sorting the edges of each triangle by length and also sorting the testing order of triangles by their areas. Relatively larger triangles are more likely to enclose a point and so end testing earlier. Note that this faster test can be applied to any polygon that is decomposed into non-overlapping triangles; convex polygons always have this property when tessellated into a triangle fan.

The exterior algorithm prestores the half-plane for each polygon edge and tests the point against this set. If the point is outside any edge, then the point must be outside the entire convex polygon. This algorithm uses less additional storage than the triangle fan and is very simple to code. The order of edges tested affects the speed of this algorithm; testing edges in the order of which cuts off the most area of the bounding box earliest on is the best ordering. Finding this optimal ordering is non-trivial, but doing the edges in order is often the worst strategy, since each neighboring edge usually cuts off little more area than the previous. Randomizing the order of the edges makes this algorithm up to 10% faster overall for regular polygons.

The exterior algorithm looks for an early exit due to the point being outside the polygon, while the triangle fan convex test looks for one due to the point being inside. For example, for 100 edge polygons, if all points tested are inside the polygon the triangle fan is 1.7 times faster; if all test points are outside the exterior test is more than 16 times faster (but only 4 times faster if the edges are not randomized). So when the polygon/bounding box area ratio is low the exterior algorithm is usually best; in fact, performance is near constant time as this ratio decreases, since after only a few edges most points are categorized as outside the polygon.

A hybrid of the exterior algorithm and the triangle fan is to test triangles and exit early when the point is outside the polygon. A point is outside the polygon if it is outside any exterior triangle edge. This strategy combines the early exit features of both algorithms and so it is less dependent on bounding box fit. Our code uses sorting by triangle area instead of randomizing the exterior edge order, so it favors a higher polygon/bounding box area ratio.

A method with O(log n) performance is the inclusion algorithm (Preparata and Shamos 1985). The polygon is preprocessed by adding a central point to it and is then divided into wedges. The angles from an anchor edge to each wedge's edges are computed and saved, along with half-plane equations for the polygon edges. When a point is tested, the angle from the anchor edge is computed and a binary search is used to determine the wedge it is in, and then the corresponding polygon edge is tested against it (Figure 8). Note that this test can be used on any polygon that can be tessellated into a non-overlapping star of triangles. This algorithm is slower for polygons with few edges because the startup cost is high, but the binary search makes for a much faster test when there are many edges. However, if the bounding box is much larger than the polygon the exterior edge test is faster.

◊ Statistics ◊

The timings given in Tables 1–3 were produced on an HP 720 RISC workstation; timings had similar performance ratios on an IBM PC 386 with no FPU. The general non-convex algorithms were tested using two sorts of polygons: those generated with random points and regular (i.e., equal length sides and vertex angles) polygons with a random rotation applied. Random polygons tend to be somewhat unlikely (no one ever uses 1000-edge random polygons for anything except testing), while regular polygons are more orderly than a "typical" polygon; normal behavior tends to be somewhere in between. Test points were generated inside the bounding box for the polygon. Figure 9 shows a typical 10-sided random polygon and some test points. Convex algorithms were tested with only regular polygons, and so have a certain bias to them.

Test points were generated inside the box bounding the polygon; looser fitting boxes yield different results. Timings are in microseconds per polygon. They are given to two significant figures, since their accuracy is roughly ±10%. However, the best way to get useful timings is to run the code on the target machine; there is a testbed program provided in the Gems IV code distribution which can be used to try new algorithms and generate timings under various test conditions. Also, of course, hacking the code for a particular machine and compiler can make a significant difference.

◊ Discussion ◊

The crossings test is generally useful, but we can do better. Testing triangles using either sorted triangle fan algorithm is more than twice as fast, though for polygons with many edges the crossings test is still faster.

Given enough resolution (and enough memory!), gridding gives near constant time performance for most normal polygons, though it performs a bit slower when entirely random polygons are tested. Interestingly, even for polygons with just a few edges the gridding algorithm outperforms most of the other tests.

Testing times can be noticeably decreased by using an algorithm optimized for convex testing when possible. For example, the convex sorted half-plane test is 1.4 times faster for 10-sided polygons than its general case counterpart. For convex polygons with many edges the inclusion test is extremely efficient because of its O(log n) behavior.

Other algorithms remain to be discovered and tested; for example, a practical general polygon algorithm with better than O(n) performance and low storage costs would fill a useful niche.

Table 1. General Algorithms, Random Polygons

Algorithm	Number of edges per polygon						
	3	**4**	**10**	**20**	**50**	**100**	**1000**
Crossings	2.8	3.1	5.7	10	25	48	470
Half Plane w/edge sort	1.1	1.7	5.7	12	32	65	650
Half Plane, no sort	1.2	2.0	6.3	14	36	72	740
Spackman w/edge sort	1.3	2.1	6.0	13	32	66	670
Spackman, no sort	1.4	2.2	6.4	14	35	70	720
Barycentric	2.4	4.0	13	29	76	150	1600
Weiler angle	3.7	4.3	8.7	16	39	77	760
Trigonometric angle	42	51	110	210	520	1030	10300
Grid (100x100)	1.8	1.9	1.9	1.9	2.2	2.5	9.2
Grid (20x20)	2.0	2.0	2.2	2.5	3.6	5.5	38

Table 2. General Algorithms, Regular Polygons

Algorithm	Number of edges per polygon						
	3	**4**	**10**	**20**	**50**	**100**	**1000**
Crossings	2.6	2.7	4.3	7.2	16	32	300
Half Plane w/edge sort	1.3	1.8	4.6	9.2	23	45	460
Half Plane, no sort	1.3	2.1	6.7	14	37	74	760
Spackman w/edge sort	1.5	2.1	5.4	10	26	51	510
Spackman, no sort	1.5	2.3	5.8	11	28	55	550
Barycentric	2.5	4.2	13	26	68	140	1400
Weiler angle	3.5	4.0	7.9	15	35	70	690
Trigonometric angle	39	51	120	230	560	1200	11100
Grid (100x100)	1.8	1.8	1.8	1.8	1.8	1.8	1.9
Grid (20x20)	2.0	2.0	2.0	2.0	2.0	2.1	2.8

Table 3. Convex Algorithms, Regular Polygons

Algorithm	Number of edges per polygon						
	3	**4**	**10**	**20**	**50**	**100**	**1000**
Inclusion	5.5	5.7	6.3	6.6	7.1	7.6	9.9
Hybrid Sorted Half Plane	1.3	1.6	3.3	6.1	14	28	280
Sorted Half Plane	1.2	1.6	3.4	6.2	15	29	280
Unsorted Half Plane	1.2	1.9	5.7	12	30	61	620
Random Exterior Edges	1.3	1.7	3.8	7.1	17	33	320
Ordered Exterior Edges	1.3	1.7	3.8	7.3	18	35	350
Convex Crossings	2.5	2.5	3.6	5.6	12	22	220

◇ Conclusions ◇

- If no preprocessing nor extra storage is available, use the **Crossings** test.
- If a little preprocessing and extra storage is available:
  - For general polygons
    - with few sides, use the **Half-Plane** or **Spackman** test.
    - with many sides, use the **Crossings** test.
  - For convex polygons
    - with few sides, use the **Hybrid Half-Plane** test.
    - with many sides, use the **Inclusion** test.
    - But if the bounding box/polygon area ratio is high, use the **Exterior Edges** test.
- If preprocessing and extra storage is available in abundance, use the **Grid Test** (except for perhaps triangles).

Of course, some of these conclusions may vary with machine architecture and compiler optimization.

◇ C Code ◇

ptinpoly.h

```
/* ptinpoly.h - point in polygon inside/outside algorithms header file.
 */

/* Define CONVEX to compile for testing only convex polygons (when possible,
 * this is faster). */
/* #define CONVEX */

/* Define HYBRID to compile triangle fan test for CONVEX with exterior edges
 * meaning an early exit (faster - recommended).
 */
/* #define HYBRID */

/* Define DISPLAY to display test triangle and test points on screen. */
/* #define DISPLAY */

/* Define RANDOM to randomize order of edges for exterior test (faster -
 * recommended). */
/* #define RANDOM */

/* Define SORT to sort triangle edges and areas for half-plane and Spackman
```

```
 * tests (faster - recommended). */
/* #define SORT */

/* Define WINDING if a non-zero winding number should be used as the criterion
 * for being inside the polygon.  Only used by the general crossings test and
 * Weiler test.  The winding number computed for each is the number of
 * counterclockwise loops the polygon makes around the point.
 */
/* #define WINDING */

/* ============================ System Related ============================== */

/* Define your own random number generator; change as needed. */
/* SRAN initializes random number generator, if needed. */
#define SRAN()          srand48(1)
/* RAN01 returns a double from [0..1) */
#define RAN01()         drand48()
double  drand48() ;

/* =========== Half-Plane stuff ================================================ */

typedef struct {
    double      vx, vy, c ;     /* edge equation  vx*X + vy*Y + c = 0 */
#ifdef CONVEX
#ifdef HYBRID
    int         ext_flag ;      /* TRUE == exterior edge of polygon */
#endif
#endif
} PlaneSet, *pPlaneSet ;


#ifdef  CONVEX
#ifdef  SORT
/* Size sorting structure for half-planes */
typedef struct {
    double      size ;
    pPlaneSet   pps ;
} SizePlanePair, *pSizePlanePair ;
#endif
#endif


#ifdef  CONVEX
pPlaneSet       ExteriorSetup() ;
void    ExteriorCleanup() ;
#ifdef  SORT
int     CompareSizePlanePairs() ;
#endif
#endif
pPlaneSet       PlaneSetup() ;
void    PlaneCleanup() ;
```

ptinpoly.c

```
/* ptinpoly.c - point in polygon inside/outside code.

   by Eric Haines, 3D/Eye Inc, erich@eye.com

   This code contains the following algorithms:
        crossings - count the crossing made by a ray from the test point
        half-plane testing - test triangle fan using half-space planes
        exterior test - for convex polygons, check exterior of polygon
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ptinpoly.h"

#define X       0
#define Y       1
#define TRUE    1

#ifndef HUGE
#define HUGE    1.79769313486232e+308
#endif

#define MALLOC_CHECK( a )       if ( !(a) ) {                               \
                                    fprintf( stderr, "out of memory\n" ) ; \
                                    exit(1) ;                               \
                                }

/* ======= Crossings algorithm ============================================ */

/* Shoot a test ray along +X axis.  The strategy, from MacMartin, is to
 * compare vertex Y values to the testing point's Y and quickly discard
 * edges which are entirely to one side of the test ray.
 *
 * Input 2D polygon _pgon_ with _numverts_ number of vertices and test point
 * _point_, returns 1 if inside, 0 if outside.  WINDING and CONVEX can be
 * defined for this test.
 */
int CrossingsTest( pgon, numverts, point )
double  pgon[][2] ;
int     numverts ;
double  point[2] ;
{
#ifdef  WINDING
register int    crossings ;
#endif
register int    j, yflag0, yflag1, inside_flag, xflag0 ;
register double ty, tx, *vtx0, *vtx1 ;
#ifdef  CONVEX
```

```
register int    line_flag ;
#endif

    tx = point[X] ;
    ty = point[Y] ;

    vtx0 = pgon[numverts-1] ;
    /* get test bit for above/below X axis */
    yflag0 = ( vtx0[Y] >= ty ) ;
    vtx1 = pgon[0] ;

#ifdef  WINDING
    crossings = 0 ;
#else
    inside_flag = 0 ;
#endif
#ifdef  CONVEX
    line_flag = 0 ;
#endif
    for ( j = numverts+1 ; --j ; ) {

        yflag1 = ( vtx1[Y] >= ty ) ;
        /* check if endpoints straddle (are on opposite sides) of X axis
         * (i.e., the Y's differ); if so, +X ray could intersect this edge.
         */
        if ( yflag0 != yflag1 ) {
            xflag0 = ( vtx0[X] >= tx ) ;
            /* check if endpoints are on same side of the Y axis (i.e., X's
             * are the same); if so, it's easy to test if edge hits or misses.
             */
            if ( xflag0 == ( vtx1[X] >= tx ) ) {

                /* if edge's X values both right of the point, must hit */
#ifdef  WINDING
                if ( xflag0 ) crossings += ( yflag0 ? -1 : 1 ) ;
#else
                if ( xflag0 ) inside_flag = !inside_flag ;
#endif
            } else {
                /* compute intersection of pgon segment with +X ray, note
                 * if >= point's X; if so, the ray hits it.
                 */
                if ( (vtx1[X] - (vtx1[Y]-ty)*
                     ( vtx0[X]-vtx1[X])/(vtx0[Y]-vtx1[Y])) >= tx ) {
#ifdef  WINDING
                    crossings += ( yflag0 ? -1 : 1 ) ;
#else
                    inside_flag = !inside_flag ;
#endif
                }
            }
#ifdef  CONVEX
            /* if this is second edge hit, then done testing */
```

```
            if ( line_flag ) goto Exit ;

            /* Note that one edge has been hit by the ray's line. */
            line_flag = TRUE ;
#endif
        }

        /* Move to next pair of vertices, retaining info as possible. */
        yflag0 = yflag1 ;
        vtx0 = vtx1 ;
        vtx1 += 2 ;
    }
#ifdef  CONVEX
    Exit: ;
#endif
#ifdef  WINDING
    /* Test if crossings is not zero. */
    inside_flag = (crossings != 0) ;
#endif

    return( inside_flag ) ;
}


/* ======= Triangle half-plane algorithm ================================== */

/* Split the polygon into a fan of triangles and for each triangle test if
 * the point is inside of the three half-planes formed by the triangle's edges.
 *
 * Call setup with 2D polygon _pgon_ with _numverts_ number of vertices,
 * which returns a pointer to a plane set array.
 * Call testing procedure with a pointer to this array, _numverts_, and
 * test point _point_, returns 1 if inside, 0 if outside.
 * Call cleanup with pointer to plane set array to free space.
 *
 * SORT and CONVEX can be defined for this test.
 */

/* Split polygons along set of x axes - call preprocess once. */
pPlaneSet       PlaneSetup( pgon, numverts )
double  pgon[][2] ;
int     numverts ;
{
int     i, p1, p2 ;
double  tx, ty, vx0, vy0 ;
pPlaneSet       pps, pps_return ;
#ifdef  SORT
double  len[3], len_temp ;
int     j ;
PlaneSet        ps_temp ;
#ifdef  CONVEX
pPlaneSet       pps_new ;
pSizePlanePair p_size_pair ;
```

```
#endif
#endif

    pps = pps_return =
            (pPlaneSet)malloc( 3 * (numverts-2) * sizeof( PlaneSet )) ;
    MALLOC_CHECK( pps ) ;
#ifdef  CONVEX
#ifdef  SORT
    p_size_pair =
        (pSizePlanePair)malloc( (numverts-2) * sizeof( SizePlanePair ) ) ;
    MALLOC_CHECK( p_size_pair ) ;
#endif
#endif

    vx0 = pgon[0][X] ;
    vy0 = pgon[0][Y] ;

    for ( p1 = 1, p2 = 2 ; p2 < numverts ; p1++, p2++ ) {
        pps->vx = vy0 - pgon[p1][Y] ;
        pps->vy = pgon[p1][X] - vx0 ;
        pps->c = pps->vx * vx0 + pps->vy * vy0 ;
#ifdef  SORT
        len[0] = pps->vx * pps->vx + pps->vy * pps->vy ;
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = ( p1 == 1 ) ;
#endif
        /* Sort triangles by areas, so compute (twice) the area here. */
        p_size_pair[p1-1].pps = pps ;
        p_size_pair[p1-1].size =
                        ( pgon[0][X] * pgon[p1][Y] ) +
                        ( pgon[p1][X] * pgon[p2][Y] ) +
                        ( pgon[p2][X] * pgon[0][Y] ) -
                        ( pgon[p1][X] * pgon[0][Y] ) -
                        ( pgon[p2][X] * pgon[p1][Y] ) -
                        ( pgon[0][X] * pgon[p2][Y] ) ;
#endif
#endif
        pps++ ;
        pps->vx = pgon[p1][Y] - pgon[p2][Y] ;
        pps->vy = pgon[p2][X] - pgon[p1][X] ;
        pps->c = pps->vx * pgon[p1][X] + pps->vy * pgon[p1][Y] ;
#ifdef  SORT
        len[1] = pps->vx * pps->vx + pps->vy * pps->vy ;
#endif
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = TRUE ;
#endif
#endif
        pps++ ;
        pps->vx = pgon[p2][Y] - vy0 ;
        pps->vy = vx0 - pgon[p2][X] ;
```

```
        pps->c = pps->vx * pgon[p2][X] + pps->vy * pgon[p2][Y] ;
#ifdef  SORT
        len[2] = pps->vx * pps->vx + pps->vy * pps->vy ;
#endif
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = ( p2 == numverts-1 ) ;
#endif
#endif

        /* Find an average point that must be inside of the triangle. */
        tx = ( vx0 + pgon[p1][X] + pgon[p2][X] ) / 3.0 ;
        ty = ( vy0 + pgon[p1][Y] + pgon[p2][Y] ) / 3.0 ;

        /* Check sense and reverse if test point is not thought to be inside
         * first triangle.
         */
        if ( pps->vx * tx + pps->vy * ty >= pps->c ) {
            /* back up to start of plane set */
            pps -= 2 ;
            /* Point is thought to be outside, so reverse sense of edge
             * normals so that it is correctly considered inside.
             */
            for ( i = 0 ; i < 3 ; i++ ) {
                pps->vx = -pps->vx ;
                pps->vy = -pps->vy ;
                pps->c  = -pps->c ;
                pps++ ;
            }
        } else {
            pps++ ;
        }

#ifdef  SORT
        /* Sort the planes based on the edge lengths. */
        pps -= 3 ;
        for ( i = 0 ; i < 2 ; i++ ) {
            for ( j = i+1 ; j < 3 ; j++ ) {
                if ( len[i] < len[j] ) {
                    ps_temp = pps[i] ;
                    pps[i] = pps[j] ;
                    pps[j] = ps_temp ;
                    len_temp = len[i] ;
                    len[i] = len[j] ;
                    len[j] = len_temp ;
                }
            }
        }
        pps += 3 ;
#endif
    }

#ifdef  CONVEX
```

```
#ifdef  SORT
    /* Sort the triangles based on their areas. */
    qsort( p_size_pair, numverts-2,
            sizeof( SizePlanePair ), CompareSizePlanePairs ) ;

    /* Make the plane sets match the sorted order. */
    for ( i = 0, pps = pps_return
        ; i < numverts-2
        ; i++ ) {

        pps_new = p_size_pair[i].pps ;
        for ( j = 0 ; j < 3 ; j++, pps++, pps_new++ ) {
            ps_temp = *pps ;
            *pps = *pps_new ;
            *pps_new = ps_temp ;
        }
    }
    free( p_size_pair ) ;
#endif
#endif

    return( pps_return ) ;
}

#ifdef  CONVEX
#ifdef  SORT
int CompareSizePlanePairs( p_sp0, p_sp1 )
pSizePlanePair  p_sp0, p_sp1 ;
{
    if ( p_sp0->size == p_sp1->size ) {
        return( 0 ) ;
    } else {
        return( p_sp0->size > p_sp1->size ? -1 : 1 ) ;
    }
}
#endif
#endif


/* Check point for inside of three "planes" formed by triangle edges. */
int PlaneTest( p_plane_set, numverts, point )
pPlaneSet       p_plane_set ;
int     numverts ;
double  point[2] ;
{
register pPlaneSet      ps ;
register int    p2 ;
#ifndef CONVEX
register int    inside_flag ;
#endif
register double tx, ty ;

    tx = point[X] ;
```

```
    ty = point[Y] ;

#ifndef CONVEX
    inside_flag = 0 ;
#endif

    for ( ps = p_plane_set, p2 = numverts-1 ; --p2 ; ) {

        if ( ps->vx * tx + ps->vy * ty < ps->c ) {
            ps++ ;
            if ( ps->vx * tx + ps->vy * ty < ps->c ) {
                ps++ ;
                /* Note: we make the third edge have a slightly different
                 * equality condition, since this third edge is in fact
                 * the next triangle's first edge.  Not fool-proof, but
                 * it doesn't hurt (better would be to keep track of the
                 * triangle's area sign so we would know which kind of
                 * triangle this is).  Note that edge sorting nullifies
                 * this special inequality, too.
                 */
                if ( ps->vx * tx + ps->vy * ty <= ps->c ) {
                    /* point is inside polygon */
#ifdef CONVEX
                    return( 1 ) ;
#else
                    inside_flag = !inside_flag ;
#endif
                }
#ifdef  CONVEX
#ifdef  HYBRID
                /* check if outside exterior edge */
                else if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif

                ps++ ;
            } else {
#ifdef  CONVEX
#ifdef  HYBRID
                /* check if outside exterior edge */
                if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif

                /* get past last two plane tests */
                ps += 2 ;
            }
        } else {
#ifdef  CONVEX
#ifdef  HYBRID
            /* check if outside exterior edge */
            if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif

            /* get past all three plane tests */
```

```
            ps += 3 ;
        }
    }

#ifdef CONVEX
    /* for convex, if we make it to here, all triangles were missed */
    return( 0 ) ;
#else
    return( inside_flag ) ;
#endif
}

void PlaneCleanup( p_plane_set )
pPlaneSet       p_plane_set ;
{
    free( p_plane_set ) ;
}


/* ======= Exterior (convex only) algorithm ================================ */

/* Test the edges of the convex polygon against the point.  If the point is
 * outside any edge, the point is outside the polygon.
 *
 * Call setup with 2D polygon _pgon_ with _numverts_ number of vertices,
 * which returns a pointer to a plane set array.
 * Call testing procedure with a pointer to this array, _numverts_, and
 * test point _point_, returns 1 if inside, 0 if outside.
 * Call cleanup with pointer to plane set array to free space.
 *
 * RANDOM can be defined for this test.
 * CONVEX must be defined for this test; it is not usable for general polygons.
 */

#ifdef  CONVEX
/* make exterior plane set */
pPlaneSet ExteriorSetup( pgon, numverts )
double  pgon[][2] ;
int     numverts ;
{
int     p1, p2, flip_edge ;
pPlaneSet       pps, pps_return ;
#ifdef  RANDOM
int     i, ind ;
PlaneSet        ps_temp ;
#endif

    pps = pps_return =
            (pPlaneSet)malloc( numverts * sizeof( PlaneSet )) ;
    MALLOC_CHECK( pps ) ;

    /* take cross product of vertex to find handedness */
    flip_edge = (pgon[0][X] - pgon[1][X]) * (pgon[1][Y] - pgon[2][Y] ) >
```

```
                (pgon[0][Y] - pgon[1][Y]) * (pgon[1][X] - pgon[2][X] ) ;

    /* Generate half-plane boundary equations now for faster testing later.
     * vx & vy are the edge's normal, c is the offset from the origin.
     */
    for ( p1 = numverts-1, p2 = 0 ; p2 < numverts ; p1 = p2, p2++, pps++ ) {
        pps->vx = pgon[p1][Y] - pgon[p2][Y] ;
        pps->vy = pgon[p2][X] - pgon[p1][X] ;
        pps->c = pps->vx * pgon[p1][X] + pps->vy * pgon[p1][Y] ;

        /* check sense and reverse plane edge if need be */
        if ( flip_edge ) {
            pps->vx = -pps->vx ;
            pps->vy = -pps->vy ;
            pps->c  = -pps->c ;
        }
    }

#ifdef  RANDOM
    /* Randomize the order of the edges to improve chance of early out. */
    /* There are better orders, but the default order is the worst. */
    for ( i = 0, pps = pps_return
        ; i < numverts
        ; i++ ) {

        ind = (int)(RAN01() * numverts ) ;
        if ( ( ind < 0 ) || ( ind >= numverts ) ) {
            fprintf( stderr,
                "Yikes, the random number generator is returning values\n" ) ;
            fprintf( stderr,
                "outside the range [0.0,1.0), so please fix the code!\n" ) ;
            ind = 0 ;
        }

        /* swap edges */
        ps_temp = *pps ;
        *pps = pps_return[ind] ;
        pps_return[ind] = ps_temp ;
    }
#endif
    return( pps_return ) ;
}

/* Check point for outside of all planes. */
/* Note that we don't need "pgon", since it's been processed into
 * its corresponding PlaneSet.
 */
int ExteriorTest( p_ext_set, numverts, point )
pPlaneSet       p_ext_set ;
int     numverts ;
double  point[2] ;
{
register PlaneSet       *pps ;
```

```
register int    p0 ;
register double tx, ty ;
int     inside_flag ;

    tx = point[X] ;
    ty = point[Y] ;

    for ( p0 = numverts+1, pps = p_ext_set ; --p0 ; pps++ ) {

        /* Test if the point is outside this edge. */
        if ( pps->vx * tx + pps->vy * ty > pps->c ) {
            return( 0 ) ;
        }
    }
    /* If we make it to here, we were inside all edges. */
    return( 1 ) ;
}

void ExteriorCleanup( p_ext_set )
pPlaneSet       p_ext_set ;
{
    free( p_ext_set ) ;
}
#endif
```

◇ Bibliography ◇

(Antonio 1992) Franklin Antonio. Faster line segment intersection. In David Kirk, editor, *Graphics Gems III*, pages 199–202. Academic Press, Boston, 1992.

(Badouel 1990) Didier Badouel. An efficient ray-polygon intersection. In Andrew Glassner, editor, *Graphics Gems*, pages 390–393. Academic Press, Boston, 1990.

(Berlin 1985) Jr. Edwin P. Berlin. Efficiency considerations in image synthesis. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, July 1985.

(Glassner 1989) A. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press, London, 1989.

(Green 1993) Chris Green. Simple, fast triangle intersection. *Ray Tracing News* 6(1), E-mail edition, anonymous ftp from princeton.edu:/pub/Graphics/RTNews, 1993.

(Hacker 1962) R. Hacker. Certification of algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:606, 1962.

(Haines 1992) Eric Haines, editor. Fastest point in polygon test. *Ray Tracing News* 5(3), E-mail edition, anonymous ftp from princeton.edu:/pub/Graphics/RTNews, 1992.

(Hanrahan and Haeberli 1990) Pat Hanrahan and Paul Haeberli. Direct WYSIWYG painting and texturing on 3d shapes. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):215–223, August 1990.

(Preparata and Shamos 1985) F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

(Salesin and Stolfi 1989) David Salesin and Jorge Stolfi. The ZZ-buffer: A simple and efficient rendering algorithm with reliable antialiasing. In *Proc. 2nd Intl. Conf. on Computer Graphics (PIXIM '89)*, pages 451–465, Paris, France, 1989.

(Schorn and Fisher 1994) Peter Schorn and Frederick Fisher. Testing the convexity of a polygon. In Paul Heckbert, editor, *Graphics Gems IV*, 7–15. Academic Press, Boston, 1994.

(Shimrat 1962) M. Shimrat. Algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:434, 1962.

(Spackman 1993) John Spackman. Simple, fast triangle intersection, part ii. *Ray Tracing News* 6(2), E-mail edition, anonymous ftp from princeton.edu:/pub/Graphics/RTNews, 1993.

(Weiler 1994) Kevin Weiler. An incremental angle point in polygon test. In Paul Heckbert, editor, *Graphics Gems IV*, 16–23. Academic Press, Boston, 1994.

(Woo 1992) Andrew Woo. Ray tracing polygons using spatial subdivision. In *Proceedings of Graphics Interface '92*, pages 184–191, May 1992.

(Worley and Haines 1993) Steve Worley and Eric Haines. Bounding areas for ray/polygon intersection. *Ray Tracing News* 6(1), E-mail edition, anonymous ftp from princeton.edu:/pub/Graphics/RTNews, 1993.